Liquid Version Climber: an automated tool for upgrading complex software systems

Christophe Pradal
UMR AGAP, CIRAD and Inria, Montpellier, France
christophe.pradal@cirad.fr

Sarah Cohen-Boulakia
Inria, Montpellier, LRI CNRS 8623, U.Paris Sud, France
cohen@lri.fr

Patrick Valduriez
Inria and LIRMM, Montpellier, France
Patrick.Valduriez@inria.fr

Dennis Shasha
Courant Institute, New York University, USA
Inria and LIRMM, Montpellier, France
shasha@cs.nyu.edu

Abstract

Suppose you are given a software system that is composed of a set of packages each at a particular version. You want to update some packages to their most recent versions possible, but you want your software to run after the upgrades, thus perhaps entailing changes to the versions of other packages. One approach is trial and error, but that quickly ends in frustration. We advocate a provenance-style approach in which tools like *ptrace*, *reprozip*, *pip*, and virtual machines combine to enable us to explore version combinations of different packages even on a variety of platforms. Our approach also contributes to reproducibility by allowing the replay of an experiment from a given configuration of software and data package-versions to work on a new configuration, thus helping to answer the question of whether a particular result was an artifact of some logic bug of some particular package-version. Because the space of versions to explore grows exponentially with the number of packages, we have developed a memoizing algorithm that avoids exponential search while guaranteeing an optimum version combination. We have ideas for more efficient algorithms under certain assumptions.

1. Introduction

Software systems are increasingly complex and compositional. This is particularly evident in the context of free and open source software which may involve very large groups of independently developed packages all evolving at different rates. End-user scientists use software systems but they have neither the interest nor the technical skill to engage in any of the continuous development practices[5, 8] that are currently considered best practices. More likely, they find themselves needing to update some software system (often embedded in a workflow) whose original developer has left. When they fail, they suffer from what has been called "workflow analysis decay" [12] where scientific experiments relying on intertwined packages cannot be executed even only a few months after they have been developed.

Ideally, such researchers want to create a running systems with certain chosen packages/data updated to recent versions. However, updating one package may entail changes in the versions of other packages. One approach to explore this package-version space can be trial and error among version combinations, but that would be unsustainably time-consuming given the number of combinations to be tested. The goal of the *Liquid Version Climber* system is to provide an intelligent automation of this version space exploration.

The remainder of this paper is organized as follows. Section 2 provides a motivating example. Section 3 describes the Liquid Climber algorithm. Section 4 describes the implementation substrate for protected execution of configurations. Section 5 shows a case study. Section 6 places Liquid Climber within related work. Section 7 concludes the paper.

2. Motivating Example

Consider a scientist Lucy who has designed a machine learning experiment on image analysis in a given environment represented by configuration C that uses Scikit-Learn version 0.8, Scikit-Image version 0.4, SciPy version 0.10.1, NumPy version 1.4.1, and Python version 2.6. Scikit-Learn and Scikit-Image depend on SciPy. Both Scikit-Learn, Scikit-Image, and scipy depend on NumPy. All packages depend on Python. Configuration C works in that the experiment of Lucy completes to execution with such versions of the Scikit-Learn, NumPy and Python packages. Now, imagine that a new version of Scikit-Learn appears (0.14) with new features that Lucy would like to use. Lucy would thus like to upgrade Scikit-Learn version 0.8 to 0.14. However, if Scikit-Learn is modified in isolation (*i.e.*, the other packages remain at the same versions as in configuration C), the new configuration will not work because Scikit-Learn 0.10 requires a newer version of NumPy (1.6.1 or greater) which in turn depnds on Python 2.7.

Even if release notes (when they are available) provides some dependency information, end-user scientist Lucy does not want to become a goddess of version dependencies. Instead, she'd like some easy-to-use software to figure out which versions to advance to when advancing and then to produce a configuration she can use. That's what the *Liquid Version Climber* system aspires to be.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Building such a system entails achieving several capabilities:

1. To identify the packages and versions of the original working configuration C.
2. To combine and link different versions of the packages from C to form a new configuration C' (a new set of package-versions).
3. To execute such a new configuration C' and to detect where it failed if it fails.
4. To use failure information to find the best possible configuration efficiently.

3. Algorithm

In this section, we provide the main definitions underlying our approach and then present the Liquid Climber algorithm.

3.1 Definitions

Queries Users express queries describing their desires (which packages to upgrade) and constraints (hereafter called query constraints). Queries are of the form: *maximize versions of the following packages P1, P2, P3, ... Pk in that order (so it is more important for the user to maximize P1 than P2 ...) with the constraint that Pi should be either version 1 through 3 and Pj should be version 2 through 7.*

Configurations. A configuration is a set of package-version pairs. The initial configuration is given and works.

Definition: Two configurations are equal if they have the same versions for every package.

Definition: Suppose the packages to maximize are P1, ... Pk in descending order of priority. We say that configuration C' = P1.v1', P2.v2', ..., Pk.vk', ..., Pn.vn' is lexicographically greater ($>$) than C = P1.v1, P2.v2, ..., Pk.vk, ..., Pn.vn

if either P1.v1' $>$ P1.v1 or (P1.v1' = P1.v1 and P2.v2' $>$ P2.v2) or ... or (P1.v1' = P1.v1 and P2.v2' = P2.v2 and ... and Pk-1.vk-1' = Pk-1.vk-1 and Pk.vk' $>$ Pk.vk)

Notice that lexicographic comparison is indifferent to packages that need not be maximized.

Definition: A configuration "works" if it executes to completion.

3.2 Assumption

In this paper, we limit ourselves to a single assumption to reduce the need to do an exhaustive (and therefore exponential) search in the space of package-versions. This gives us a polynomial but somewhat expensive algorithm.

Pairwise/Global compatibility (pairwise compatible implies global compatibility): An execution using configuration P1.v1, ... , Pn.vn works if for all i, j ($1 \leq$ i,j $\leq$ n) such that Pi calls Pj, no call from Pi.vi to Pj.vj crashes.

3.3 LiquidClimber – basic

Goal of Algorithm. The LiquidClimber algorithm ends with a configuration that is lexicographically maximum (e.g. maximize packages P1, P2 in that order) and that satisfies the query constraints on versions (e.g. only use versions 4, 5, and 6 of package P7). The query constraints are used to form sourcemap but play no other role in the algorithm. The order of packages to maximize are encoded in todolist.

The variable *memo* keeps tracks of calls that have failed, e.g. Pi.vi calling Pj.vj has failed. If a configuration contains a pair of package-versions that have failed, then that configuration will fail so will not need to be executed.

```
current    := package version pairs in the initial
              configuration
todolist   := packages requiring maximization in
              descending order of priority
constraints := some query constraints on versions of
               various packages
sourcemap := for each package the versions that
             exist and satisfy query constraints
memo := calls for some Pi.vi to some Pj.vj that fail.
             Initially this is empty

LiquidClimber()
  for each package p in todolist in descending order of priority
      update current to the highest version of p that works
      # use git binary on sourcemap
       if current has less than highest version of p
       in sourcemap then
        versionstodo := find all greater versions
                        of p or major releases in
                        ascending order of version
                        number within sourcemap
       for each v in versionstodo in order
          temp := current with p set to v
          ret := TryToMakeWork(p, temp)
          if ret is not null then # we've had success
             current:= ret

# adjust temp to make it work
# p is the package we're trying to push
TryToMakeWork(p,temp)
  execute temp unless we know from memo that temp will fail
  if the execution of temp fails
    Find the first call, say from Pi.vi' to Pj.vj',
    that fails
    Record in memo that Pi.vi' calling Pj.vj' fails
    keepfixed := {x | x == p
       or a package earlier than p in the todolist}
    if Pi in keepfixed then
      possible(Pi) := {vi'}
    else
      possible(Pi) := {all versions of Pi in sourcemap}
    end if
    if Pj in keepfixed then
      possible(Pj) := {vj'}
    else
      possible(Pj) := {all versions of Pj in sourcemap}
    end if
    for each untried configuration c that can
      be constructed from the cross-product of possible(Pi)
      and possible(Pj) such that c does not
      include some pair (Pk.vk, Pm.vm) in memo
          ret := TryToMakeWork(p,c)
          if (ret is not null) then return ret
    end for
    return null
  else return temp # this configuration works
```

Theorem: The algorithm finds the lexicographically maximum configuration that works.

Proof sketch: The query constraints are taken into account by the construction of sourcemap. Todolist is ordered from most important package to least. Suppose that there is a another configuration C=P1.v1", P2.v2", ..., Pn.vn" that is lexicographically greater and that also works. We prove that P1.v1", P2.v2", ..., Pn.vn" must have been tried by our algorithm.

Let Px be the first package in Q for which Px.Vx $>$ Px.Vx. By construction of versionstodo, Px.vx was tried by the algorithm. If TryToMakeWork did not succeed to make it work, there was some call from Pi.vi to Pj.vj that failed. But TryToMakeWork will try all combinations of versions between Pi and Pj as long as those combinations aren't in memo and don't affect previously optimized

packages or the package that is currently pushed (for which other versions will be tried in the for each loop LiquidClimber). The recursive call continues this exploration in a depth first manner. Done.

Theorem: If there are a maximum of M versions per package and n packages, then the number of tests $\leq n \times (n-1) \times M \times M$

Proof: The number of possible calling pairs $\leq n \times (n-1)$. Equality holds only in the case that every package calls every other package. For each package Pi that calls package Pj, in the worst case, we might have to test each version of Pi calling each version of Pj. Each time such a call of the form Pi.vi calling Pj.vj causes an error, we record that fact in memo. Once recorded, no configuration that includes Pi.vi and Pj.vj will be executed. So, for every such pair Pi and Pj, there need be at most $M \times M$ executions. Done.

We have started here with the most minimal assumption. Stronger assumptions could be considered (e.g., if Pi.vi calls Pj.vj and fails, then any version greater than vi will fail on any version less than vj). Considering such stronger assumptions is part of our ongoing work.

3.4 LiquidClimber – optimized

Sarah: In this pass, we need to make our proof sketches solid proofs in this section and elsewhere

The previous section assumed that there was no particular semantics to version order. That of course is false in practice. In real software, if Pi.vi calls Pj.vj and fails then Pj.vj is missing a function that Pi.vi needs. Functions are ofren added to packages (Pj in this case) as versions increase, but functions are normally not removed. This model is encoded in the CUDF dependencies of the form Pi version 8 depends on Pj version 10 or greater.[1]

Further we assume that newer versions of a caller will depend on newer versions of a callee. To continue our example from the previous paragraph, Pi version 9 depends on Pj version 13 or greater is more reasonable than Pi version 9 depends on Pj version 7 or greater.

We incorporate all this in the following assumption.

Dependency monotonicity: Suppose that Pi.vi succeeds on a call to Pj.vj. If Pi.vi' depends on vj' or greater of Pj and vi $<$ vi', then Pi.vi will succeed on a call to Pj.y for all y $\geq$ vj'.

Lemma 1: If Pi.vi fails on a call to Pj.vj, then for all x $\geq$ vi, y $\leq$ vj, Pi.x will fail on a call to Pj.y.

Proof: Consider x $>$ vi, for Pi.x to succeed on a call to Pj.y, there must be an x $\geq$ x' $>$ vi and a y' $\leq$ y such that Pi.x' depends on y' or greater. But in that case by dependency monotonicity and the fact that x $\geq$ x' $>$ vi and y' $\leq$ y $\leq$ vj, Pi.vi will succeed on Pj.vj. Contradiction. Done.

Our algorithm doesn't know whether or not these dependencies hold. However, if the (hidden) dependencies satisfy dependency monotonicity, the following will achieve a lexicographically maximum configuration.

Algorithm: StrongMono LiquidClimber

Sarah: the simplest way to state the algorithm is as a variant of the previous one with a few changes marked below as CHANGED

```
current    := package version pairs in the initial
              configuration
todolist   := packages requiring maximization in
              descending order of priority
constraints := some query constraints on versions of
               various packages
sourcemap := for each package the versions that
             exist and satisfy query constraints
memo := calls for some Pi.vi to some Pj.vj that fail.
            Initially this is empty

LiquidClimber()
  for each package p in todolist in descending order of priority
      update current to the highest version of p that works
      # use git binary on sourcemap
       if current has less than highest version of p
       in sourcemap then
        versionstodo := find all greater versions
                        of p or major releases in
                        ascending order of version
                        number within sourcemap
       for each v in versionstodo in order
          temp := current with p set to v
          ret := TryToMakeWork(p, temp)
          if ret is not null then # we've had success
             current:= ret

# adjust temp to make it work
# p is the package we're trying to push
# Knowing that a version will fail
# makes use of strong monotonicity:
# if Pk.vk calling Pm.vm fails then for all x >= vk and y <= vm,
# Pk.x will fail on Pm.y.
TryToMakeWork(p,temp)
  execute temp unless we know from memo and strong
    monotonicity that temp will fail # CHANGED
  if the execution of temp fails
    Find the first call, say from Pi.vi' to Pj.vj',
    that fails
    Record in memo that Pi.vi' calling Pj.vj' fails
    keepfixed := {x | x == p
       or a package earlier than p in the todolist}
    if Pi in keepfixed then
      possible(Pi) := {vi'}
    else
      possible(Pi) := {all versions of Pi in sourcemap
        that are less than or equal to vi} # CHANGED
    end if
    if Pj in keepfixed then
      possible(Pj) := {vj'}
    else
      possible(Pj) := {all versions of Pj in sourcemap
       that are greater than or equal to vj} # CHANGED
    end if
    for each untried configuration c that can
      be constructed from the cross-product of possible(Pi)
      and possible(Pj) such that c does not
      include some pair (Pk.vk, Pm.vm) that strong
      monotonoicity would eliminate based on memo # CHANGED
          ret := TryToMakeWork(p,c)
          if (ret is not null) then return ret
    end for
    return null
  else return temp # this configuration works
```

Complexity: Every execution advances the version of at least one package (either Pi or Pj) so complexity is linear with the number of versions.

Lemma 2: Under the two assumptions of local/global and strong monotonicity, any configuration that is ignored will fail.

Proof Sketch: The main thing to justify is that the set of versions that are still tried can be ignored. Let us say that Pi.vi has called Pj.vj and fails. Then for vi' $\geq$ vi, Pi.vi' will also fail on Pj.y for y $\leq$ vj, by lemma 1. Done.

Theorem: Under the two assumptions of local/global and strong monotonicity, the algorithm finds the lexicographically maximum configuration that works.

[1] NB. Consider a dependency set D = that Pi.1 calls and is successful on Pj.1 and Pi.3 calls and is successful on Pj.3, but there are no other compatibilities. D would not satisfy the CUDF model, because we wouldn't have Pi.1 being able to call Pj.3 successfully.

Proof Sketch: LiquidClimber will go in priority order through the packages to be maximized. Any configuration that is skipped will fail by lemma 2.

Overall Algorithm

Start with the strong monotonicity algorithm and record all successful executions as well as all unsuccessful pairs (not the inferred ones, just the ones directly tried).

Then if in the order of the todolist, we have achieved the maximum out of the first L $\leq$ K, take those as fixed and call the earlier version of LiquidClimber.

So, if strong monotonicity holds, we have a linear algorithm, but even if it doesn't, we have an algorithm that works.

4. Implementation

Our implementation uses the general strategy of Figure 1.

- Execute the original configuration C and then wrap all relevant parts of the execution into a virtual machine. We call such a virtual machine *frozen* because all its versions are fixed in binary. In addition, infer the package-versions used within the execution. *Reprozip* allows us to do this by simply running the execution.
- Abstract from the package-versions in the frozen virtual machine the ability to create virtual machines having arbitrary versions of those same packages. We call the result a *liquid virtual machine*, because it has no fixed binding to particular versions of packages. From the liquid virtual machine, multiple frozen ones can be built each corresponding to a different configuration by pulling versions from a repository like *git*.
- Take a query consisting of packages whose versions should be maximized (in descending order of priority) and query constraints on the versions of some of these or other packages. Use LiquidClimber to find the lexicographically maximum such configuration and freeze it. Specifically, LiquidClimber will call the operational subsystem with a configuration *config* to try. The operational subsystem executes the following protocol.

```
Starting with the liquid virtual machine,
create an isolated environment (e.g. VirtualEnv)
for each package, version in config:
   Checkout the package at a given version
   Install this package with its dependencies in
    the isolated environment (pip install)
Run the script using reprozip
if the script succeeds report success
else report failure and
     the cause of failure (which call)
```

5. Experiments

Lucy wants to build a classifier to predict the value of a handwritten digit from an image. To train the classifier, the MNIST database of handwritten digits is used and the HOG features are calculated for each feature of the database. Finally, a multi-class linear SVM classifier is trained with the HOG features.

The Scikit-Image package is used to calculate the HOG features while the Scikit-Learn package is used perform prediction and training of a linear support vector machine. Execute the script under Reprozip[4]. Reprozip infers an initial configuration containing Scikit-Image version 0.4, Scikit-Learn version 0.8, NumPy, SciPy, and Python.

Lucy wants to maximize NumPy, SciPy, Scikit-Image and Scikit-Learn in that order. From github, the git repositories are cloned. A liquidVM automatically extracts the list of commits and tags of all the packages.

In our case, the number of commits for Scikit-Learn between the version 0.8.0 and as of May 1, 2015 is 14812 and for Scikit-image 5952. We choose to apply the LiquidClimber routine on the tags rather than on the commits. There are only 33 tags of Scikit-Learn since version 0.8 and 35 tags of Scikit-Image since version 0.4. These packages are tested with 23 versions of NumPy and 17 versions of SciPy.

For each configuration that LiquidClimber tests, a virtual environment is created. Each package of the configuration is checked out and installed in the virtual environment. Then the script or scientific workflow is run using Reprozip. If the execution fails, Reprozip provides the two last packages that have been in conflict. This information is returned to Liquid Climber to push the version of the concerned packages. After 288 tests, the system reaches NumPy : version 1.9.2 SciPy : version 0.15.1 Scikit-Image : version 0.11.3 Scikit-Learn : version 0.16.0. Without any manual effort, Lucy now has a working configuration consisting of recent versions of the packages she is interested in.

6. Related Work

Given a configuration of package-version pairs of a software system, the general approach of Liquid Version Climbing is to upgrade certain packages Pupgrade as much as possible given query constraints C. The packages constrained in C may overlap in any arbitrary way with the packages in Pupgrade.

The first step in our approach is to capture the initial configuration. Most tools that do that use some variant of *ptrace*. The particular tool we use is ReproZip [4], though CARE [7] or CDE-SP [9] could have been used with some more enhancements. These tools find the configuration that was used in the initial execution. Our software must make explicit the package-versions that are involved and then "liquify" the packages.

The next issue is to discover how much upgrading we can do of the packages in Pupgrade. Many excellent package managers are available such as Debian's Apt (with front end tool Aptitude), and the P2 tool in Eclipse [2] Package managers fetch components with particular versions from remote repositories and perform the deployment, aborting if there is a problem. Liquid Version Climber is a tool that searches for configurations satisfying query constraints C to optimize Pugrade. Thus, it could use these package managers as components. The essential novelty of Liquid Version Climber is that it explores the space of possible configurations without requiring any a priori information about package-version compatibilities.

The project CUDF (Common Upgradeability Description Format) [10] assumes an input of compatibility dependencies, e.g. P1 version 8 depends on P2 version 18 or above and P3 version 8 or above etc. Given those, the system supports the user's being able to state a request of the form Px version greater than 4 and Py version greater than 5. CUDF thus has similar objectives and a similar query language to ours. (CUDF is based on the very nice theoretical work on dependency solving in [1] which we commend to the reader.)

However, there are two fundamental differences between our approach and CUDF. First, CUDF depends on having the CUDF information input by someone or something. By contrast, Liquid Version Climber discovers compatibilities through execution. Thus, CUDF assumes the pre-existence of information that Liquid Version Climber learns. Second, CUDF might be too conservative in that Pi.vi (package Pi version vi) might be incompatible with Pj.vj, but only for an obscure reason that is not relevant to a particular application. Liquid Version Climber finds compatibilities empirically with respect to the application of interest.

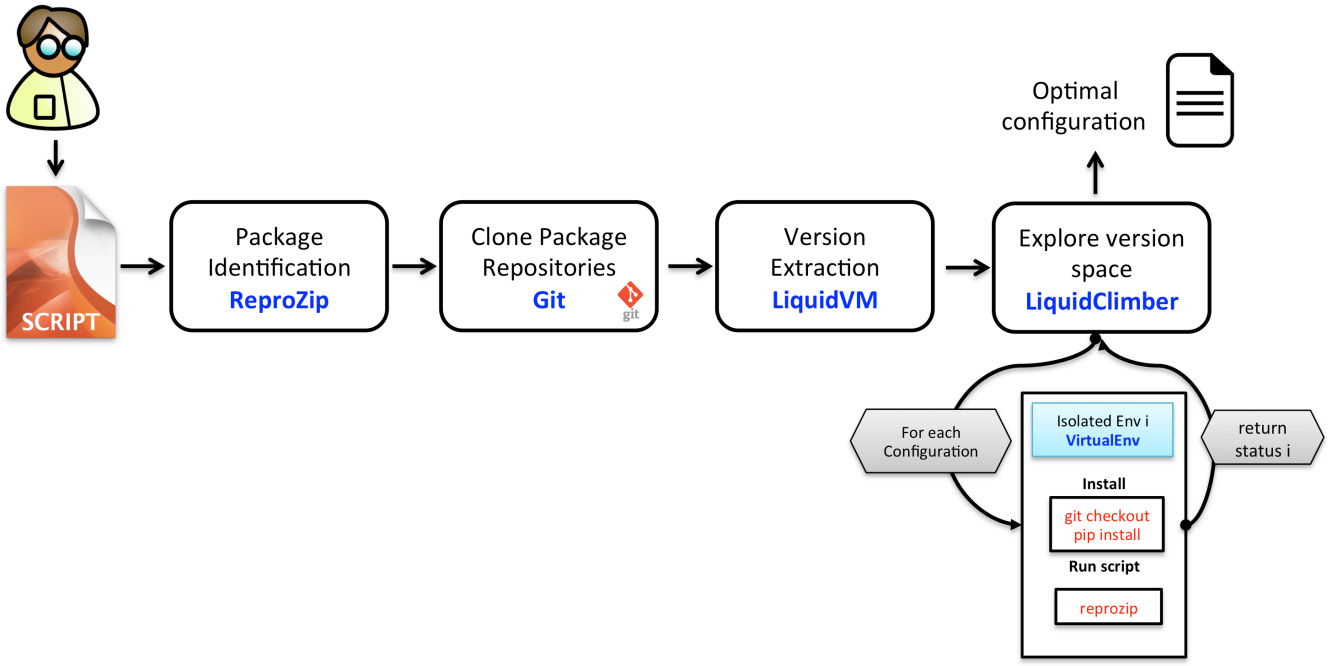

Figure 1. The steps of the operational subsystem: capture the execution of the initial configuration, liquify, fetch versions from git/svn etc., then deploy as directed by LiquidClimber.

Operationally, Liquid Version Climber follows a generate-and-test paradigm. There is a long history of such work in large software development projects using tools like git bisect to find logical bugs or performance bugs in single packages[3, 6]. Our essential novelty with respect to that work is that Liquid Version Climber works on multiple packages that may have been developed independently.

Last but not least, because our audience consists of natural scientists who have neither the desire nor the training to become software developers, our tool is meant to support an interface in which the user simply states goals and the system is guaranteed to achieve the best possible outcome. In this way, it is completely compatible with good scientific software practices, urged in for example in the paper [11]: The software carpentry initiative recommends that scientists who develop their own software follow some best practices, such as writing readable well-formatted programs, using version control, writing test cases, documenting and using components as much as possible. Liquid Version Climber requires only the use of components and some version control system. So Liquid Version Climber would be a good tool for a would-be software carpenter.

7. Conclusion

Liquid Version Climber is a collection of techniques that incorporate provenance-inspired tools to help software system users (such as natural scientists) take an existing working configuration of package-versions and advance to more up-to-date working configurations. The goal is to do this rationally and efficiently while requiring minimal user effort (give a query and let it rip).

Our basic approach is to gather the package-versions of the initial configuration, abstract this so that arbitrary versions can be fetched, establish an order of priority among packages, test configurations and learn from success or failure which other configurations to test. At the end we find a working configuration that is lexicographically, (based on package priority) maximum.

Future work involves making this system more efficient and easier to use.

Acknowledgments

This work was performed at the Institut de Biologie Computationnelle (www.ibc-montpellier.fr) and has been supported by the RecProv Project. Support for Shasha through an INRIA International Chair and the U.S. National Science Foundation under grants MCB-1412232, IOS-1339362, MCB-1355462, MCB-1158273, IOS-0922738, and MCB-0929339. This support is greatly appreciated.

References

[1] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, 2012.

[2] D. Berre and P. Rapicault. Dependency management for the eclipse ecosystem. *IWOCE*, 2009.

[3] T. Chen, L. I. Ananiev, and A. V. Tikhonov. Keeping kernel performance from regressions. In *Linux Symposium*, volume 1, pages 93–102, 2007.

[4] F. S. Chirigati, D. Shasha, and J. Freire. Reprozip: Using provenance to support computational reproducibility. In *TaPP*, 2013.

[5] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[6] C. Heger, J. Happe, and R. Farahbod. Automated root cause isolation of performance regressions during software development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 27–38. ACM, 2013.

[7] Y. Janin, C. Vincent, and R. Duraffort. Care, the comprehensive archiver for reproducible execution. In *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and*

New Publication Models in Computer Engineering, page 1. ACM, 2014.

[8] C. Larman. *Agile and iterative development: a manager's guide*. Addison-Wesley Professional, 2004.

[9] Q. Pham, T. Malik, and I. Foster. Auditing and maintaining provenance in software packages. In *Provenance and Annotation of Data and Processes*, pages 97–109. Springer, 2014.

[10] R. Treinen and S. Zacchiroli. Common upgradeability description format (cudf) 2.0. *The Mancoosi project (FP7)*, 3, 2009.

[11] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. Chue Hong, M. Davis, R. T. Guy, S. H. D. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, B. Waugh, E. P. White, and P. Wilson. Best practices for scientific computing. *PLoS Biol*, 12(1):e1001745, 01 2014.

[12] J. Zhao, J. M. Gomez-Perez, K. Belhajjame, G. Klyne, E. Garcia-Cuesta, A. Garrido, K. Hettne, M. Roos, D. De Roure, and C. Goble. Why workflows breakunderstanding and combating decay in taverna workflows. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–9. IEEE, 2012.