Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 182 (2007) 187–200
www.elsevier.com/locate/entcs

Component Updates as a Boolean Optimization Problem

Alexander Stuckenholz[1]

Department of Data Processing Technologies
FernUniversität in Hagen
Hagen, Germany

Abstract

Component updates always bear the risk of negatively influencing the operativeness of software systems. Due to improper combinations of component versions, dependencies may break. In practise this often turns out to be due to missing or incompatible interfaces and signatures (syntactical interface) but may also be caused by changes in behavior or quality. In this paper we model the problem of finding a well-configured system consisting of multiple component versions as a Boolean Optimization Problem. To achieve this, we introduce objective functions and constraints that lead to most recent, minimal systems and use Branch-and-Bound to restrict the search space.

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 often causing breakdowns and sleepless nights. But sometimes parts like components have to be replaced by newer versions because of serious security holes, functional limitations or quality improvements.

By updating a system, thus replacing components by newer versions, dependencies may be added or removed. The system changes over time which is called *system evolution*. Most recent research investigating the source of defect of object oriented software systems (cf. [2]) indicate that missing components or wrong component versions are the most frequent reasons for configuration problems.

Nevertheless, there are almost no tools that are able to prevent these situations. Tools for automated configurations which combine compositional reasoning with

[1] Email: Alexander.Stuckenholz@FernUni-Hagen.de

doi:10.1016/j.entcs.2006.09.039

automated versioning and dependency analysis (cf. [17]) are missing in configuration management.

In this paper, we sketch a mechanism to construct well-configured component based systems by combining available component versions and respecting constraints like favoring recent versions, minimizing the count of components in a system and minimizing the count of replacements between updates.

The system is specially designed for situations in which new component versions are not exact substitutes of their predecessors, but introducing new dependencies or change parts, on which other components in the system rely. We assume that in such situations, a balanced configuration can be found by a smart combination of available component versions. This way, we do not require backward compatibility between component versions, which is the basis of most packaging systems in this area, but allow arbitrary evolution of 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 mentions related approaches like Linux packaging to solve the update-problem and the application of optimization methods. Section 2 introduces a simple component based system, which we will use as a running example for the rest of the paper. In section 3 we establish objective functions and constraints for the problem in the normal form of a Boolean Optimization Problem. Furthermore we mention the complexity of the combinatorial problem for finding well-configured system configurations. Section 4 addresses the search for solutions by calculating upper and lower bounds and their usage in branching the search-tree by a Branch-and-Bound algorithm. Section 5 mentions the real-world projects in which these methods are currently evaluated and presents the results attained so far. Finally section 6 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. [3]). The main goal is to reduce development costs, shorten the time-to market by massive reuse and to increase product quality by frequently utilized software artifacts.

At the same time, even today many software developers spend their time 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 compared to conventional software architectures. Direct and indirect dependency relations between the components of a system arise when software components are reused in several application at the same time. Figure 1 clarifies this structural difference between monolithic in contrast to component based architectures.

In conventional, monolithic software architectures, all functions required by 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, as all applications of the platform have to be analyzed for their usage of that or similar

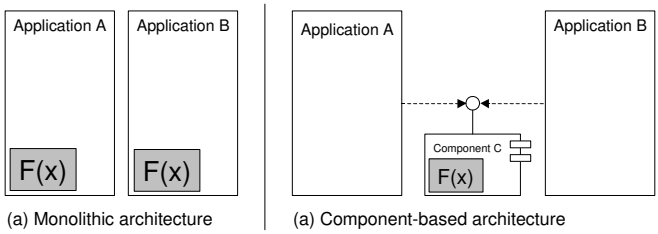

Fig. 1. Code-redundancy and coupling with and without the usage of component based architectures

functions.

Such situations have precarious impacts to the maintainability of software systems. In december 2004 a dangerous security hole was detected in the GDI+ component, which renders JPEG files on Windows platforms (see [4]). As multiple applications deployed the component as a Dynamic Linked Library (DLL) locally, so that it was installed in parallel in many cases, the security hole could not be fixed by simply substituting one centrally installed component.

CBSD tries to avoid such problems. Here the commonly used functions are bundled and delivered as components, reused by multiple applications in the system. 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 extent. If a commonly used component (like C in figure 1) has to be substituted, this can lead to incompatibilities, because the interface of the substitute has been changed. Such changes must not necessarily impact all dependent components, but only those that make use of the changed or removed interface parts.

One real-world example of such situations is the so called DLL-Hell (cf. [5,14]). The setup procedure of different applications on Windows platforms overwrote commonly used components by their own, not necessarily newer versions. Accordingly, the new applications worked as expected, but some of the already installed programs behaved unexpected, if operating at all.

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

A conservative approach simply prohibits that components or interfaces, once installed in a system, may change at all [2]. New versions of components are considered as completely new components, and new interfaces supplement predecessors without replacing them. Thus, different evolutionary states of the same component exist on a platform in parallel, which leads to the aforementioned drawbacks. In case of an update, the new version is installed additionally. Therefore, an update does not influence the operativeness of the system, but it is difficult to impossible to replace erroneous components by other versions. The capability to change systems subsequently, usually regarded as a strength of component based architectures, is

[2] The component model COM forbids interface evolution at all (cf. [15]

hindered or even prevented by such approaches.

1.2 Related Work

There are a couple of approaches, which are able to check the conformity between different evolutionary states of a component and automatically detect incompatible changes. The most promising systems in this area have been compared by us in [17]. Depending on the system, more or less contract levels (syntax, behavior, synchronization, quality) are included in the analysis, if a certain component is in the position to replace another completely. The upgrade is allowed, if the conformance check yields the substitutability for all components, the administrator wants to replace.

In situations in which one or more components are not exact substitutes of their predecessors, these 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.

Such a perspective is also presented by the EDOS Project [3] which especially aims at the distribution of *Free Open Source Software* (FOSS) and the quality assurance of system configurations.

The problem, we target at, is to find a valid combination of component versions, so that 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 mathematics, but still belong to the most complex problems at all. The graph coloring problem described in [16] shows another example of this, 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 introduction of smart heuristics to prune the search space, solutions can be found very efficiently in average cases.

In open source operating systems (Linux flavors) there is also another notion of a component, namely a software package (like TeX, the C compiler etc.) which can be installed, updated, or removed. Some packages require other packages or wont work, when others are installed. Linux distributions provide sophisticated management tools based on package formats, like the Debian [8] 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 may lead to inconsistent system states in the worst-case.

Both systems and their tools rest upon different heuristics that are pragmatic but minimize the solution space for finding compatible systems. In case of an update, tools like *apt* or *rpm* will always try to install newest packages only. If a well-defined system state could only be achieved by moving some packages back to an older version, these tools are not the best choice. Furthermore it is accepted that

[3] http://www.edos-project.org

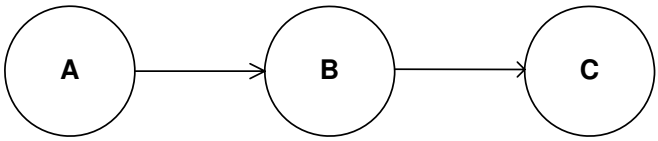

Fig. 2. Component dependency graph of a system consisting of three components.

deb- or *rpm*-packages are always backward compatible to older versions. Packages that would violate this rule are simply renamed (e.g. with the new major-revision-number in the package name) and from then on exist in parallel with older version in the package-repositories.

Our mechanisms for finding best system states leave those indirections behind and include all potential candidates in form of component versions into the search for compatible system states.

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

2 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. In addition, the front-end A shows the results of the rendering engine B to the user by invoking its services. Thus, we have dependency relations between the three components in the system. The usual way to visualize these dependencies is to draw them in a dependency graph [4]. Figure 2 shows the dependency graph for the example introduced above. The components are illustrated as nodes and the dependencies as directed edges.

Because of bug-fixing, performance optimization or feature enhancements new versions of the components come to existence over time. In worst case these components are not simple substitutes for the old versions but introduce new dependencies or remove old interfaces. Each of the versions may introduce different dependencies to other component versions. Finding well-configured systems is, if even possible, a non-trivial task. This is especially true in cases with large quantities of components and component versions or frequent incompatibilities.

To anticipate component evolution and the requirements of different versions the version reachability graph has been introduced in [18]. 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 of a single component denote an alternative, e.g. a component version A_1 requires

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

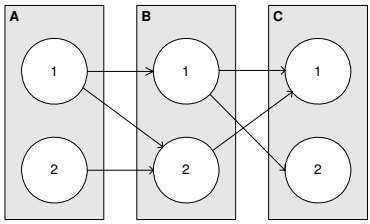

Fig. 3. Version reachability graph of a simple system consisting of three components and two versions each.

the existence of either a component version B_1 or B_2. Figure 3 shows the according version reachability graph, assuming that for each of the three components in the example mentioned above, two versions with different dependencies are available.

Within this simple example it is easy to find the four well-configured 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 count of required component swaps. Bigger systems require an automated method to solve this problem.

3 Modeling the problem as a Boolean Optimization Problem

In order to use standardized methods with known complexity properties we will model the described 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. [13], p. 6).

3.1 Objective Functions

In order to get the most recent system, we have to reward the usage of component versions with highest version numbers during the system synthesis, which has to precipitate in a objective function of the BOP-problem.

To respect the up-to-dateness of the components 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 schemas 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* schema [5]. 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 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 values can only be zero or one, determine, wether the according component version will be installed in the system at the end. For our example, the function is defined as follows:

$$x_{A_1} \cdot v(A_1) + x_{A_2} \cdot v(A_2) + x_{B_1} \cdot v(B_1) + ... + x_{C_2} \cdot v(C_2) = T_1(x_i)\ (max) \quad (2)$$

Another target is to get a minimal system with the smallest possible amount of components. Especially against the background of the objective function defined above, we have to punish the usage of otherwise unnecessary components in a system which increases the version sum. Therefore, we have to define another objective function for ensuring a minimal system regarding the count of used components by accumulating the sum over all decision variables x_{R_i}. For the example this leads to:

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

Finally we want to embrace the required count of component swaps between the target system we look for and a current system. Our goal is to keep the number of swaps as small as possible because every component substitution takes time, in which the system is usually partly or completely unavailable. For our example, we assume that our current system consists of the components A_1, B_1 and C_2. To calculate the count of swappings we sum the decision variables for all versions that are not already used in the system. For the example this results to:

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

We now have multiple objective functions against which we want to optimize our solution. It is provable 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 3.2). In a best case, the relaxation is also an integer solution. Otherwise we found a good upper bound for a search (see section 4.1). In case of multiple objective functions, the optimal solution is located on the edge of the n-polyhedron between the vertices of the single optimal solutions.

In order to combine the target functions defined afore, we could create a new objective function T by superposing the sub-goals and weighting them with weighting factors w_i. But in fact it is difficult to find constant weighting factors influencing the optimization-process in the favored way. Hence we solve the problem against

[5] The *major-minor-build* and related versioning schemas 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.

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. [19], p. 118).

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 prior calculated results, we can influence the optimization process in a smooth way.

3.2 Constraints

So far, we only defined characteristics of optimal solutions, but the state space in which the search is allowed 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 the 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. [17]). 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 fixing important security problems by replacing a single, centrally installed component almost impossible.

In our example, we have to ensure that if A_1 is installed, it is not allowed that A_2 is installed, too. Such requirements can be expressed in propositional logic in an easy way. 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 [19] 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 transformed into an own inequation, i.e. sum ≥ 1. In this sum, every literal X is replaced by its according 0/1-variable and every negative literal $\neg X$ is replaced by $(1 - x)$.

For our example we derive:

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

After 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.

Again, the dependencies can be expressed in propositional logic easily:

$$\begin{aligned}X_{A_1} &\rightarrow X_{B_1} \vee X_{B_2}\\ X_{A_2} &\rightarrow X_{B_2}\\ X_{B_1} &\rightarrow X_{C_1} \vee X_{C_2}\\ X_{B_2} &\rightarrow X_{C_1}\end{aligned} \tag{7}$$

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 results in:

$$\begin{aligned}(1 - x_{A_1}) + x_{B_1} + x_{B_2} &\geq 1\\ (1 - x_{A_2}) + x_{B_2} &\geq 1\\ (1 - x_{B_1}) + x_{C_1} + x_{C_2} &\geq 1\\ (1 - x_{B_2}) + x_{C_1} &\geq 1\end{aligned} \tag{8}$$

Including 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

$$A = \begin{pmatrix} -1 & -1 & & & & & \\ & & -1 & -1 & & & \\ & & & & -1 & -1 & \\ -1 & & 1 & 1 & & & \\ & -1 & & 1 & & & \\ & & -1 & & 1 & 1 & \\ & & & -1 & 1 & & \end{pmatrix}, b = \begin{pmatrix} -1 \\ -1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{9}$$

3.3 Complexity

The constraints in propositional logic specified before, 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 and-conjunctions. For the example we derive:

$$\begin{aligned}&(\neg X_{A_1} \vee \neg X_{A_2}) \wedge (\neg X_{B_1} \vee \neg X_{B_2}) \wedge \\ &(\neg X_{C_1} \vee \neg X_{C_2}) \wedge (\neg X_{A_1} \vee X_{B_1} \vee X_{B_2}) \wedge \\ &(\neg X_{A_2} \vee X_{B_2}) \wedge (\neg X_{B_1} \vee X_{C_1} \vee X_{C_2}) \wedge \\ &(\neg X_{B_2} \vee X_{C_1})\end{aligned} \tag{10}$$

The problem to decide, for a given formula, whether it is satisfiable or not is called SAT which is proven to be an NP-complete problem [7]. We renounce to drive a formal prove that our problem is NP-complete here, but we point out the exponential coherence between the count of components and the number of possible system configurations. With only 132 components and 3 versions each, $3 \cdot 10^{79}$ system configurations are possible.

In a very similar context, [6] shows that the decision, whether a software package can be installed in a system is an NP-complete problem by reducing it to 3-SAT.

4 Solving the problem

So far, we specified constraints, which allows the evaluation of results for validity. The next step is to create a mechanism to find such results, namely valid system configurations, by searching through the state space of component versions. As the state space grows exponentially with the count of component versions, it is not reasonable to use a brute force approach and just trying all possible combinations one after another.

Hence we need a mechanism that cuts non valuable branches off the search space, reducing the search complexity to a minimum. Branch-and-Bound (see [20]) is one representative of such mechanisms. The core 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. In combination with a measurement for unexpanded branches, we can reduce the search space to a minimum.

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

Using miscalculated bounds or bad measurement for the branches not expanded yet, raises the risk to cut off states that may contain solutions or even the optimum.

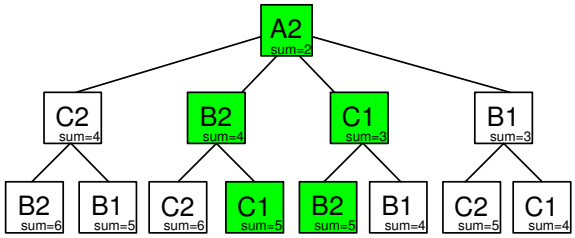

Fig. 4. The state space of the search from the example. Sum denotes the version sum in the current search branch.

4.1 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, for which we can use the model of a Boolean Optimization Problem introduced before.

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. Thus the search for an optimal solution can be reasoned by a geometric perception. In principle, an optimal solution can be found on vertices of that polyhedron. Simplex algorithms (cf. [19]) move from vertex to vertex via the edges of the polyhedron to find the optimal solution while interior point algorithms (cf. [13]) 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. There are interior point algorithms (e.g. [9]) that solve the linear optimization problem in worst-case polynomial time while the existence of a worst-case polynomial running simplex algorithm remains an unsolved problem[6]. In practice simplex algorithm 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 constraints, we derive continuous values for the decision variables and an upper-bounds estimation. If the calculated bound in a certain search state is less than a bound calculated before, then we do not have to expand this branch.

4.2 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.

[6] (i) For a large class of simplex algorithms it is known that they have exponential worst-case complexity [11]. (ii) A randomized polynomial-time simplex algorithm is known [10].

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 expand. Then we either have an optimal solution or no solution at all.

We can minimize the search tree, if we do not just pick arbitrary components from the repository, but sort the candidates 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 *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 done by sorting the PriorityQueue of component versions by their version identifiers in descending order. This strategy is know as *best first search*.

Figure 4 shows the state space of the combinatorical problem as a tree, which will be traversed by depth-first search, described in [12]. 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 highlighted. As every subnode of the tree represents independent combinatorial subproblems, the Branch-and-Bound algorithm is particularly suitable.

In section 3.1 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 other approach that would be able to generate such a solution.

5 Evaluation

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

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

A signature matcher has been defined in first order logic and implemented in Prolog. With the help of SWI-Prolog, the matcher has been integrated into our tool *Componentor*, written in Java. The required dependencies can thus be generated automatically.

These dependencies can then be utilized to first check a component's substi-

tutability of an old version against a new one. If substitution is not possible, we can use *Componentor*, to analyze the reasons of this conflict and try to find valid combinations using the sketched optimization mechanisms. If all this fails, the tool is able to generate advices, how different components could be changed in order to ensure compatibility again.

We currently working 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 form direct dependencies. The more generic software becomes, the less applicable are our methods. Dependencies that arise from dynamic instantiation and the use of reflection are not seizable by static code analysis. Such dependencies could only be discovered by dynamic analysis, e.g. during unit testing.

6 Summary and Prospects

In the previous sections we have 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-complete, so optimization mechanisms must be used to derive acceptable results in appropriate time. Therefore we modeled the problem as a Boolean Optimization Problem (BOP) and used Branch-and-Bound for cutting the search space.

The approach acquits from requirements like backward compatibility between different component versions but admits and supports unplanned component and system evolution. We are currently in the process of validating our results with real-world projects in the open-source scene which are often subject to unplanned evolution.

We have 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 heuristic methods like Greedy and Local Search to find configurations for such environments.

References

[1] Bailey, E. C., "Maximum RPM," Sams, 1997.

[2] Borner, L. et al., *Fehlerhäufigkeiten in objektorientierten Systemen: Basisauswertung einer Online-Umfrage*, Technical report, Arbeitsgruppe Software Systems Engineering, Institut für Informatik, Ruprecht-Karls-Universität Heidelberg (2006), last visited: 03/2006.
URL http://www-swe.informatik.uni-heidelberg.de/research/publications/SWEHD_TR2006_01.pdf

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

[4] Corporation, M., *Microsoft Security Bulletin MS04-028 - Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution (833987)*, last visited: 02/2006.
URL http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx

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

[6] Di Cosmo, R., B. Durak, X. Leroy, F. Mancinelli and J. Vouillon, *Maintaining large software distributions: new challenges from the FOSS era*, in: *Proceedings of the FRCSS 2006 workshop*, 2006, to appear in EASST Newsletter.
URL http://gallium.inria.fr/~xleroy/publi/edos-frcss06.pdf

[7] Garey, M. R. and D. S. Johnson, "Computers and Intractability; A Guide to the Theory of NP-Completeness," W. H. Freeman & Co., New York, NY, USA, 1990.

[8] Jackson, I. and C. Schwarz, *Debian policy manual* (1998), last visited: 12/2005.
URL http://www.debian.org/doc/debian-policy/

[9] Karmarkar, N., *A new polynomial-time algorithm for linear programming*, Combinatorica **4** (1984), pp. 373–395.

[10] Kelmer, J. A. and D. A. Spielman, *A randomized polynomial-time simplex algorithm for linear programming*, in: *Symposium on Theory of Computing*, 2006, to be published.

[11] Klee, V. and G. J. Minty, *How good is the simplex algorithm*, Inequalities (1972), pp. 159–175.

[12] Knuth, D. E., "The Art of Computer Programming," Addison-Wesley Professional, 1998.

[13] Martin, R. K., "Large Scale Linear and Integer Optimization - A Unified Approach," Kluwer Academic Publishers, 1999, 764 pp.

[14] Pratschner, S., *Simplifying Deployment and Solving DLL Hell with the .NET Framework*, Technical report, Microsoft Corporation (2001), last visited: 04/2004.
URL
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dplywithnet.asp

[15] Rogerson, D., "Inside COM - Microsofts Component Object Model," Microsoft Press, Redmond, Washington, 1997.

[16] Russell, S. J. and P. Norvig, "Artificial Intelligence: A Modern Approach," Prentice Hall, 2002, second edition.

[17] Stuckenholz, A., *Component Evolution and Versioning - State of the Art*, SIGSOFT Softw. Eng. Notes **30** (2005), p. 7.

[18] Stuckenholz, A. and O. Zwintzscher, *Compatible component upgrades through smart component swapping*, in: R. Reusner, J. Stafford and C. Szyperski, editors, *Architecting Systems with Trustworthy Components*, LNCS 3938 (2006).

[19] Suhl, L. and T. Mellouli, "Optimierungssysteme," Springer, 2005, (in German).

[20] Wolsey, L. A., "Integer Programming," Wiley-Interscience, 1998.