Loading Multiple Versions of an ASDF System in the Same Lisp Image

Vsevolod Domkin
vseloved@gmail.com

Abstract
In this paper, we present a proof-of-concept solution that implements consecutive loading of several versions of the same ASDF system in a single Lisp image. It uses package renaming to dynamically adjust the names of the packages loaded by a particular version of a system to avoid name conflicts on the package level. The paper describes the implementation, possible usage, and limitations of this approach. We also discuss the deficiencies of ASDF that impede its use as a basis for developing such alternative system manipulation strategies and potential ways to address them.

CCS Concepts: • **Software and its engineering~Software configuration management and version control systems** • *Software and its engineering~Software libraries and repositories* • *Software and its engineering~Software evolution*

1. Introduction

The problem of supporting simultaneous access to multiple versions of the same library in the same software artifact is relevant to the software projects that rely on many third-party components and/or have a long development time span. Due to the separate evolution of third-party libraries, the situations may arise when they may depend on different incompatible versions of software that share the same name. Besides, even the software project under direct control of the user itself may necessitate dependency of several versions of the same library that support different behaviors and functionality. This problem is often called dependency hell[1] (and, in different programming language environments, is known as "DLL hell," "jar hell" etc.) It manifests either in the inability to build the target software as a result of name conflicts or in the unsolicited redefinition of parts or whole functionality by the conflicting packages, which may happen silently or vocally, depending on the particular environment.

In Common Lisp, packages[2] provide namespacing capabilities to reduce the risk of name conflicts between symbols. The packages are first-class globally-accessible dynamic objects. Due to the existence of a centralized “registry” of known packages in the running Lisp image, name conflicts may arise when two independent software artifacts that include the definitions of the packages with the same names or nicknames are loaded into the same image. The conflict will manifest in the redefinition of the previously loaded package by the one loaded later, which will result in an extension of the package's external API and, possibly, an unexpected redefinition of parts of its functionality that have the same names (be it functions, classes or variables). This risk grows with the development of the library ecosystem, and such cases have been already reported[3] for the Quicklisp[4] distribution, which is the largest repository of Common Lisp open source libraries. An even higher risk of conflict exists not between independent pieces of software, but between different versions of the same software. In this case, a redefinition of the previous version of the package with a newer one may be intended (in case of upgrade), but if non-backward compatible changes are introduced, this will, potentially, mandate the upgrade of all of the package's dependents. Such situation may be undesired, especially in the case of third-party dependents that are not under the control of the user. Moreover, it may be beneficial to utilize both old and new versions of the upgraded package's functionality. The described risks are most critical for production software that is usually dependent on many external libraries and is produced via a process of automatic build (often using Continuous Integration[5] systems), not allowing for manual intervention in case of unexpected conflict.
The ways to approach dependency conflicts include administrative measures (adherence to a particular versioning or naming policy - see Semantic Versioning[6] or package renaming on incompatible changes proposal[7]) and

programmatic solutions. Not questioning the value of proper software development practices, it should still be noted that administrative measures have a crucial limitation of impossibility to fully regulate the activity of third-party developers, especially for the software that already exists and may not even be maintained at the moment. That is why a programmatic solution is essential, but, currently, there is no library or feature of an existing tool that allows dealing with them.

Most programming language environments do not provide a comprehensive user-friendly way of automating version conflict resolution due to the limitations of their namespacing capabilities (see, for example, the situation in Python[8]). One notable exception is the JVM, which allows to extend the standard classlloader[9] to dynamically load several classes that have the same name — the capability used by OSGi[10] to systematically handle version conflicts. Furthermore, the upcoming Java 9 will include the project Jigsaw[11] that introduces a new module system also capable of handling version conflicts by default. JavaScript is another interesting case as it initially lacked the concept of a package or module, and when it was later introduced via the Module pattern[12] and its derivatives, the standard objects were used to host modules with incapsulation of dependencies within the object's private scope that allows to not register the loaded dependency's name in the global public scope, when it is not necessary, thus preventing the version conflict altogether.

In Common Lisp, the low-level solution to conflicts of package name clashes is the standard `rename-package`[13] function. Using it allows possible to avoid name conflicts by changing the reference to the first of the conflicting artifacts before the second one is initialized. If the primarily loaded version of the package is renamed, a new one may be loaded without name conflicts. Such renaming, however, requires careful orchestration as the process of loading different packages is usually complex and not fully transparent, and the renaming should take place after the other packages, which are the users of the one being renamed, are loaded. This may not be possible in the general use case because of the potential redefinition and additions to the packages at program runtime. However, in the common case of loading the source code and then working with the image without any subsequent modifications to the dependencies, the renaming can be performed reliably.

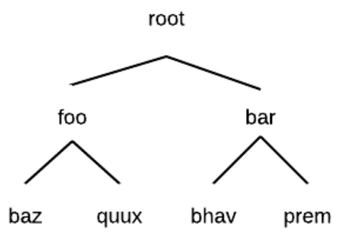
Packages are a source code level concept, while for the purpose of automation of the compilation and loading of the source code itself, a de facto standard abstraction provided in Common Lisp is a “system”[14]. It provides a way to group relevant source code files and other file-based resources and to specify the order of their compilation/loading. The currently adopted implementation of the system concept and related APIs is ASDF[15]. ASDF performs the similar role to make[16] and Ant[17] in other programming environments, and it allows for reproducible programmatic bundling, distribution, and initialization of both software libraries and applications. The system in ASDF supports the notion of version, which allows to logically distinguish different versions of the same software packages. It also allows specifying dependencies between systems (including versioned ones). Putting different packages (even with the same names) in different ASDF systems or putting different versions of the same-named package in different versions of an ASDF system allows to approach the problem of name conflicts, provided there is a way to control the loading of those systems and perform package renaming at the necessary points of the process. Currently, ASDF doesn't implement such functionality. Moreover, it has a number of key limitations preventing the implementation. First of all, at any moment in the running Lisp image, only a single version of a system may be accessible to ASDF. In case of an attempt to load another version (that may be discovered by ASDF even accidentally), several conflict resolution strategies may be utilized, the default being to load the system with the latest sysdef file access timestamp. This constraint is conditioned on the ASDF reliance on a central in-memory registry of known systems (similar to the package registry) that is a key-value store keyed by system names only, without the version information. Secondly, the ASDF approach to version conflict resolution is restricted to a single pre-defined strategy for determining the acceptable versions given a certain constraint[18].

To sum up, there is no end-to-end solution to potential system-level name and version conflicts in the Common Lisp environment, but it is desirable in order to support future growth of the Lisp library ecosystem and large-scale projects. The approach should support ASDF systems. Consequently, the proposed solution is based on the standard `rename-package` and low-level ASDF APIs.

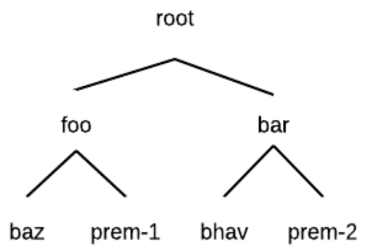
2. Possible conflict scenarios

In order to validate the correctness of a version conflict resolution approach, the following conflict scenarios should be analyzed. More complex possible configurations will be a combination of these primitive cases.

1. "Zero" scenario. No name conflicts. A fallback to `asdf:load-system` is expected.

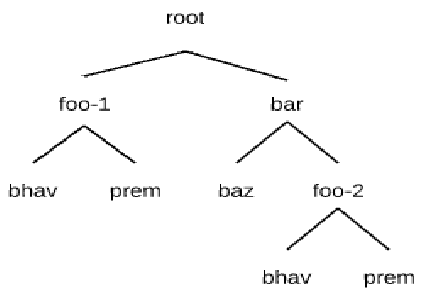


2. "Basic" scenario. There is a single name conflict `between` prem `v.1 (required by` foo) and v.2 (required by

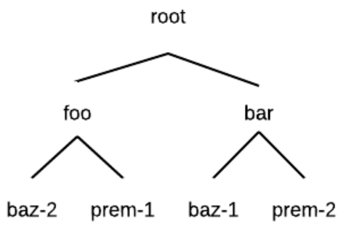


`bar`).

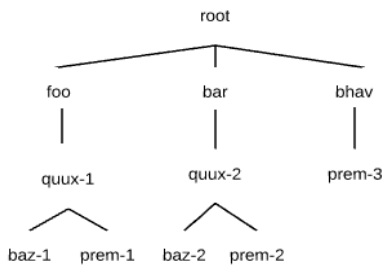
3. "Subroot" scenario. There is a single conflict (in system `foo`), and one of the conflicting packages is a direct dependency of the root system.



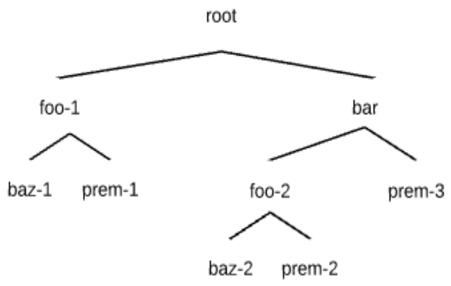
4. "Cross" scenario. There are 2 conflicting systems at the same level in the dependency tree: `prem` and `baz`.



5. "Inter" scenario. There are 3 conflicting systems with one of them (`quux`) being the dependent on the two others: `baz` and `prem`.



6. "Subinter" scenario. There are 3 conflicting systems with one of them (`foo`) being the dependent on the two others (`baz` and `prem`), and one of the conflicting systems (`foo`) a direct dependency of the root system.



3. Implementation

We propose an ASDF-compatible algorithm for conflict-free loading of a particular system's dependencies with on-demand renaming of their packages in case of discovered name/version conflicts happening at the right moment in the program loading sequence. The algorithm comprises of the following steps:

1. Assemble a dependency tree for the system to be loaded based on ASDF systems' dependency information and, using it, discover the dependencies,

which produce name conflicts.

2. In case of no conflicts, fallback to regular ASDF load sequence.
3. In case of conflicts, for each conflicting system determine the topmost possible user of the system in the dependency hierarchy that doesn't have two conflicting dependencies (the one, below the lowest common ancestor of the conflicting systems).
4. Determine the load order of systems using topological sort with an additional constraint that, among the children of the current node of the dependency tree, the ones that require conflict resolution will be loaded last.
5. Load the systems' components (plain load without loading the dependencies) in the selected order caching the fact of visiting a particular system to avoid multiple reloading of the same dependencies that are referenced from various systems in the dependency tree.
6. During the load process, record all package additions and associate them with the system being loaded.
7. After a particular system has been loaded, check whether it was determined as a point of renaming for one or more of its dependencies, and perform the renaming.

In step 4, load-last strategy is necessary for the renaming of the alternative system to happen before the load of the current one: in case of the opposite order, the current system will be loaded but not renamed, as the renaming will happen only after load of the parent node, which will result in a name conflict. This is relevant to the Subroot (4) and Subinter (6) text scenarios.

The algorithm is implemented in the function `load-system-with-renamings`[19] that is summarized in Figure 1. It operates on the instances of a `sys` structure that is used as a simple named tuple: `(defstruct sys name version parent)`.

```
(defun load-system-with-renamings (sys)
  (multiple-value-bind (deps reverse-load-order renamings)
      (traverse-dep-tree sys)
    (when (zerop (hash-table-count renamings))
      (return-from load-system-with-renamings (asdf:load-system sys)))
    (let ((already-loaded (make-hash-table :test 'equal))
          (dep-packages (make-hash-table)))
      ;; load dependencies one by one in topological sort order
      ;; renaming packages when necessary and caching the results
      (dolist (dep (reverse reverse-load-order))
        (let ((conflict (detect-conflict)))
          (when (or conflict
                    (not (gethash (sys-name dep) already-loaded)))
            (renaming-packages
             (if conflict
                 (load-system dep)
                 (load-components (asdf:find-system (sys-name dep)))))
          (unless conflict (setf (gethash name already-loaded) t))))))))
```

Figure 1. Source code for the `load-system-with-renamings` procedure

In the actual function, the `renaming-packages` and `detect-conflict` macros are implemented in-place, but, here, for the sake of clarity, they are extracted. `detect-conflict` is omitted as it is trivial to implement, and `renaming-packages` is listed separately (see Figure 2). That's why it references the seemingly free (but, in fact, the parent's) `dep` and `dep-packages` variables.

The `traverse-dep-tree`[19] function implements the first stage of the algorithm: building a dependency tree, discovering conflicts and arranging the dependencies in proper order for loading. It recurses on the current system's dependencies and keeps a set of the encountered systems and their versions to spot version conflicts via set intersection with a specialized `:key` function that takes into account different system versions.

We also provide an alternative to the ASDF's implementation of system loading facility in `load-system` and `load-components` functions.

```
(defmacro renaming-packages (&body body)
  `(let ((known-packages (list-all-packages)))
     ,@body
     ;; record newly added packages
     (setf (gethash dep dep-packages)
           (set-difference (list-all-packages) known-packages))
     ;; it's safe to rename pending packages now
     (dolist (d (gethash dep renamings)))
       (let ((suff (format nil "~:@(~A-~A-~A~)"
                           (sys-version d) (sys-name dep) (gensym))))
         (dolist (pkg (gethash d dep-packages))
           (rename-package pkg (format nil "~A-~A"
                                       (package-name package) suff)
                           (mapcar (lambda (nickname)
                                     (format nil "~A-~A" nickname suff))
                                   (package-nicknames pkg)))))))
```

Figure 2. Source code for the `renaming-packages` macro

3. Working around ASDF

The initial assumption for the development of this algorithm was to build it on top of the public ASDF API as an alternative system loading strategy. However, during its implementation, several obstacles were encountered in ASDF, which forced us to develop alternative procedures to the existing ASDF public counterparts, using the low-level internal ASDF utilities.

The main blocker was an ASDF's core choice to have a central registry of known systems that uses unversioned system names as keys.

In a lot of ways, ASDF is very tightly-coupled and not transparent:

- The source code, in general, is rather extensive and abstraction-heavy, but not well-documented.
- Most of the ASDF `actions`, even the ones that could be implemented in a purely functional manner (for instance, `find-system`), trigger internal state changes.
- The ASDF operations class hierarchy is based on a number of abstract classes, such as `downward-`, `upward-` or `sideway-operations`, which form implicit interdependencies between concrete operations, but this is not documented in a clear manner.
- The ASDF operations are performed not directly, but according to an order specified in a `plan`[20] object. The plan API is also not documented.
- ASDF caching behavior is undocumented.

This makes ASDF a monolithic tool tuned towards implementing a particular strategy of handling systems, which is substantially hard to repurpose in order to support alternative strategies, using the existing machinery for system discovery, orchestration of compilation, and loading of single files. Consequently, there are many unexpected omissions from the ASDF public API. Here are a few that were encountered in the process of this work:

- There is no direct way to load a system from a specific filesystem location: only a system that is previously found using the ASDF algorithm can be loaded.

- There is no direct way to enumerate all potential candidate locations for loading a system: each ASDF system search function should terminate the discovery process as soon as it finds a candidate.
- There is no direct way to find a system with a specified version: the version argument to ASDF operations may only be used as a constraint that the current candidate system should satisfy, not as a guide for selecting the candidate.
- There is no direct way to load just the source files for the system's components without checking and, possibly, reloading its dependencies: calling `load-op` on a source file invokes implicit operation planning machinery that is specified partly in the operations hierarchy and partly in the associated generic functions, and it will cause the call to `prepare-op` on the same file, which triggers `prepare-op` on the whole system, which, in turn, checks the system's dependencies and may invoke their full reload. Overall, a simple call to `(operate 'load-op <component>)` may produce a call stack of 10 or more levels of just ASDF operations.
- It is impossible to read the contents of an ASDF system definition without changing the global state, although this is often needed to determine some property of the candidate ASDF system, like its version or set of dependencies.

As a result, we had to define a number of utility functions to patch the missing parts of ASDF, which, definitely, are not well-integrated with the current vision of how ASDF parts should play together, and which use a number of private ASDF utilities that might be changed or removed in the next versions. Such approach is, obviously, not scalable and, ideally, the implementation should be performed based solely on the ASDF public API. But, to allow that, the API has to be expanded significantly, which will require some major changes to ASDF core: adding deeper support for versions, decoupling some of the functions, and making others less dependent on side effects.

Below is an example of some of the alternatives to ASDF operations that we have developed. The `sysdef-exhaustive-central-registry-search` (see Figure 3) is a version of `asdf::sysdef-central-registry-search` that doesn't stop as soon as the first candidate ASD-file is found. It is used instead of `asdf:search-for-system-definition`, and has a drawback of limiting the search to only the legacy `*central-registry*` locations.

```
(defun sysdef-exhaustive-central-registry-search (system)
  (let ((name (asdf:primary-system-name system))
        rez)
    (dolist (dir asdf:*central-registry*)
      (let ((defaults (eval dir)))
        (when (and defaults (uiop:directory-pathname-p defaults))
          (let ((file (asdf::probe-asd name defaults
                                       :truename asdf:*resolve-symlinks*)))
            (when file (push file rez))))))
    (reverse rez)))
```

Figure 3. Source code for the `sysdef-exhaustive-central-registry-search` function

We, also, had to resort to an interesting way of getting a record for a specific system by its ASD-file (see Figure 4) as a part of an alternative implementation of `find-`

`system`. Unfortunately, there is no ASDF function that will load an ASD file and return a list of ASDF system objects for the systems defined in it.

```
(asdf:load-asd asd)
(cdr (asdf:system-registered-p system))
```

Figure 4. Source code for the `sysdef-exhaustive-central-registry-search` function

Finally, in Figure 5 you may find a workaround to shortcircuit the ASDF operations' interdependency mechanism and prevent it from performing any other actions except directly loading the components of a current system. In general, it is a sign of excessive code coupling when a simpler operation requires more code than a more complex one, which includes it.

```
(defparameter *loading-with-renamings* nil)
(defmethod asdf:component-depends-on :around ((o asdf:prepare-op)
                                               (s asdf:system))
  (unless *loading-with-renamings*
    (call-next-method)))
(defun load-components (sys)
  (let ((*loading-with-renamings* t))
    (dolist (c (asdf:module-components sys))
      (asdf:operate 'asdf:load-op c)))
  t)
```

Figure 5. Source code for the simplified mechanism of ASDF components loading

To sum up, the current version of ASDF is tightly-coupled and lacks referential transparency at core, while at the middle level it's not well-documented and lacks a comprehensive API that could be used for the development of alternative top-level system management utilities based on a solid foundation of ASDF's system discovery and individual component manipulation machinery.

4. Limitations of the Solution

The proposed solution is, primarily, intended for the use case of loading the whole target system at once without future modifications of its dependencies in-memory, which is necessitated by production build environments. The alternative system load scenarios, that are, mostly, interactive and allow for the programmer to remain in control of the environment, resorting, in case of conflicts, to manual intervention ranging from explicit renaming to changing the source code of the conflicting dependencies and "vendoring" them as part of the project, are not in such desperate need of an automatic solution.

Our approach has a number of limitations that should be listed to avoid unexpected and unexplainable edge cases. The risk of their manifestation in the intended environment is low, but, nevertheless, the users should be aware of the possible shortcomings.

The first limitation is the passive mechanism of capturing package changes after-the-fact, which is not transactional. Parallel invocation of `load-system-with-renamings` has a race condition. The critical section is the process of recording the changes to the global package table in `renaming-packages`. To remove the limitation, this part may be protected by a mutex. This is not done in the presented code to avoid additional complexity. Ideally, the sequential and parallel versions of this procedure should be provided with the sequential one being the default. An alternative solution would be to perform full source code analysis of the system to be loaded in order to determine, which packages will be defined in it. Such complexity is definitely an overkill.

Our approach also relies on the assumption that all the packages from the currently loaded systems where not defined previously. It is a reasonable constraint for the vanilla production environment, which may, however, be violated during an interactive session. Unfortunately, the only measure that may be taken here is a disciplined approach to package loading. At least, the Lisp compiler will issue a warning on package redefinition, which will alert the programmer that the name conflict has occurred. It is possible to expand the loading function to intercept this warning and terminate its operation, if necessary.

Elaborating on this point, it should be also obvious that this procedure will not be able to catch changes to existing packages (that may be regarded as monkey-patching, in this context). It is debatable, whether such changes should be prevented by our system, as their purpose is usually contradictory to the idea of immutable dependencies that our solution upholds.

Next potential issue is associated with implicit transitive dependencies: if a system *foo* depends on *bar* and *quux*, and *bar* also depends on *quux*, in ASDF system definition, it is sufficient to list only *bar* as *foo*'s direct dependency. This implicit dependency may break if *quux*'s packages are renamed during the loading process: according to the algorithm, the renaming will happen directly after loading of *bar*. In such situation, all the references to *quux*'s packages in *foo*'s code will be invalidated, as they will be read when the packages will not be accessible by the old canonical names. However, such situation is relevant only to the newly defined systems that are under full control of the developer, as for the deeper dependencies there should be no version conflicts, as they would not have allowed the system to be built by the normal ASDF procedure, and conflicts introduced when combining multiple dependencies are resolved at the topmost level by the algorithm, thus not effecting the dependency subtrees of the combined systems. Adding an explicit dependency on *quux* in *foo* allows solving the problem in a straightforward way.

Also, our approach doesn't address the possibility of two independent packages having the same name and version, but it, probably, should be handled not at the code-base level, but rather the social one. Additionally, our conflict-finding mechanism may be extended to catch such case.

Finally, the additional minor inconvenience is that the conflicting packages will be available under altered names, which can be discovered from the environment but are not apparent. This may impede interactive redefinition, monkey-patching, hot-patching and other interactive programming practices that might occasionally be of interest to the user. It will surely break the code relying on runtime manipulations using `intern` or `eval`: the references to the renamed packages in the code not yet evaled will be invalidated after the renaming, unlike the references in the code that was read and loaded, which will be associated with new names automatically.

5. Conclusions and Future Work

Our name conflict resolution algorithm and its proof-of-concept implementation provide a feasible solution to potential dependency hell problems for Common Lisp software, specifically targeted to production environments. The paper also explores its corner cases, which require special handling. Although this solution may not be final, it is already usable in the environments that are faced with dependency conflicts.

The proposed approach has several directions of improvement:

- It should be expanded to cover other non-load-based scenarios of system manipulation. In particular, for the compilation use case, the implementation should be almost identical.
- It should be made more compatible

with ASDF (provided ASDF is also changed to be less hostile to such solutions).

Exploration of the possible implementation strategies for this program also helped uncover the deficiencies in the current implementation of ASDF and showed one of the directions for its future development. ASDF is underutilizing its position as a de facto standard toolkit for dependency management in Common Lisp by not providing a comprehensive API for manipulation of both systems and system definition files. In order to allow for this algorithm and other possible non-default build strategies to be implemented on top of ASDF public API, a number of changes are necessary. In general, those should include decoupling of the ASDF code base (specifically from the assumptions of a 1-to-1 mapping of a system record in the ASDF registry to a system currently loaded), comprehensive documentation of the `plan` and `action` APIs, development of utility wrapper functions for common middle-level actions, and a general review of the public API according to the scenarios we'd like it to support.

References

[1] “Dependency Hell” definition - https://en.wikipedia.org/wiki/Dependency_hell
[2] Pitman, K. M., Common Lisp HyperSpec, 1996. Chapter 11. Packages - http://www.lispworks.com/documentation/lw50/CLHS/Body/11_.htm
[3] Nickname collision: bordeaux-threads and binary-types - https://github.com/quicklisp/quicklisp-projects/issues/296
[4] Beane, Z., Quicklisp - http://quicklisp.org
[5] “Continuous Integration” definition - https://en.wikipedia.org/wiki/Continuous_integration
[6] Semantic Versioning - http://semver.org/
[7] Vodonosov, A., Backward compatibility of libraries (case study in Common Lisp) - https://www.european-lisp-symposium.org/editions/2016/lightning-talks-1.pdf
[8] Kernfeld, P., The Nine Circles of Python Dependency Hell - https://tech.knewton.com/blog/2015/09/the-nine-circles-of-python-dependency-hell/
[9] Liang, S., Bracha, G., Dynamic class loading in the Java virtual machine - Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA ’98, pages 36–44, 1998 - http://www.humbertocervantes.net/coursdea/DynamicClassLoadingInTheJavaVirtualMachine.pdf
[10] OSGi service platform core specification - http://www.osgi.org/
[11] Project Jigsaw. Project Jigsaw website - http://openjdk.java.net/projects/jigsaw/
[12] Herman, D., Tobin-Hochstadt, S., Modules for JavaScript Simple, Compilable, and Dynamic Libraries on the Web - homes.soic.indiana.edu/samth/js-modules.pdf
[13] Pitman, K. M., Common Lisp HyperSpec, 1996. Function RENAME-PACKAGE - http://www.lispworks.com/documentation/HyperSpec/Body/f_rn_pkg.htm
[14] ASDF System - http://www.cliki.net/ASDF+System
[15] ASDF - https://common-lisp.net/project/asdf/
[16] make. Gnu make website - https://www.gnu.org/software/make/
[17] ant. Apache ant website - http://ant.apache.org/
[18] Rideau, F.-R., Goldman, R.P., ASDF Manual. Chapter 7.4 Functions, Function: version-satisfies - https://common-lisp.net/project/asdf/asdf/Functions.html
[19] Domkin, V., ASDFx - https://github.com/vseloved/asdfx/blob/master/asdfx.lisp
[20] Rideau, F.-R., Goldman, R.P., ASDF Manual. Chapter 7. The Object model of ASDF - https://common-lisp.net/project/asdf/asdf/The-object-model-of-ASDF.html