Assessing the Complexity of Upgrading Software Modules

Bram Schoenmakers, Niels van den Broek, Istvan Nagy
ASML Netherlands B.V.,
De Run 6501,
5504 DR, Veldhoven, The Netherlands
{bram.schoenmakers, niels.van.den.broek, istvan.nagy}@asml.com

Bogdan Vasilescu, Alexander Serebrenik
Technische Universiteit Eindhoven,
Den Dolech 2, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
{b.n.vasilescu, a.serebrenik}@tue.nl

***Abstract*—Modern software development frequently involves developing multiple codelines simultaneously. Improvements to one codeline should often be applied to other codelines as well, which is typically a time consuming and error-prone process. In order to reduce this (manual) effort, changes are applied to the system's modules and those affected modules are upgraded on the target system. This is a more coarse-grained approach than upgrading the affected files only. However, when a module is upgraded, one must make sure that all its dependencies are still satisfied. This paper proposes an approach to assess the ease of upgrading a software system. An algorithm was developed to compute the smallest set of upgrade dependencies, given the current version of a module and the version it has to be upgraded to. Furthermore, a visualization has been designed to explain why upgrading one module requires upgrading many additional modules. A case study has been performed at ASML to study the ease of upgrading the TwinScan software. The analysis shows that removing elements from interfaces leads to many additional upgrade dependencies. Moreover, based on our analysis we have formulated a number improvement suggestions such as a clear separation between the test code and the production code as well as an introduction of a structured process of symbols deprecation and removal.**

I. Introduction

Modern software development frequently involves multiple codelines (branches), being canonical sets of source files required to produce a specific software instance [1]. Codelines correspond, *e.g.,* to maintenance, release and development versions of a system, or to variants targeting different user groups or platforms. A typical scenario involves one mainline, containing the latest features and bug fixes and being continuously updated, and a number of customer codelines, containing different configurations being used by different customers. Such customer codelines need to be updated frequently, *e.g.,* to provide new features or bug fixes. The updates then translate to patches or entirely new releases of the software, which are shipped to customers and have to be integrated into their environments.

However, integrating the patches on the customer-side (referred to as upgrading) can become particularly costly, especially for safety-critical or real-time embedded software, that require extensive integration testing and complex initialisation routines. Consequently, large upgrades (*e.g.,* upgrading the entire codeline to a new release) are typically undesirable, and performing upgrades in a module-based fashion is preferred [2], [3]. This way, customers receive only the features they requested. Moreover, as less changes are introduced modular upgrades reduce testing and integration effort, as well as limit the risk of the system downtime.

Nonetheless, modules are often interdependent, hence upgrading a particular one may introduce the need to upgrade additional others, until all dependencies have been satisfied. In the worst case, upgrading one module may lead to upgrading the entire codeline, *e.g.,* if the change to this particular module introduces dependencies to newer versions of all other modules in the codeline, not yet present.

In this paper, we propose a framework to assess the difficulty of upgrading a software module. Our framework is in use by ASML Netherlands B.V., a large manufacturer of photolithography systems, and consists of three stages. First, we evaluate a module's independence, by looking at which functionality it provides through an interface, and which functionality it requires from other modules. Using this interface usage data, in Section II-C we present an algorithm computing a minimal set of modules which need to be upgraded (referred to as *upgrade dependencies*) in order to satisfy all dependencies in the codeline. Finally, in Section III we propose a visualisation which shows how the upgrade dependencies have evolved over time, and allows the user to study the nature of these upgrade dependencies.

II. Finding Minimal Upgrades

Given the upgrade of a module from one version to another, in this section we develop a formalization and an algorithm finding versions of all other modules such that all dependencies are satisfied, and the number of modules that have to be upgraded is minimal.

Dependencies arise as a result of some modules providing interfaces to be used by other modules. Each interface can disclose various program elements, which we call *symbols*, such as constants, enumerations, data types and functions. Each symbol can be provided by only one module.

A. Motivating Example

Consider the three system states in Figure 1. Suppose that the system is initially at $t = 1$ and we would like to upgrade

978-1-4799-2931-3/13 © 2013 IEEE

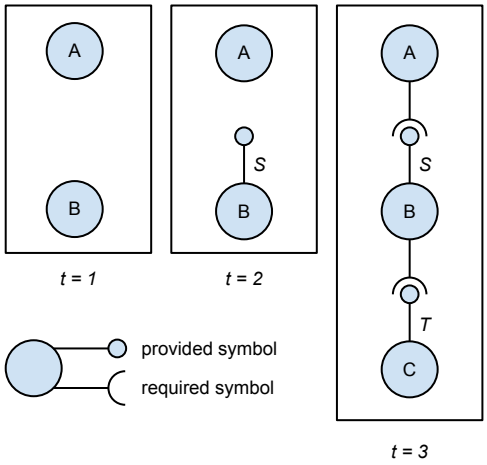


Fig. 1: An example system at three versions. Choosing module B at $t = 2$ yields the cheapest solution.

module A to the version at $t = 3$. Between these two versions, A has started to depend on symbol S of module B, which is not available at $t = 1$. Therefore, module B has to be upgraded as well. Here we have a choice: upgrade module B to the version at $t = 2$ or $t = 3$. Both versions provide the symbol S which is required by the latest version of A. However, B at $t = 3$ has a dependency on a new module C. So there are two valid ways to satisfy the dependencies: $\mathfrak{C}_1 = \{(A \to 3), (B \to 2), (C \to \bot)\}$, where $\bot$ represents absence of a module in the configuration, and $\mathfrak{C}_2 = \{(A \to 3), (B \to 3), (C \to 3)\}$.

However, recall that the system is at $t = 1$. Hence, to obtain $\mathfrak{C}_1$ we need to upgrade two modules (A itself and B) and to obtain $\mathfrak{C}_2$ we need three modules (A, B and C). $\mathfrak{C}_1$ is therefore preferred to $\mathfrak{C}_2$.

B. Formalization

Let $\mathbb{M}$ be a set of modules within a system. At each moment of time the system is composed from different versions of different modules: this composition is called a *configuration*. Formally, a configuration $\mathfrak{C}$ is a mapping from $\mathbb{M}$ to $\mathbb{V} \cup \{\bot\}$, where $\mathbb{V} \subset \mathbb{N}$ is the set of versions, $\mathbb{N}$ is the set of the natural numbers.

Given a module m that has to be updated from version i to version j, our goal is to find a valid configuration $\mathfrak{C}$ inducing a minimal upgrade such that $\mathfrak{C}(m) = j$. Configuration $\mathfrak{C}$ is *valid* if for any pair of modules n_1 and n_2, the symbols required by the version $\mathfrak{C}(n_1)$ of n_1 from n_2 are provided by the version $\mathfrak{C}(n_2)$ of n_2. We stress that we do not consider semantics of the symbols, and potential incompatibilities between the expectations of requiring module and providing module.

With $\mathfrak{C}$ and i as above we say that $\mathfrak{C}$ *induces an upgrade* $\mathfrak{u}(\mathfrak{C}, i)$ defined as $\{n | n \in \mathbb{M} \wedge \mathfrak{C}(n) > i \wedge \mathfrak{C}(n) \neq \bot\}$. Finally, we say that an upgrade $\mathfrak{u}(\mathfrak{C}, i)$ is *minimal* if for any valid $\mathfrak{C}'$ with $\mathfrak{C}'(m) = j$, $|\mathfrak{u}(\mathfrak{C}, i)| \leq |\mathfrak{u}(\mathfrak{C}', i)|$, where $|Z|$ denotes the cardinality of a set Z. We stress that a valid configuration inducing a minimal upgrade is not necessarily unique.

C. Algorithm

We sketch the algorithm searching for a configuration defined in Section II-B (a more detailed discussion, including the pseudo-code, can be found in [4]). As above, the module m is upgraded from version i to version j ($i < j$). The search then involves the following steps:

- **Init.** Create an initial configuration $\mathfrak{C}_0$, where all modules are at version i, except for module m, which is set to version j. Add $\mathfrak{C}_0$ to the stack $\mathfrak{S}$. Set *threshold* to ∞.
- **Loop.** Repeat until $\mathfrak{S} = \emptyset$.
  1) Pop a configuration $\mathfrak{C}$ from $\mathfrak{S}$.
  2) If $\mathfrak{C}$ is valid

     If $|\mathfrak{u}(\mathfrak{C}, i)| <$ *threshold*, remember $\mathfrak{C}$ and update *threshold* to $|\mathfrak{u}(\mathfrak{C}, i)|$.
  3) Otherwise $\mathfrak{C}$ is invalid, *i.e.,* there exists at least one module n_1 upgraded in $\mathfrak{C}$ (*i.e.,* $\mathfrak{C}(n_1) \neq \bot$ and $\mathfrak{C}(n_1) > i$) that requires a symbol σ from n_2, but either $\mathfrak{C}(n_2) = \bot$ or $\mathfrak{C}(n_2)$ does not provide σ. We call n_2 a *missing dependency*. For all missing dependencies (say k, $k < |\mathbb{M}|$), push to $\mathfrak{S}$ all configurations $\mathfrak{C}'$ which attempt to resolve the existing incompatibilities, *i.e.,* those based on all combinations of missing dependencies n and version numbers $\mathfrak{C}'(n)$ such that $\mathfrak{C}(n) < \mathfrak{C}'(n) \leq j$.
- Return the last $\mathfrak{C}$ remembered at Step 2. If no $\mathfrak{C}$ has been remembered, return "no upgrade possible".

D. Discussion

Requirement $\mathfrak{C}'(n) \leq j$ implies that each Step 3 can add only finitely many configurations. For k missing dependencies, the version range $(i, j]$ yields at most $(j - i)^k$ new configurations to be added. Moreover, $\mathfrak{C}(n) < \mathfrak{C}'(n)$ ensures that if new configurations are pushed on $\mathfrak{S}$, then at least one module has a version higher than in the configuration popped from $\mathfrak{S}$, thus the loop terminates. Hence, the loop implements an exhaustive search and upon its termination either a valid configuration inducing a minimal upgrade is found (*i.e.,* an optimal solution), or absence of a valid configuration is reported (*i.e.,* no solution).

Unfortunately, finding an optimal solution is hopelessly inefficient: each new configuration $\mathfrak{C}'$ may incur even more configurations if not all dependencies in $\mathfrak{C}'$ have been met, hence upgrading one module may require upgrading all other modules. In the worst case, the search has to analyze $|\mathbb{V}|^{|\mathbb{M}|-1}$ possible configurations. Moreover, generating new configurations at Step 3 has high memory demands: for example, if a configuration is invalid and requires 15 additional modules, each having eight versions to choose from, a total of $8^{15} \simeq 3.52 \times 10^{13}$ new configurations would be pushed on the stack. While we cannot hope to obtain a *theoretically* efficient algorithm as the problem is known to be NP-complete and might have exponentially many solutions [5], we prefer a *practically* efficient approach delivering an approximate solution based on a number of heuristics discussed below.

E. Enhancements

To address the inefficiency of the naive algorithm we have implemented a number of heuristics.

1) Improve Step 3:

a) Limit the Number of Added Configurations: The memory demands can be reduced by limiting the number of new configurations being added simultaneously in Step 3. As explained above, for k missing dependencies, the number of new configurations added simultaneously in one iteration can reach $(j-i)^k$. By limiting k, the newly added configurations will have missing dependencies, but these dependencies will be discovered as soon as it is their turn to be checked for validity, in future iterations. If all missing dependencies are considered at the same time, the stack $\mathfrak{S}$ is likely to overflow. If only one dependency is considered at a time, the size of $\mathfrak{S}$ will be limited but the number of iterations (hence the running time of the algorithm) will increase. Determining a suitable cutoff threshold for k in order to balance these two contradicting objectives (*i.e.,* memory usage and running time) is an experimental process, and depends on the characteristics of the machine onto which the algorithm is run. Our experience suggests that $\lfloor \frac{10}{\log(j-i)} \rfloor$ provides good results in practice.

b) Limit Missing Dependencies: Furthermore, it is sufficient to consider solely n_2 such that $\mathfrak{C}(n_2) = \bot$ or $\mathfrak{C}(n_2) = i$. Indeed, if $\mathfrak{C}(n_2) > i$ either $\mathfrak{C}(n_2)$ has been increased at Step 3 of one of the previous iterations or n_2 coincides with m and $\mathfrak{C}(n_2) = j$. If $\mathfrak{C}(n_2)$ has been increased at Step 3 of one of the previous iterations, then either $\mathfrak{C}(n_2) = \bot$ or n_2 has been a missing dependency in some configuration $\mathfrak{C}_1$ such that $i \leq \mathfrak{C}_1(n_2) < \mathfrak{C}(n_2)$. Repeating this argument we can observe that there exists a configuration $\mathfrak{C}_2$ such that either $\mathfrak{C}_2(n_2) = \bot$ or n_2 is a missing dependency in it and $\mathfrak{C}_2(n_2) = i$. If n_2 coincides with m, then the only way to obtain a valid configuration is by considering $\mathfrak{C}_1$ with $\mathfrak{C}_1(n_1) > \mathfrak{C}(n_1)$. However, such $\mathfrak{C}_1$ has been added to $\mathfrak{S}$ together with $\mathfrak{C}$ (since if Step 3 added $\mathfrak{C}$ it added all $\mathfrak{C}'$ with $\mathfrak{C}(n_1) < \mathfrak{C}'(n_1) \leq j$).

c) Ordering Versions of a Module: Step 3 adds configurations differing in versions of a module. To improve the search, we consider heuristics determining whether "older" (closer to i) or "newer" (closer to j) versions of a module should be considered first. The first heuristic is to look at the latest version of a module first, *i.e.,* if modules n_1, n_2 and n_3 have to be upgraded we first consider a configuration with the three modules being mapped to version j, than two modules being mapped to j and one to $j-1, \ldots$ We will refer to this heuristic as "3-2-1". The idea is that more versions are more likely to be compatible with m which is being upgraded to the latest version, leading towards a solution—not necessarily the best one—in less steps. Alternatively, we might prefer the older versions of a module. Indeed, Lehman's law of increasing complexity [6] suggests that older versions have less dependencies than the newer ones, *i.e.,* by preferring older versions the search is focussed on finding a minimal solution rather than *a* solution. We call this heuristic "1-2-3".

2) Improve the Search:

a) Search Strategy: A further efficiency gain can be obtained by following the branch-and-bound strategy, *i.e.,* first checking whether $|\mathfrak{u}(\mathfrak{C}, i)| < \mathit{threshold}$ and only then either recording $\mathfrak{C}$ if $\mathfrak{C}$ is valid or adding new configurations to $\mathfrak{S}$ if $\mathfrak{C}$ is invalid. Moreover, to reduce the time required to find the first candidate solution (*i.e.,* the first finite value of the *threshold*) we perform a *preliminary search* by considering only configurations where Step 3 increases versions of all missing dependencies directly to j. This preliminary search is guaranteed to terminate after $|\mathbb{M}|$ iterations.

b) Limit Search: Given the size of the search space, the search needs to be aborted if the optimal solution is not found within a reasonable amount of time. To keep the running time of the search acceptable, a limit has been put in place which aborts the search after considering a specified amount of invalid configurations. We discuss the impact of the choice of the limit value on the optimality of the solutions found vs. the running time of the algorithm in Section II-G1.

c) Time vs. Memory: Finally, to improve the time performance of the algorithm we are ready to trade memory for time by storing functions' output for each input they receive. When the same input is encountered at a later stage, the function can immediately return the stored value. As a result, subsequent calls with the same parameters are served in nearly constant time, potentially achieving major performance boosts. This process is known as memoization [7], [8].

F. Heatmap

The *heatmap* shows a high-level overview of the number of modules that have to be upgraded if the module represented by the row is upgraded from one version to another, as represented by the column; *i.e.,* in terms of Section II-B the module m is represented by the row, version i by the column, and $|\mathfrak{u}(\mathfrak{C}, i)|$ by the colour of the cell, where $\mathfrak{C}$ is a valid configuration inducing a minimal upgrade and satisfying $\mathfrak{C}(m) = j$ for a given version $j > i$. The colours in the heatmap range from white ($|\mathfrak{u}(\mathfrak{C}, i)| = 1$) to dark blue ($|\mathfrak{u}(\mathfrak{C}, i)| = |\mathbb{M}|$). A logarithmic scale is applied such that small changes among the lower values result in a more observable colour change.

G. Case Study

To evaluate the approach proposed we have applied it to software of a large photolithography system, developed by ASML. The software is developed by approximately 1000 developers at ASML and currently consists of more than 40 million lines of code. At the time of the case study, the software contains almost 400 modules and 7000 interfaces. To identify dependencies between the modules we have applied CScout [9], a source code analysis tool for C source files collecting information about the scope and usage of identifiers, as well as a number of proprietary tools providing similar functionality for Python, proprietary data definition files and configuration files. We have successfully extracted information from 327 modules: the remaining modules did not contain

TABLE I: List of versions used for the analysis.

Version	Date
0	October 10, 2011
1	November 15, 2011
2	December 19, 2011
3	January 23, 2012
4	February 20, 2012
5	March 27, 2012
6	May 1, 2012
7	June 4, 2012
8	July 5, 2012

TABLE II: Allowing more invalid configuration barely increases the number of optimal solutions but slows down the search.

Number of invalid configurations	Number of optimal solutions	%	Running time (HH:MM)
1000	1465	56	16:13
2000	1478	56.50	44:07
4000	1513	57.8	62:04
16000	1540	58.87	310:51

source code or contain code in languages not supported by the dependency identification tools.

We have considered nine versions of the system summarized in Table I and 2616 = 327 * 8 scenarios involving upgrade of each one of the 327 modules from version each one of the versions 0–7 to the most recent version, version 8.

1) Evaluation of the Enhancements: We start by discussing the impact of the nhancements discussed in Section II-E on the performance of the algorithm.

In the first series of experiments we study the impact of *the choice of the maximal number of invalid configurations* on the optimality of the solutions found vs. the running time of the algorithm. Recall, that if the algorithm performs limited exhaustive search, *i.e.,* if the algorithm terminates prior to reaching the maximal number of invalid configurations, then the solution found is optimal. Table II shows that higher numbers of invalid configurations result in a significantly higher running time, without significant improvement of the number of optimal solutions. Hence, in the subsequent series of experiments we abort the search after 1000 invalid configurations.

Next we compare the two *heuristics*: "3-2-1" and "1-2-3" by the number of solutions and the number of optimal solutions found during the search. While "3-2-1" always finds a solution, it finds an optimal solution only in 1424 cases (54.4%). The "1-2-3" heuristics finds a solution only in 1723 cases (65.9%), but it is more successful in finding optimal solutions: 1627 (62.2%). The heuristics find configurations inducing equally-sized upgrades in 1424 cases. In 880 cases the "3-2-1" heuristic returns a configuration inducing a smaller upgrade, vs. 302 cases for the "1-2-3" heuristic. However, if "3-2-1" yields a smaller upgrade, the difference is at most 10 modules compared to the "1-2-3" heuristic. When "1-2-3" yields a smaller upgrade, it is often considerably smaller than the one provided by the "3-2-1" heuristic: on average 62 modules less with the maximal difference in upgrade sizes reaching 149 modules.

Further investigation revealed that if the search with heuristic "1-2-3" finds a solution, it does so within 25 steps for 94% of the cases. This means that in a few cases a much cheaper solution can be found in practically no time. Hence, we combine the heuristics as follows: first try to find a solution with the "1-2-3" heuristic and abort the search after 25 steps. If no solution is found, a search will be performed using the "3-2-1" heuristic. The combined heuristics finds an optimal solution for 1524 scenarios: with a negligible amount of additional time, we find 100 more optimal solutions than before.

Memoization has significantly improved the running time of the algorithm. Together with the combined heuristics 2616 upgrade scenarios have been analysed in 112214 seconds of CPU time, *i.e.,* 1 day and 7 hours. The algorithm checked 1358396 possible configurations for validity, while it generated almost 485 million configurations. Due to the limit of 1000 failed configurations per upgrade scenario, most configurations could not be processed. For 1524 upgrade scenarios (58%), the algorithm was able to find an optimum. In most of these cases, the initial solution found by the preliminary search using the "1-2-3" heuristic was also the optimal one. Merely in 33 cases the initial solution was not optimal, and the full search resulted in obtaining the optimal solution. For the remaining 1092 scenarios no optimal solution was found. The improvement with respect to the initial solution is modest: the largest improvement that the algorithm was able to find was a solution with 8 modules less, where the initial solution on average required upgrading 90 modules.

2) Evaluation of the Heatmap Visualization: A partial heatmap of the system is shown in Figure 2. The color of a cell $(\mathit{Module}, \mathit{Version})$ corresponds to the number of modules that have to be upgraded when *Module* is being upgraded from *Version* to the most recent version, version 8. Inspecting this figure we observe that in many cases, the colors in a row become lighter from left to right, *i.e.,* the older the version, the more upgrade dependencies are involved. This can be expected because the time span to the latest version is longer, suggesting that more changes could have occurred. Moreover, most modules show dark cells in columns 0, 1 and 2, and much lighter cells in columns 3–7. This means that most modules are difficult to upgrade to the most recent version 8 if they are of version 2 or older. The heat map does not reveal the cause of this "cliff" from version 2 to version 3: we reconsider this issue when discussing the second visualization proposed, an upgrade dependency graph. Finally, module `AF` is easy to upgrade: its row is completely blank, indicating that there are no upgrade dependencies. This is typical for modules which see little to no development. It could still be the case that this module has changed, but that these changes were internal to the module.

We postpone the discussion of the case study to Section III-C.

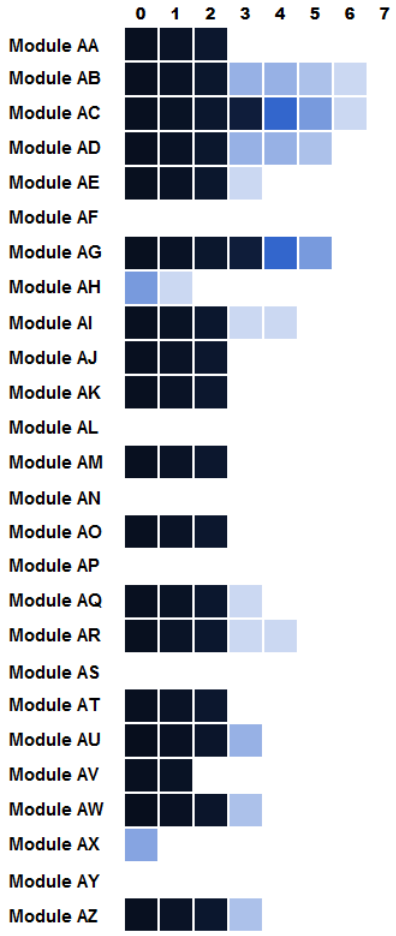


Fig. 2: A partial heatmap showing the upgrade dependencies from versions 0 through 7 to version 8. The more upgrade dependencies, the darker the cell.

III. Why Does Upgrading One Module Require Upgrading Many Additional Modules?

We have observed that frequently upgrading one module requires an upgrade of numerous other modules. Unfortunately, the techniques discussed in Section II cannot provide insights as to why does this happens. Therefore, in this section we present a visualization supporting identification of the reason why one upgrade triggers numerous additional upgrades.

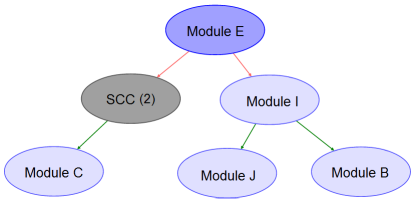
A. Upgrade Dependencies

When a module is being upgraded to a more recent version, modules it uses or modules using it might require an upgrade as well. Consider the following scenarios:

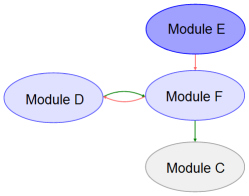
Scenario 1 Let module A be upgraded from version i to version j. If version j of A requires a symbol S from B, that was not required by version i, then upgrading A necessitates the upgrade of B if the current version of B does not already provide S. We say that there is an *upgrade dependency from A to B (caused by adding S).*

Scenario 2 Let module B be upgraded from version i to version j. If version j of B no longer provides a symbol S, provided in version i, and module A requires S, then upgrading B necessitates the upgrade of A. We say that there is an *upgrade dependency from B to A (caused by removing S).*

In the motivation example introduced in Section II-A there are two upgrade dependencies: from A to B caused by adding S and from B to C caused by adding T.



(a) Upgrade dependency graph representing the upgrade of `Module E` (dark blue) and containing an SCC-vertex (dark gray).



(b) Graph shown when the SCC-vertex is clicked upon: module `Module C` is shown in light grayed since it does not belong to the SCC but has an edge coming from one of the SCC vertices.

Fig. 3: Zooming in a SCC-vertex of an upgrade dependency graph.

B. Upgrade Dependency Graph

The *upgrade dependency graph* represents a single upgrade of a module m, based on the configuration $\mathfrak{C}$ which was determined by the algorithm. The upgrade dependency graph is a directed graph with vertices $\{n|\mathfrak{C}(n) > i \wedge \mathfrak{C}(n) \neq \bot\}$. There is an edge from module n_1 to module n_2 if and only if there is an upgrade dependency from n_1 to n_2. Each edge in the upgrade dependency graph is associated with a set of symbols which addition or removal caused the upgrade dependency. Edges are shown in green if all symbols in the set have been added, in red if all symbols in the set have been removed and in black if some symbols in the set have been added and while some other symbols in the set have been removed. Cardinality of the set of symbols is represented by thickness of the edge.

In order to ease comprehension of the graph, the strongly connected components (SCC) are collapsed to a single vertex. A modified version of Tarjan's algorithm [10] is used to locate the strongly connected components. To simplify the inspection of the upgrade dependency graph, we allow the user expanded a SCC-vertex by clicking on it (Figure 3). A new graph is shown with the modules inside the SCC as well as the modules with incoming/outgoing edges from the SCC.

We stress that strongly connected components in an upgrade dependency graph are different from strongly connected components in traditional dependency graphs [11]. Upgrade dependency graphs represent changes in dependencies between the modules. Hence, if the required or provided symbols leading

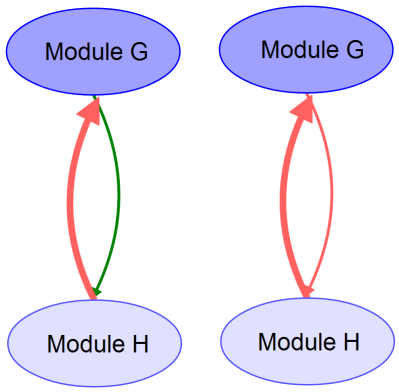


Fig. 4: Cyclic upgrade dependencies not causing new cyclic dependencies between modules.

to a cyclic dependency in a traditional dependency graph have not changed, no relation is shown in the upgrade dependency graph. Moreover, not every cyclic upgrade dependency yields a new cyclic dependency in the software, as shown in Figure 4. In Figure 4a, `Module G` has changed such that it uses new symbols from `Module H` and removes symbols used by that same module. No new cyclic dependency has been introduced because `Module H` does not require additional symbols from `Module G`. In Figure 4b, symbols were removed at both sides. While they depend on each other during an upgrade, they are made more independent because they require less symbols from each other.

C. Discussion of the Case Study

One of the first observations while developing the tooling was the presence of suspicious dependencies, whose existence cannot be immediately explained given the functionality of the modules involved. In many cases these upgrade dependencies involved the test code: while 264 out of 327 modules had more than 150 upgrade dependencies, after the test code has been excluded only 4 out of 327 modules had more than 150 dependencies.

Based on this observation we stress the importance of separating the test code from the production code.

Even after the test code has been excluded, we observe that a module upgrade from the version of October 2011 to the version of July 2012 often includes many additional upgrade dependencies. The heat map in Figure 2 tells us that upgrades are much easier from version 3 (January 2012) and onwards. From there, more than half of the modules are upgradeable with only 10 or less additional modules, where the majority of these modules have no additional upgrade dependencies at all.

By inspecting upgrade dependency graphs we further discover that many edges are red, *i.e.,* many upgrade dependencies are being caused by symbols being removed. If removal of symbols is dismissed, the difficulty of upgrading the modules decreases. As opposed to 204 modules (55%) that require upgrading ten modules or less when upgrading from version 3 to version 8 if symbol removal is taken into account, 91% of the upgrades involve ten modules or less if symbol removal is dismissed.

Therefore, to facilitate the upgrades a policy disallowing symbols removal should be considered, or at least a structured process of symbols deprecation and removal. As an example of such a structured process one might consider not to remove symbols from an interface until they are no longer used in any supported release.

Finally, we have evaluated application of the tool at ASML. ASML developers reported that the tool provided valuable insights in the upgrade structure of the system in an easy and transparent way.

IV. Related Work

Our work should be situated in the area of the update management, and more specifically, component-based update management [3], [12], [13], [14]. This problem is frequently considered *e.g.,* open-source software distributions, development platforms like Eclipse plugins, and Web browser extensions. The key problem that has to be addressed is the problem of dependency solving, *i.e.,* identification of versions of modules that have to be upgraded when another module is being upgraded. State-of-the-art package managers tend to prefer the resulting system to be as up-to-date as possible, *i.e.,* they prefer to update as many modules as possible [5]. This is known as a “progressive” [3] or “trendy” [5] upgrade. Unlike these approaches we prefer to upgrade as few modules as possible, since every module upgraded has to be installed and configured. Our approach is closer to “conservative” [3] or “paranoid” [5] upgrades. We stress, however, that while a “paranoid” upgrade first attempts at minimizing the number of modules removed, and then at minimizing the number of modules changed, we focus solely on the modules changed as removed modules do not require additional installation or configuration effort.

Academic research has explored the possibilities of applying advanced logic-based techniques (*e.g.,* pseudo-boolean optimization [15], [16] and mixed integer linear programming [17]) to dependency solving. The latter approach has became a clear winner in the recent Mancoosi International Solver Competition [5], while *p2cudf* [15] showed better results than the other solvers on the dependency requirements extracted from a real software system (Debian) rather than synthesised by the competition organizers. A detailed comparison of our approach with the results of these solvers is considered as a future work.

To the best of our knowledge none of the approaches targeting dependency solving considering dependency solving addressed the challenge of explaining why upgrading some modules is much more complex than others. This is not surprising as in Eclipse, Debian and similar systems the decomposition is fixed, while in our case the company developers are ready to reconsider the decomposition depending on the feedback provided by the techniques proposed.

Monitoring dependencies through addition and removal of symbols carried out by CScout [9] is reminiscent of the API change and ripple effect detection through the Ecco-Evol metamodel [18]. Furthermore, in context of Java dependencies our search for missing dependencies is similar to automated dependency resolution detection in Maven [19] or in Eclipse plugins [20]. As opposed to [19], [20] we are not interested in finding *a* configuration that satisfies dependency requirements but rather in finding the configuration that satisfies dependency requirements and the number of modules that have to be upgraded is minimal.

We have considered off-line dependency resolution and upgrades. A complementary research domain considers on-line upgrades [21], [22], topic related to run-time evolution [23]. Furthermore, upgrade dependencies are but one of the kinds of dependencies between modules studied in the literature [24], [25], [26].

Finally, directed graph visualization is a well-known problem [27]. We prefer the visualization with a traditional force-directed layout: it was preferred by the company developers to more advanced techniques [28], [29] that seemed to provide few benefits when attempting to discover undesirable upgrade dependencies.

V. Conclusion

We have reported on an industrial approach to complexity assessment of software modules' upgrades. The approach combines a high-level assessment ("is the software structured such that a module can be upgraded with few additional dependencies?") with a lower-level visual feedback to developers ("if an upgrade causes many additional dependencies, what is the cause of this?"). Based on the feedback the developers can consider restructuring the system to simplify future upgrades.

The approach has been applied to a software of a large photolithography system, developed by ASML. The software is developed by approximately 1000 developers and currently consists of more than 40 million lines of code. We have considered nine versions of the system and 2616 = 327 * 8 scenarios involving upgrade of each one of the 327 modules from each one of the versions 0–7 to the most recent version, version 8. In total, the analysis took 112214 seconds, *i.e.,* 1 day and 7 hours. Optimal solution was found in 58% of the scenarios.

ASML developers reported that the tool developed provided valuable insights in the upgrade structure of the system in an easy and transparent way.

Future work A number of directions pertaining to designing the techniques and evaluating it can be considered as a future work.

First of all, it is relatively straight-forward to extend the algorithm in Section II-C such that the upgrade scenario is triggered by a simultaneous update of *multiple* modules. While the result of the algorithm cannot be used to analyse the upgrade effort for each one of the modules separately, this algorithm extension allows one to consider more realistic upgrade scenarios. Furthermore, we did not consider semantic changes in the symbols, *i.e.,* symbol S provided by module B at $t = 2$ is considered to be identical to symbol S provided by module B at $t = 3$. Distinguishing between different versions of S can be carried out by identifying elements in B and modules B depends upon that can affect S, and investigating evolution of these elements by mining the version control system [30]. Similarly, to support comprehension of the upgrade dependencies between modules, graph-based visualization discussed in Section III can be extended by hoover-on annotations derived from commit comments in the version control system. One can also augment the visualization approach with metrics, derived from the upgrade dependency graph, such as different centrality measures. This metrics should allow the developers to pinpoint modules "responsible" for high upgrade effort without manual inspection of large graphs (cf. the "highlight problems" requirement [31] and a drill-down approach for measuring maintainability [32]). Finally, applying econometric techniques [31], [33], [34] to these metrics will allow us to assess concentration of upgrade dependencies over the modules, and prefer configurations with the highest concentration of the upgrade dependencies as only few modules need to be considered to eliminate these dependencies.

To evaluate the technique we intend to conduct a formal user study. This would allow us to experiment with different usage scenarios as well as with different visualization aspects (such as the color palette).

Acknowledgment

The authors would like to thank Cor Hurkens, Tom Verhoeff, Michel Westenberg, Anton Wijs and Joost Zonneveld for their ideas, insights and information, which helped us to improve various aspects of the approach.

References

[1] L. Wingerd and C. Seiwald, "High-level best practices in software configuration management," in *System Configuration Management*, ser. Lecture Notes in Computer Science. Springer, 1998, vol. 1439, pp. 57–66.

[2] E. Grossman, "An update on software updates," *ACM Queue (March 2005)*, 2005.

[3] T. van der Storm, "Continuous release and upgrade of component-based software," in *SCM*. ACM, 2005, pp. 43–57.

[4] B. Schoenmakers, "Assessment of software module upgrade complexity," Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, Aug. 2012.

[5] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "Dependency solving: A separate concern in component evolution management," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2228–2240, 2012, automated Software Evolution.

[6] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, September 1980.

[7] R. S. Bird, "Tabulation Techniques for Recursive Programs," *ACM Comput. Surv.*, vol. 12, no. 4, pp. 403–417, Dec. 1980.

[8] U. A. Acar, G. E. Blelloch, and R. Harper, "Selective memoization," *SIGPLAN Notes*, vol. 38, no. 1, pp. 14–25, Jan. 2003.

[9] D. D. Spinellis, "CScout: A refactoring browser for C," *Science of Computer Programming*, vol. 75, no. 4, pp. 216–231, 2010.

[10] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal of Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[11] D. L. Parnas, "Designing software for ease of extension and contraction," in *Proceedings of the 3rd international conference on Software engineering*, ser. ICSE '78. Piscataway, NJ, USA: IEEE Press, 1978, pp. 264–277.

[12] D. J. Brown, "An update on software updates," *Queue*, vol. 3, no. 2, pp. 10–11, Mar. 2005.

[13] S. Jansen, G. Ballintijn, and S. Brinkkemper, "A process model and typology for software product updaters," in *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, 2005, pp. 265–274.

[14] M. C. Annosi, M. Di Penta, and G. Tortora, "Managing and assessing the risk of component upgrades," in *Product Line Approaches in Software Engineering (PLEASE), 2012 3rd International Workshop on*, 2012, pp. 9–12.

[15] J. Argelich, D. L. Berre, I. Lynce, J. P. Marques Silva, and P. Rapicault, "Solving Linux upgradeability problems using boolean optimization," in *Proceedings First International Workshop on Logics for Component Configuration*, ser. EPTCS, I. Lynce and R. Treinen, Eds., vol. 29, 2010, pp. 11–22.

[16] P. Trezentos, I. Lynce, and A. L. Oliveira, "Apt-pbo: solving the software dependency problem using pseudo-boolean optimization," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 427–436.

[17] C. Michel and M. Rueher, "Handling software upgradeability problems with MILP solvers," in *Proceedings First International Workshop on Logics for Component Configuration*, ser. EPTCS, I. Lynce and R. Treinen, Eds., 2010, pp. 1–10.

[18] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation?: the case of a smalltalk ecosystem," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 56:1–56:11.

[19] J. Ossher, S. Bajracharya, and C. Lopes, "Automated dependency resolution for open source software," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, 2010, pp. 130–140.

[20] J. Businge, A. Serebrenik, and M. G. J. van den Brand, "Survival of eclipse third-party plug-ins," in *ICSM*. IEEE Computer Society, 2012, pp. 368–377.

[21] T. Dumitraş and P. Narasimhan, "Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware '09. New York, NY, USA: Springer-Verlag New York, Inc., 2009, pp. 18:1–18:20.

[22] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Safe and automatic live update for operating systems," *SIGPLAN Not.*, vol. 48, no. 4, pp. 279–292, Mar. 2013.

[23] N. M. Villegas, G. Tamura, H. A. Mller, L. Duchien, and R. Casallas, "Dynamico: A reference model for governing control objectives and context relevance in self-adaptive software systems," in *Software Engineering for Self-Adaptive Systems II*, ser. Lecture Notes in Computer Science, R. Lemos, H. Giese, H. Mller, and M. Shaw, Eds. Springer Berlin Heidelberg, 2013, vol. 7475, pp. 265–293.

[24] H. H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *WCRE*, G. Antoniol, M. Pinzger, and E. J. Chikofsky, Eds. IEEE Computer Society, 2010, pp. 119–128.

[25] S. A. Roubtsov, A. Serebrenik, and M. G. J. van den Brand, "Detecting modularity "smells" in dependencies injected with java annotations," in *CSMR*, R. Capilla, R. Ferenc, and J. C. Dueñas, Eds. IEEE, 2010, pp. 244–247.

[26] A. Sutii, S. Roubtsov, and A. Serebrenik, "Detecting dependencies in Enterprise JavaBeans with SQuAVisiT," in *WCRE*, R. Oliveto and R. Robbes, Eds. IEEE Computer Society, 2013.

[27] I. Herman, G. Melancon, and M. Marshall, "Graph visualization and navigation in information visualization: A survey," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 6, no. 1, pp. 24–43, jan-mar 2000.

[28] D. H. R. Holten, "Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 5, pp. 741–748, 2006.

[29] M. Krzywinski, I. Birol, S. J. M. Jones, and M. A. Marra, "Hive plots–rational approach to visualizing networks," *Briefings in Bioinformatics*, vol. 13, no. 5, pp. 627–644, 2012.

[30] W. Poncin, A. Serebrenik, and M. G. J. van den Brand, "Process mining software repositories," in *CSMR*, T. Mens, Y. Kanellopoulos, and A. Winter, Eds. IEEE Computer Society, 2011, pp. 5–14.

[31] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse, "Software quality metrics aggregation in industry," *Journal of Software: Evolution and Process*, pp. n/a–n/a, 2012. [Online]. Available: http://dx.doi.org/10.1002/smr.1558

[32] P. Hegedűs, T. Bakota, G. Ladányi, C. Faragó, and R. Ferenc, "A drill-down approach for measuring maintainability at source code element level," in *Proceedings of the Seventh International Workshop on Software Quality and Maintainability*, ser. SQM '13, 2013.

[33] A. Serebrenik and M. G. J. van den Brand, "Theil index for aggregation of software metrics values," in *ICSM*. IEEE Computer Society, 2010, pp. 1–9.

[34] B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, "You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics," in *ICSM*. IEEE, 2011, pp. 313–322.