Change in Software Ecosystems

Social Challenges of Automating Upgrades

Gabriel Matute [*1], **Alvin Cheung** [†1] **and Sarah E. Chasins** [‡1]

[1] *University of California, Berkeley, CA*

Abstract

As software continually changes, communities must propagate and adopt potentially disruptive updates. This is difficult because software ecosystems are complex social systems that struggle with scale, mismatched priorities, heterogeneous levels and areas of expertise, and limits on communication and collaboration. In this paper, we aim to explore the social challenges of rolling out software changes. We first characterize some of these challenges. Next we describe a community structure that has allowed some organizations to develop efficient and scalable tooling for adapting code to handle software changes. We end by discussing current and proposed solutions for propagating changes—and the problems they still face.

Keywords: Software ecosystems. Software evolution. Dependency management. Social computing.

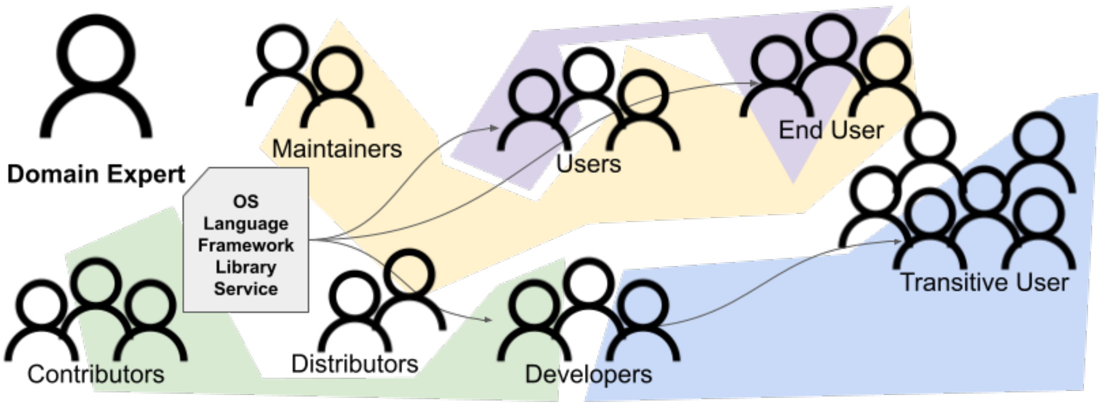

Figure 1. Representation of the social challenges in a hypothetical *software ecosystem* from the perspective of a *domain expert* (top left) when introducing a change in *software* (rectangle) that might affect its *dependants* (arrows). There is large number of *community members* (black figures) present, grouped by their *roles* (labelled) and encompassed within different *organizations* (colored polygons) highlighting the potentially different priorities and expertise that forms the ecosystem.

1 Software Ecosystems

Researchers have observed that software systems continually change and evolve [1]. Developers often add or expand features. Sometimes these changes require refactoring the existing structure to increase maintainability. New platforms or even government policies might require changes to adapt or remain compliant. Finally, discovered issues might require corrections, like security fixes.

However, software does not exist in a vacuum. It is usually designed and implemented by many different developers and domain experts. It is distributed and maintained by a potentially different set of developers, IT operators, and system admins. It is then used by technical and non-technical users to achieve a variety of tasks with diverse requirements. There are many human factors that affect how changes propagate through a community.

PLATEAU
12th Annual Workshop at the Intersection of PL and HCI

[*] Email: gmatute@berkeley.edu
[†] Email: akcheung@berkeley.edu
[‡] Email: schasins@berkeley.edu

With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.

—Hyrum's Law

2 Example Changes

Any discernible (and sometimes even apparently trivial) software change will affect, for better or worse, its surrounding ecosystem. Many programming languages and tools give developers ways to specify interfaces and manage dependencies, which can help users handle change-induced conflicts. Nevertheless, as software is intrinsically part of a human ecosystem, many social factors affect the ability of any automated tool to propagate changes.

To explore the social challenges of propagating an update, we present two hypothetical examples of interface changes. These were manually crafted for simplicity, but were inspired from real changes that different communities have struggled with. We then analyze the different factors that could play into deploying each change within their given communities.

2.1 System Library

First, we consider an update to a `C` library (Lista 1) to get the system time. The library could update the time representation, i.e. the type alias `time_t`, from an `int` to a `long` in order to extend the range of possible values or to better reflect modern hardware.

Lista 1. C Library

```
  // time.h

- typedef int time_t;
- typedef long time_t;

  time_t time();
```

Lista 2. Library User

```
  // user.c

  set_deadline(int);

  int now = time();
  set_deadline(now + 3);
```

Nevertheless, as seen with recent attempts to update the `C` standard library [2], this is a change that could potentially affect many users of the library. Type aliases do not introduce a new type so the user code (e.g. Lista 2) might use `int` in its source, leading to a bug-prone narrowing or a compilation error. Fixing the type of `now` to use the alias might just push the problem throughout the codebase, in this example to the `set_deadline` call would inherit the problem after the change. There might be *hundreds* of locations that might need to be updated in a *single* project.

Even after finding all affected code, it is not always clear how to update it. In the case above, `set_deadline` might be in the interface of a different library, so updating the code could be hard or even impossible. Additionally, it could be part of the public interface of the user's library, so there may be constraints from transitive clients or their other projects. For widely distributed libraries, it might also be hard to reach all users affected by an incompatible change.

2.2 Browser API

Next, we look at a change to a Web API (Lista 3), specifically to the interface to add event handlers. A handler may *prevent* propagation of an event, so they usually need to be processed sequentially despite the significant impact on performance. To enable optimizations, there is an option to indicate if a handler ever `prevents` propagation, set conservatively to allow them. If most handlers are *passive*, the default could be changed to disable them and improve the navigation experience [3].

Lista 3. Browser API

```
  // window

  class Target {
    listen(event, cb, options) {
      let defaults = {
-       prevents: true;
+       prevents: false;
        ...
```

Lista 4. A Website

```
// user.js

let handler = event => {
  if (some_cond)
    event.prevent();
};

element.listen('tap', handler)
```

Clearly this change might break some websites. If anyone relied on the old behavior and attempts to `prevent` an event the site might break unexpectedly. However, not every maintainer will be able to understand or even learn of the subtle change. Browser developers won't be able to reach every user that owns or maintains a website. Even if they could, some of them might no longer be supported or might not have anyone with the technical expertise to understand and update the codebase.

Different browsers might also choose if and when they adopt the new default, so users might need to choose how to work around the offending browser. They have the alternatives , from dropping support for that specific browser (potentially no code change), to monkey patching the API to opt-out of the new behavior and its optimizations (likely a small change), or even to invest resources to update the codebase to take advantage of the update (significant changes).

3 Social Challenges

Having considered some examples, we can notice some common themes.

- **Scale**: As the number of affected projects grows, so does the complexity of coordinating and deploying changes across the community. Straightforward attempts to put any one team in charge of manually identifying and fixing all the affected source code quickly becomes intractable. This is one of the key reasons why some communities have produced automatic migration tools.
- **Priorities**: As a software ecosystem grows larger, there will be a variety of priorities within the community. Organizations have different needs and goals, and individuals within the organizations have their own values and motivations. Therefore, different segments of a community will exhibit different levels of support for the same change. This suggests that tooling can empower developers and users to control the propagation and adoption of updates.
- **Expertise**: As the scope of the community widens, so too does the diversity of expertise available in the ecosystem. Although this often produces an exciting and thriving environment, it also makes it harder to reach a shared understanding. A specific user might struggle to understand or implement a change imposed by a dependency. Similarly, a developer might be a domain expert, but not a language expert. This suggests that upgrade tools should be accessible and usable both for the developers aiming to deploy their upgrade *and* for users aiming to adopt an upgrade.
- **Visibility**: In many software ecosystems, developers cannot see all or most of the code that uses their software. Many users are within private organizations or simply do not share their work. If a developer or maintainer cannot access the uses of their software, it becomes impossible to assess the impact of changes or to aid in the upgrade process. This suggests that tools must account for unknown use cases or allow for privacy-preserving coordination.

4 An Alternative Community Structure

One of the few software ecosystems that, by construction, lacks many of the challenges above is corporate monorepos. For instance, in the Google monorepo [4], most source code is available in a single centralized repository, with perfect visibility into all users of a package. A majority of developers contributing to this ecosystem are software engineers by training, so they have a high degree of shared background and expertise. Finally, they are all part of the same company, so they may share the goal of maintaining the codebase. The main challenge of rolling out software changes in this ecosystem is therefore scale. So how do they deploy their software changes?

4.1 Pushing Upgrades

After realizing that many teams had to devote time to keep their software in a functional state after announced deprecations, Google implemented an internal *Churn Rule*. This policy requires that core library and API teams must do the work to upgrade their internal users or else keep their interfaces backward-compatible [4]. This had a number of advantages. The team making the changes to client code was the team that deeply understood what had changed and why. Also, only one team needed to know how to make the changes—in contrast to the prior approach, in which every affected team needed to learn how to make the changes—reducing duplicated effort across the organization.

Recent work by Google's `C++` team to incorporate type-safe time and duration interfaces into the codebase [5] offers an example of how this *push* model for software changes works in practice. To make the upgrade, the team developed a custom tool similar to a dataflow analysis that finds integral values that *sink* into low-level functions known to take a time or duration. The tool then incrementally converts these to the appropriate types. They automatically generated more than 20,000 changes over millions of lines of code. Before the *Churn Rule*, users of the infrastructure would have had to make these same tens of thousands of changes by hand, potentially even making the change too disruptive for the team to consider it useful in the first place.

It is not immediately clear how this process could be replicated outside of such an environment. The migration tool was difficult and time consuming to build. In a large and distributed open-source community, can developers muster the resources to go through this difficult tool building process? Outside of a corporate environment, developers may also lack the incentive to provide this level of support for all the users who depend on their software, especially outside of their organization. Without the monorepo itself, a migration tool may be hard to test and deploy. How would developers find all the affected source code? What about source code that is not publicly available? How would the tool or patches be distributed? Finally, the users themselves might have different preferences about the best way to adopt a potentially disruptive change.

5 Current Approaches

We now turn to describing current solutions adopted in more challenging software ecosystems as well as recent work that suggests promising directions.

5.1 Manual Coordination

The most common way of dealing with the social challenges around software changes is communication and negotiation within the community [6]. For example, developers might coordinate parallel releases to ensure that users have an easier time finding compatible versions of software. Project teams might plan feature releases and get community feedback to ensure changes are acceptable and visible to a wide swath of the ecosystem. Users themselves are often actively monitoring dependencies and the dependencies' communication channels, to plan ahead and receive forewarning about future potentially-disruptive changes.

Some amount of human intervention and coordination will always be required, but there are opportunities to improve these interactions. As an example, recent work has identified that many developers struggle to collect information about their users when designing interfaces [7].

5.2 Package Managers

The most common software-supported approach for managing disruptive changes in software ecosystems is the use of release versioning and package managers. A community sets a version convention and uses specialized tools to fetch and upgrade dependencies. Examples of widely used standards are Semantic Versioning[8] (i.e. `MAJOR.MINOR.PATCH`) and Calendar Versioning[9] (e.g. `YY.MM.DD`).

However, summarizing upgrade compatibility information into a few numbers has proven difficult. Researchers have identified that assessing the compatibility between releases is hard for non-language experts and that developers must make compromises when a change is unlikely to break most users [10]. Developers have also expressed that many simple changes (like renaming), become cost-prohibitive when it would require introducing a backward-incompatible release [6].

5.3 Source Transformation

Researchers are also actively exploring the use of domain-specific languages and tools for automating source transformations. An early example of this approach is TXL, a declarative language that uses a grammar and a set of rules to rewrite source code [11]. Recent examples of this approach include Cubix [12] and Comby [13], which are similar tools targeting multi-language transformations.

Many languages also have associated tooling available to perform source transformations. For example, the `C++` Clang-Tidy[14] extensible framework allows developers to specify custom diagnostics and even text-replacement fixes that can be automatically applied. The Go language goes a step further and includes a simple rule rewrite engine with their built-in `gofmt`[15] command.

Using these tools requires substantial expertise. Not only do users need to learn the interface to the tool, but they often need to understand low-level details of the language they are transforming. For example, TXL users must port the grammar and understand its representation details when specifying rules. Meanwhile, Clang-Tidy is already capable of understanding `C++`, but the user must learn the internal, Clang-specific AST representation (with its quirks).

5.4 Program Synthesis

Source transformation tools can also be extended via program synthesis techniques to lower the barriers to entry. `LASE` [16] and `REFAZER` [17] use input-output examples to synthesize generic and reusable AST edit rules. In follow up work, `Blue-Pencil` [18] is even capable of synthesizing edits in real-time and providing them as suggestions within an IDE.

As in many other program synthesis applications, it is not yet clear whether the generated edit scripts are complete or easy for developers to understand. This technique also relies on the availability and visibility of change adoption examples, which can be challenging for developers to obtain. Finally, it is not clear how programmers trying to use these tools to update their dependencies can exercise control over the edits to match their expectations or priorities.

5.5 Other Solutions

Researchers have also proposed many interesting approaches, specifically to centralize change management. For example, CatchUp! [19] is an Eclipse plugin that capture the use of IDE-provided refactorings to generate edit scripts, which can then be replayed by external dependants of the software when performing an upgrade. Another line of work produced a plugin to migrate between classes, given a mapping between equivalent methods [20]. However, these approaches are only able to express a limited set of transformations and have not seen major adoption.

6 Conclusion

Software ecosystems produce many *social* barriers to deploying changes. The streamlined process employed in some corporate monorepos suggests that there are opportunities to automate and reduce duplicated effort. We hope that by highlighting the social challenges inherent in modern software ecosystems, we can inform the design of the next generation of software upgrade processes and tools.

7 Acknowledgements

This work is supported in part by the NSF through grants CA-HDR 2029457, CA-HDR 1936731, CA-HDR 2033558, IIS-1546083, IIS-1955488, IIS-2027575, and CCF-1723352; DOE award DE-SC0016260; the Intel-NSF CAPA center; and gifts from Apple, Adobe, Facebook, Google, and VMware.

References

[1] M. Kim, N. Meng, and T. Zhang, "Software Evolution," in *Handbook of Software Engineering*, S. Cha, R. N. Taylor, and K. Kang, Eds. 2019, ISBN: 978-3-030-00262-6. DOI: 10.1007/978-3-030-00262-6_6.

[2] R. C. Seacord and NCC Group, "Add extended integer types without breaking the ABI," 2020. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2465.pdf.

[3] *Improving Scroll Performance with Passive Event Listeners*, https://developers.google.com/web/updates/2016/06/passive-event-listeners.

[4] T. Winters, T. Manshreck, and H. Wright, *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, 2020, ISBN: 9781492082743.

[5] H. K. Wright, "Incremental Type Migration Using Type Algebra," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020. DOI: 10.1109/ICSME46990.2020.00085.

[6] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016, ISBN: 9781450342186. DOI: 10.1145/2950290.2950325.

[7] T. Zhang, B. Hartmann, M. Kim, and E. L. Glassman, "Enabling Data-Driven API Design with Community Usage Data: A Need-Finding Study," in *CHI Conference on Human Factors in Computing Systems*. 2020, ISBN: 9781450367080. DOI: 10.1145/3313831.3376382.

[8] *Semantic versioning*, https://semver.org.

[9] *Calendar versioning*, https://calver.org.

[10] P. Lam, J. Dietrich, and D. J. Pearce, "Putting the Semantics into Semantic Versioning," in *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2020, ISBN: 9781450381789. DOI: 10.1145/3426428.3426922.

[11] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow, "TXL: A rapid prototyping system for programming language dialects," *Computer Languages*, vol. 16, no. 1, 1991, ICCL'88-Part I Computer languages: A perspective, ISSN: 0096-0551. DOI: 10.1016/0096-0551(91)90019-6.

[12] J. Koppel, V. Premtoon, and A. Solar-Lezama, "One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. DOI: 10.1145/3276492.

[13] R. van Tonder and C. Le Goues, "Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019, ISBN: 9781450367127. DOI: 10.1145/3314221.3314589.

[14] *Clang tidy*, https://clang.llvm.org/extra/clang-tidy.

[15] *Gofmt*, https://pkg.go.dev/cmd/gofmt.

[16] N. Meng, M. Kim, and K. S. McKinley, "LASE: Locating and Applying Systematic Edits by Learning from Examples," in *International Conference on Software Engineering (ICSE)*, 2013. DOI: 10.1109/ICSE.2013.6606596.

[17] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning Syntactic Program Transformations from Examples," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2017. DOI: 10.1109/ICSE.2017.44.

[18] A. Miltner, S. Gulwani, V. Le, A. Leung, A. Radhakrishna, G. Soares, A. Tiwari, and A. Udupa, "On the Fly Synthesis of Edit Suggestions," *Proc. ACM Program. Lang.*, no. OOPSLA, Oct. 2019. DOI: 10.1145/3360569.

[19] J. Henkel and A. Diwan, "CatchUp! Capturing and replaying refactorings to support API evolution," in *International Conference on Software Engineering (ICSE)*, 2005. DOI: 10.1109/ICSE.2005.1553570.

[20] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring Support for Class Library Migration," 2005. DOI: 10.1145/1094811.1094832.