Language-Level Support for Multiple Versions for Software Evolution

Tomoyuki Aotani[1], Satsuki Kasuya[2], Lubis Luthfan Anshar, Hidehiko Masuhara[2] and Yudai Tanabe[2]

[1]Sanyo-Onoda City University, Yamaguchi, Japan
[2]Institute of Science Tokyo, Tokyo, Japan

Abstract

While versioned packages are widely used in today's software development, existing programming languages allow to use them on the 'one-version-at-a-time' principle, which makes software evolution inflexible. We proposed a notion called *programming with versions* that programming languages should allow to use multiple versions of a package and designed programming languages based on this notion. In this presentation, we overview the design principle of these languages and discuss how these languages can make software evolution more flexible and future challenges in programming language design.

Keywords

Programming with versions, versioned software packages, package management systems, VL, BatakJava, Vython

1. Introduction

A *versioned package* is a unit of managing programs for modular software development. From a programming language, a package is often associated with a module that provides a set of classes, functions, types, variables, etc. Outside of a programming language, a package is often managed by a *package manager* in which a *version number* of a package is used for distinguishing compatibility between different implementations of the same package. There have been many package managers developed for different programming languages such as npm[1], Gradle[1], PyPI[1], RubyGems[1], Cargo[1], to name a few.

Versioned packages play an important role in software evolution. Modules of a large-scale software system are often managed as versioned packages so that each package can independently evolve without concerning about the other packages. The notion of compatibility, i.e., whether one implementation of a package can be replaced with another implementation, makes the developers easier to expect if they can use a new implementation of a package without having serious problems. This is especially so when a software system uses third-party packages.

However, current programming languages are associated with versioned packages in a limited and less-flexible way because most languages can import merely *one version of a package at a time*. While this limitation is reasonable (if there had been two implementations of a function, which one would be used?), it causes many problems in software evolution, some of which can be exemplified by the following scenarios.

- Alice is developing a web application and wants to update its application framework to a new version for better rendering API. However, the new version has many incompatible changes and her team had to modify many places in the application code even though they are irrelevant to the new feature.
- Bob is developing an enterprise system that uses a version of a network library that known to have a security flaw. However, his team cannot switch the library to a new secure version for a long time because the new version has many incompatible changes.

BENEVOL 2025: The 24th Belgium-Netherlands Software Evolution Workshop, November 17–18, 2025, Enschede, The Netherlands
aotani@rs.socu.ac.jp (T. Aotani); satsuki.kasuya@prg.is.titech.ac.jp (S. Kasuya); masuhara@acm.org (H. Masuhara); yudaitnb@prg.is.titech.ac.jp (Y. Tanabe)

[1]https://www.npmjs.com/, https://gradle.org/, https://pypi.org/, https://rubygems.org/, https://lib.rs/crates/cargo

- Charlie is developing an arcade game and try to use a numerical package for improving character movements. After writing the simulation code, he finds that the graphics library used in the game requires an older version of the same numerical package. Unfortunately, those two versions are not compatible with each other, and the language requires to choose one of them.

2. Programming with Versions

The authors propose the notion of *programming with versions* (PwV), which is to support *multiple* versioned packages at the level of programming languages [1, 2, 3, 4]. The principle is to allow to use multiple versions of a package in one program. From the programming language design point of view, it is not difficult to design a language that can use multiple versions. It is more difficult to *prevent inconsistent usage of multiple versions.* For example, when we use two versions of an encryption package, a data encrypted by one version must be decrypted by the same version. To this end, we developed the following language designs.

- We designed a functional PwV language [1, 3] where every function can have different versions of implementations. Its type system guarantees that a program, even though it uses multiple versions of implementations, each data is processed by the functions that are implemented in the same version in order to maintain consistency.
- We designed a class-based PwV language based on Java [2] where not only functions but also data can have different version of implementations. Its type system guarantees that a bundle of data, i.e., an object, has its own version, and is always processed by functions, i.e., methods, implemented in the same version.
- We designed a dynamically-typed PwV language based on Python [4]. Rather than relying on type systems to guarantee version consistency, this language dynamically detects version inconsistency based on data provenance.

3. Software Evolution with Multiple Versions at a Time

Though PwV languages need more work on their design and implementation, those languages will bring more interesting challenges into software evolution. Given a greater freedom of using new versions of packages, we need to guide developers so that they can *gradually* incorporate new versions into their systems. Then notion of compatibility should also be reconsidered. For example, it is the developer who judge compatibility for the semantic versioning. When versioned packages are integrated into programming language semantics, we could mechanically judge semantic compatibility with a theoretical background.

As there have been proposed a tremendous amount of methodologies for software evolution (with traditional programming languages), revisiting those methodologies under the light of PwV languages would also be interesting.

References

[1] Y. Tanabe, L. L. Anshar, T. Aotani, H. Masuhara, A functional programming language with versions, The Art, Science, and Engineering of Programming 6 (2021).

[2] L. L. Anshar, Y. Tanabe, T. Aotani, H. Masuhara, BatakJava: an object-oriented programming language with versions, in Proceedings of the International Conference on Software Language Engineering, SLE 2022, 2022, pp. 222–234.

[3] Y. Tanabe, L. L. Anshar, T. Aotani, H. Masuhara, Compilation semantics for a programming language with versions, in Proceedings of Asian Symposium on Programming Languages and Systems (APLAS 2023), LNCS, 2023.

[4] S. Kasuya, Y. Tanabe, H. Masuhara, Dynamic version checking for gradual updating, Journal of Information Processing 33 (2025) 471–486.