Alire: a library repository manager for the open source Ada ecosystem

Alejandro R. Mosteo[1,2]

[1] Instituto de Investigación en Ingeniería de Aragón (I3A)
amosteo@unizar.es,
Mariano Esquillor s/n, 50018, Zaragoza, Spain
[2] Centro Universitario de la Defensa de Zaragoza (CUD)
Ctra. de Huesca s/n, 50090, Zaragoza, Spain

Abstract. Open source movements are main players in today's software landscape. Communities spring around programming languages, providing compilers, tooling and, chiefly, libraries built with these languages. Once a community reaches a certain critical mass, management of available libraries becomes a point of contention. Operating system providers and distributions often support but the most significant or mature libraries so, usually, language communities develop their own cross-platform software management tools. Examples abound with languages such as Python, OCaml, Rust, Haskell and others.
The Ada community has been an exception to date, perhaps due to its smaller open source community. To advance in this direction, this work presents a working prototype tailored to the Ada compiler available to open source enthusiasts, GNAT. This tool is designed from two main principles: zero-cost infrastructure and a pure Ada implementation. Initially available for Linux-based systems, it relies on the semantic versioning paradigm for dependency resolution and uses Ada specification files to describe project releases and dependencies.

Keywords: Library Management, Dependency resolution, Open Source, Ada 2012

1 Introduction

"If I have seen further it is by standing on ye sholders of Giants" wrote Sir Isaac Newton in a letter to Robert Hooke [9]. Believers in the virtues of open source licenses may recognize the sentiment; in nowadays rapidly evolving technological landscape, reuse of code is critical to adapt to new technologies, avoid past errors, stay on top of vulnerabilities, and foster collaboration. In the communities built around programming languages this can be seen in the publishing of software under more or less permissive licenses [11], but free of charge. Open source programmers want their code to be run and built upon.

However, the availability of code and simplicity of distribution, compared to pre-Internet generalization, has brought with itself its own problems, such as a difficulty to be aware of available libraries, obsolescence of code that becomes unmaintained (a form of *bit rot* [10]) and incompatibilities between versions of a same library, or among different libraries being used simultaneously.

To address those problems, one of the most notable efforts in the open source world are the different Linux distributions. Either based on distribution of source code, like Gentoo [17], or of binaries, like Debian [2], these communities have since long dealt with the problem of packaging consistent systems for different architectures. The difficulty of such task is captured in the *dependency* or *DLL hell* expressions [6], and one of the most dreaded experiences is ending in a *broken* configuration during and upgrade.

Programmers, however, do not all use the same distribution, nor even the same operating system, since today they can cater to about half a dozen generalist operating systems. Given the polarizing nature of programming languages [15] it is then unsurprising that many languages have seen efforts aimed at providing an easy way of distributing libraries for those languages, as we shall discuss in Section 2. In some cases, like Rust [8], the tool for the distribution of libraries is an integral effort of the team developing the language.

The Ada language, perhaps because of its ties to closed development and today's considered niche place in the language landscape [7], has not seen such a tool appear (to the best of our knowledge), despite the notable amount of open source libraries available [3]. This work presents a tool that could be a first step in this direction, with the main contribution being the tool itself. In the technical aspect, the tool tries to appeal to the Ada programmer by using native Ada code to describe releases and its dependencies, thus avoiding the need to learn new formats. To use this information, the tool uses self-compilation to incorporate the required data into its catalog of libraries. Finally, as a byproduct, we contribute a semantic versioning library[3] that is used to describe dependencies among releases.

The project started as an informal discussion[4] under the name of Alire (from Ada Library Repository), and this work reflects the views of the authors on how a minimal tool that addressed the low-hanging problems of the open source Ada community could be brought to life. The tool itself is termed `alr`[5], in the vein of other reputable command-line tools such as `git`, `svn`, etc., and to distinguish it from the general project.

The paper is structured as follows: Section 2 examines the situation in other languages and points the referents taken for this tool. Section 3 presents the ideas adopted for the design of `alr`. Next, Section 4 presents details about the implementation mechanisms underpinning the design. A brief discussion follows on the open questions this design leaves and, lastly, concluding remarks and future directions close the paper in Section 5.

[3] `https://bitbucket.org/aleteolabs/semver`

[4] `https://github.com/mosteo/alire/issues`

[5] A monospace font is used throughout the paper to denote actual executable commands or logical entities such as files.

2 Related Work

The problem of library distribution has been tackled in two main ways, namely distribution of binaries and of source code. The former has the advantage of speed for the user, because it saves the step of compilation. The latter allows the complete tailoring of the building process to one's environment, and reduces the work load and hardware requirements on maintainers. For purely interpreted languages the only available way is the distribution of sources.

Once libraries are obtained, we see yet two possibilities: installation of packages system-wide, as if they where integral parts of the platform, or local installation in a confined or user sandbox (that sometimes can be the default user environment).

In Python's `pip` [13], for example, libraries are installed globally if run as superuser. If run as a regular user, they will be installed in the user's environment. These two options present to the user a default environment that can become broken [6] when dependencies are improperly managed, and for that reason it is recommended [13] to use a sandbox or virtual environment for each development context (which in turn entails possible library duplication in several virtual environments). Some packagers, like OPAM [18] or Nix [5], avoid that duplication by using a common store where individual releases are isolated (i.e., there is not a "current" version of any library).

When one inspects the many solutions out there, like Rust's `cargo`, Python's `pip` and `easy_install`, OCaml's `opam`, D's `dub`, Haskell's stackage and `cabal-install`, to name a few, a few common traits arise. The backend is usually some kind of database that in its simplest form is merely a set of files under version control in a public repository or in dedicated servers. Submission of new libraries becomes then the merging of a pull request into the stable branch of the catalog. Fetching of a library involves the download of a file bundle or checkout of a particular commit.

The other salient aspect these tools address is dependency resolution. When building a project with a complex set of dependencies, it may happen that two (or more) subprojects depend on the same libraries with some version restrictions. From all the possible combinations, only one that satisfies all dependent projects can be chosen, or if an incompatible request is made a resolution conflict appears. Again, a common approach is to use semantic versions [12] of the form `M.m.p`, where `M` stands for *major* version (one that is backwards incompatible), `m` is the *minor* version (one that is backwards compatible within the same major version) and `p` is a *patch*, a mere bug fix release that should be API compatible with other `M.m` releases. These dependencies are usually represented in some textual description of a release, like key-value lists, JSON, XML, or the own language syntax when it is interpreted.

3 Design objectives and basics of operation

For `alr`, after reviewing these solutions, the following decisions were taken for a bare minimum working tool:

- The objective is to help develop software, but not to configure the system. As such, it will not install compiled libraries, entirely avoiding any possibility of breaking the user's system.
- To avoid redundancy of code but also broken dependencies, OPAM's approach is used in which every distinct library release is stored under a unique name, although in the same caching folder.
- To not depend on private servers, the Alire catalog and code releases are stored in public Version Control System (VCS) services such as GitHub, BitBucket, etc., with which the open source community is used to work.
- New releases are incorporated into Alire by means of a pull request into the catalog repository. Since this is a manual process, at this time Alire can only be considered a curated system[6].
- An indexed release is described using Ada code that is verified by means of compiling it, relying only on a single specification file that is part of the `alr` source code. The aim is to stay within the Ada realm as much as possible. In its present form, the `alr` tool only requires familiarity with the GNAT [14] compiler.
- Library developers should be minimally impacted for integration into Alire, if at all. This is achieved ultimately by only requiring a GNAT project file (GPR file) that could be created by Alire maintainers without bothering library authors uninterested in this tool.

Ada was (again) a pioneer when compared with languages designed even later by adopting the idea of library items [16] that can be submitted to the compiler independently. This concept, together with the well-defined dependency and elaboration rules, has spared Ada developers another quagmire which is the dependency-building tools such as autoconf, automake and CMake [1]. Given that nowadays there is a single open source Ada compiler, namely GNAT in its GPL and FSF editions, at this time `alr` relies on GNAT aggregate project files to completely manage the building process, without the need to modify the environment. This solution lets programmers use dependencies as usual, merely "with-ing" their project file.

3.1 Components of the Alire project

Alire has adopted an initial zero-cost approach, which structures the project in the following parts:

[6] The same happens in other languages. For example, in the Haskell community the Stackage project arose as a curated alternative to the `cabal-install` breakage-prone tool.

Listing 3.1. Help screen of `alr`

```
Ada Library Repository manager
Usage : alr command [options] [arguments]

Valid commands:

        build   Upgrades and compiles current project
        clean   GPRclean current project
        compile GPRbuild current project
        get     Fetches and optionally compiles a project
        help    Shows hopefully helpful information
        init    Creates a new project with alr metadata
        lock    Lock dependency versions
        run     Launch the current project executable
        search  Search a string in release names
        update  Updates alire catalog and project dependencies
        version Shows alr compilation metadata

Use "alr help [command]" for more information about a command.
```

- The catalog of projects is a repository hosted under the name of `alire`. It fulfills the same role as, e.g., the `crates.io-index`[7] project in the Rust community. It comprises the database of known projects and the minimal Ada types needed to represent that information. This way, commits to its repository should be for the most part, once development stabilizes, just additions to the catalog.
- The command-line tool available to users to interact with the Alire catalog is named `alr`, as is its own project repository. Again, this allows development on the tool with minimal disturbance to the catalog. It fulfills the role of the `cargo`[8] tool for Rust.
- The code releases from third parties can be in any online repository, with the implicit assumption that the longest lived a repository is, the better. Current free offerings favored by developers are the usual suspects: GitHub, BitBucket, GitLab, etc. Of course, forks of particular releases could be made to ensure high availability.

3.2 Introduction to `alr`

The prototype being discussed in this work is available at BitBucket[9]. Once installed and run without arguments, the user is greeted by the help screen shown in Listing 3.1, which will not be unfamiliar to `cargo` or `opam` users.

Before diving into these commands, an explanation on the terminology being used (in the remainder of the paper and in the Alire source code) is in order (see also Fig. 1):

[7] `https://github.com/rust-lang/crates.io-index`
[8] `https://github.com/rust-lang/cargo`
[9] `https://bitbucket.org/aleteolabs/alire`

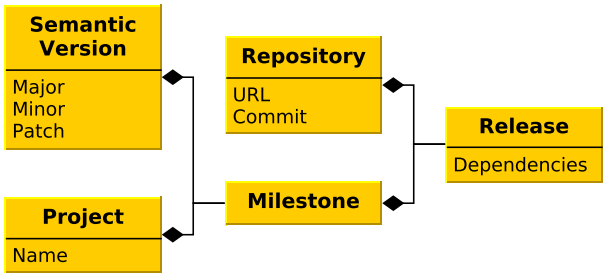

Fig. 1. Entities in the Alire catalog.

- A *project* refers to what also is typically called a library in the software world; e.g., GtkAda[10], AWS[11], etc.
- A *milestone* is a project name plus a semantic version; i.e., a particular version of a project.
- A *release* is the actual materialization of a milestone, available from somewhere online (usually a VCS server). Internally, each project must provide at its root a `project_name.gpr` file that builds the project.

The most straightforward function of `alr` is to retrieve a particular project and build it. Projects can contain libraries, which are useful to other projects, but also executables, in which case the compilation process will result in a file ready to be run. This is achieved with the `alr get <project>` command. The result will be a folder containing the requested project, but also its dependencies will be downloaded to the the dependency caching folder, so compilation should immediately succeed.

Alternatively, `alr` can create new projects to start easily working within the Alire ecosystem. This is achieved with the `alr init [--bin|--lib] <project>` invocation. Initially the project will depend only on Alr itself, for reasons explained in Section 4, but this dependency is not mandatory and will typically be replaced by the user with any projects needed by the newly created one.

Any project obtained by each of these two means can be called an alr-enabled or aware project, since it contains a metadata file that allows `alr` to perform its functions. Once within the folder tree of an alr-aware project, we can use the rest of commands (see Fig. 2). The `compile` command launches the `gprbuild` tool with a generated project file that makes dependencies available. The `update` command refreshes the catalog and obtains if necessary a compatible set of projects needed by the current one.

[10] `https://github.com/AdaCore/gtkada`
[11] `https://github.com/AdaCore/aws`

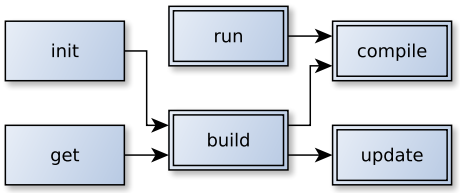

Fig. 2. Relationships among commands. Single-frame commands can be used anywhere in the filesystem, whereas double-framed ones are to be used within an alr-enabled project.

There are also compound commands that group functions for common combinations: `run` will compile and then launch the resulting executable, whereas `build` will ensure that dependencies are up to date to then compile the project.

The commands interrelations have been designed to guarantee success, in the sense that compilation should always succeed if the requested dependencies are valid. `alr` will detect the addition of new dependencies by the user and fetch them before a new compilation.

To conclude this section, we show how dependencies are represented in a just-created project. As advanced, this is done in a package specification that can be compiled to verify its correctness, and which is initially generated by `alr`:

Listing 3.2. Metadata file in an alr-enabled project.

```
with Alr.Project; use Alr.Project;

package Shiny_Project_Alr is

   Working_Release : constant Release := Set_Root_Project (
      "shiny_project",
      Depends_On =>
         At_Least_Within_Major ("alr", V ("1.0.0")));

end Shiny_Project_Alr;
```

The only dependency is on the Alr project itself, precisely so this file can be compiled. Once the syntax of this file is correct this dependency can be removed, since the user project itself does not need to compile this file (although `alr` does, so it must remain compilable).

Restrictions on dependencies are described using high-level functions. This way there is no possible confusion on what is being asked for. In the example, we request any future version of Alr that is within the same major number, hence backwards-compatible[12]. Listing 3.3 shows other restrictions on versions that are currently available.

[12] This is the caret "^" operator in other semantic versioning implementations

Listing 3.3. Main declarations in the Semantic Versioning support library.

```
package Semantic_Versioning with Preelaborate is

   subtype Version_String is String
     with Dynamic_Predicate => (for all S of Version_String => S /= ' ');

   type Version is private;
   --  A version is a major, minor and patch number
   --  Optionally it may include pre-release name and build metadata, e.g.:
   --  1.2.0-alpha+c3423fab

   type Version_Set is private;
   --  A collection of versions (usually a compatible subset)

   function New_Version (Major : Natural;
                         Minor : Natural := 0;
                         Patch : Natural := 0;
                         --  Optional parts:
                         Pre_Release,
                         Build : String := "") return Version;
   -- Refer to http://semver.org/ for the exact meaning of each part, but:
   -- A change of major version implies API incompatibility
   -- A change of minor version implies backwards compatible changes
   -- A change of patch version implies fixes with no new functionality

   function "<" (L, R : Version) return Boolean;
   -- Refer to http://semver.org/ for the exact ordering. Most notably:
   -- A version with pre-release tag is earlier than its regular version.
   -- Build info is not taken into account to determine ordering.

   function At_Least_Within_Major (V : Version) return Version_Set;
   function At_Least  (V : Version) return Version_Set;
   function At_Most   (V : Version) return Version_Set;
   function Less_Than (V : Version) return Version_Set;
   function More_Than (V : Version) return Version_Set;
   function Exactly   (V : Version) return Version_Set;
   function Except    (V : Version) return Version_Set;

   function "and" (VS1, VS2 : Version_Set) return Version_Set;

   function Is_In (V : Version; VS : Version_Set) return Boolean;
   --  Checks membership

private
```

4 Implementation details

This section presents some lower level details on `alr` implementation, particularly those aspects that present a specific idiosyncrasy of the tool when compared with its homologues for other languages.

GNAT is currently the only open source Ada compiler available, and its project files are the preferred way to conveniently manage the building process. For these reasons, `alr` takes advantage as much as possible of GNAT project files, and in particular uses aggregate projects to select the dependencies to be included in the compilation of a project.

4.1 Alire-mandated files

For `alr` to be able to perform its project-specific commands (see Fig. 2), it needs three critical files to be present[13]:

- `myproject.gpr` (henceforth the project file): this is a GPR project file that must be able to build the project. If different scenarios are important for the project, library authors can take into account that `alr` will set an "ALIRE=True" external variable during compilation. This way, the project file can define a specific scenario for Alire builds.
  In practice, GNAT projects typically already have one or several project files, so this is not a special requirement, but for the naming, that must coincide with the project name. This enables users of the project to simply use `with "myproject.gpr"` in their own project files.
- `myproject_alr.ads` (henceforth the metadata file): this file is used by `alr` as a telltale that it is being run inside a project folder. It must contain the project name and its milestone dependencies, as already shown in 3.2. It is initially generated by `alr init`, or could be hand-crafted if needed. It can also be regenerated on demand.
- `myproject_alr.gpr` (henceforth the environment file): this file is generated by `alr` to set up the environment paths required to find any projects the current project depends on. It can also be used to work in the GNAT GPS IDE.

Of these three files, the only one that is entirely the responsibility of the project author (or maintainer) is the `myproject.gpr` one. Its contents are arbitrary, as long as they succeed in building the library or executable. At a minimum, they must point the compiler to the source files of the project. On the other extreme, `myproject_alr.gpr` is regenerated by `alr` whenever necessary to properly configure the building environment (namely, whenever dependencies change or the file is not found). `myproject_alr.ads` lies in the middle, since it is initially generated by `alr` but it must by tailored by the developer to their needs to indicate their dependencies.

Finally, note that for the inclusion of a project into the Alire catalog, only the project file is needed, since the contents of the metadata file will appear in the Alire index itself (see Listing 4.1), and the environment file is regenerated from that information.

4.2 Self-compilation of `alr` and project dependencies

A tool such this one is expected to have an up-to-date catalog, and also that the tool itself is up-to-date. The catalog update could be achieved in several fashions: parsing text files that contain some specific format, or loading a binary

[13] In these descriptions, "myproject" is a placeholder for an actual project name.

Listing 4.1. An example of library in the `alr` index with one dependency on another library.

```
with Alire.Index.Libhello;

package Alire.Index.Hello is

   Name : constant Project_Name := "hello";
   Repo : constant URL          := "https://bitbucket.org/alire/hello.git";

   V_1_0_1  : constant Release :=
                Register_Git
                   (Project    => Name,
                    Version    => V ("1.0.1"),
                    Hosting    => Repo,
                    Commit     => "65725c20778875eef12b61a01b437120932965f3",
                    Depends_On => At_Least_Within_Major (Libhello.V_1_0_0));

   --  Older release might follow

end Alire.Index.Hello;
```

database, for example. However, maintaining the tool up-to-date will involve compiling it from updated sources and replace the current executable. Also, incorporating the dependencies of a working project (parsing the `_alr.ads` file) would need either a custom parser or compilation and processing with ASIS [4].

As an alternative, `alr` solves all these necessities in a single and perhaps uncommon way: whenever the need is detected, it recompiles itself incorporating the metadata file into the build (when one is not available, a default one is used). Also, whenever a rebuild is triggered, an indexing file is generated that depends on all Alire catalog files. This way all needed information is incorporated into `alr` without the need to parse any external files, since the compiler does it itself.

After self-compiling, `alr` spawns a new instance of itself that is able to detect that it is up to date. This is done by checking the following points (also graphically depicted in Fig. 3):

1. That the running executable is the one in a canonical location hard-coded into `alr`.
2. That there is no metadata file within reach[14], hence no need to know the dependencies of a particular project, or:
3. If there is a metadata file, its hash is compared to the one computed prior to self-compilation and inclusion of such metadata file into the build, hence ensuring the `alr` executable contains up-to-date dependencies of the current project.

[14] The metadata file is looked for in the root folder and in its immediate children

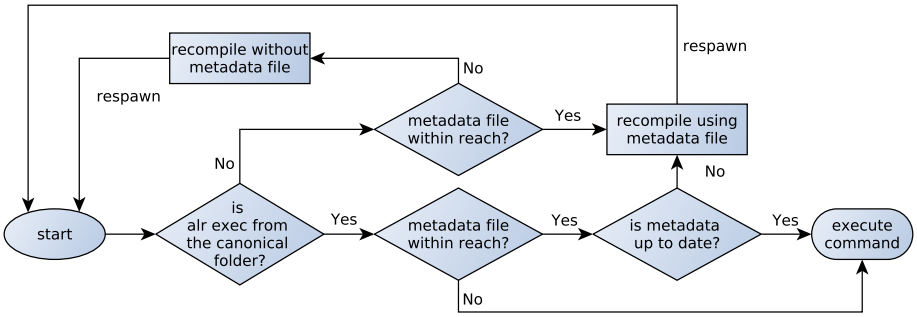

Fig. 3. Checks on self-consistency performed during `alr` start-up.

Once `alr` considers itself updated and with all needed information, it can apply its dependency solver. For now, it is a straightforward deep-first, newest-release-first search on the dependency tree rooted at the working project, that will return the first valid combination of dependencies found.

4.3 Final example

The creation of new projects from templates or downloading of releases do not really merit any special discussion, since they do not pose particular technological challenges. However, inspecting the filesystem after the issuing of an `alr get --compile hello` command will allow to bring into focus everything that has been reported up to this point. This command simultaneously fetches a project and its dependencies, generates the needed files and builds the whole configuration. The project itself is a plain "Hello, world!" example artificially split into having to depend on a library (libhello) that performs the actual output to the terminal.

Fig. 4 shows the relevant parts of a filesystem in which such a command were issued in the user's home folder. From top to bottom, the following relevant folders and files can be located:

- `$XDG_CONFIG_HOME/alire/` is the canonical Alire configuration data path, which currently is used to check out up-to-date versions of the tool. Within, the `alire/index/` folder contains the catalog files, and the most recently built `alr` executable is found in `alr/bin/alr`.
- `$XDG_CACHE_HOME/alire/` contains files that could be deleted without critical consequences, as they are either downloaded or generated when needed. Inside, there is the `projects/` folder in which dependencies are downloaded (in this case a particular release of the libhello dependency), and the `sessions/` folder that is used to store generated files for a rebuild of `alr`.

Fig. 4. Filesystem of a system using `alr`.

In this example, since `hello` is the current project, we find the following files included in the build:

- `alr-index.ads`, generated to include all catalog files that are available in `alire/index/`.
- `alr-session.ads`, that contains the hash of the current metadata file (`hello_alr.ads` in this example) and also depends on it.
- `hello_alr.ads`, which contains the project dependencies (that is, project libhello).

– Finally, the folder containing the release of project hello. The project `alr` is asked to get is not placed in the cache, but in the current folder, so the user can have easy access to it and its build results.

To conclude, proper library versions during a build are selected in the generated environment file, as Listing 4.2 shows for this example.

Listing 4.2. The environment file is a GPR aggregate project file.

```
aggregate project Hello_Alr is

   for Project_Files use ("hello.gpr"); -- Root project being compiled

   for Project_Path use (".",
      "/home/mosteo/.cache/alire/projects/libhello_1.0.0_ce78e770");

   for External ("ALIRE") use "True"; -- This is an Alire build

end Hello_Alr;
```

4.4 Discussion

At the time of this writing, `alr` offers the basic commands to demonstrate that distribution of Ada libraries is feasible using exclusively Ada tooling and free, public repositories; at least for libraries simple enough to be entirely built from GNAT project files. More advanced features could be built on top of this basic functionality that are clearly desirable: automated build tests and automatic generation of platform packages are but two examples; also, including licensing information and restrictions based on operating system or compiler version could be considered to limit which libraries can be depended upon.

Given the presented design, compilation times of `alr` itself could be a point of contention since such compilations happen every time the metadata file changes (i.e., whenever dependencies are added or removed). To assess that point, experimental runs were conducted for different catalog sizes. However, since only a few files are recompiled every time (session and metadata files, and one body that uses them in `alr`), the impact is mostly limited to the time it takes to redo the binding and linking. Times measured with a middle-range[15] computer are shown in table 1. Although not negligible, there is wiggle room until the issue becomes a pressing bottleneck.

	Indexed files		
Releases per file	100	1000	10000
1	1.82	3.73	34.09
10	1.94	4.52	44.83

Table 1. Average times (in seconds) for 100 `alr` recompilations after metadata changes, for different number of files in the catalog and releases per file. These times do not include an initial full compilation, which takes longer. Compiler version was GNAT GPL 2017 using `-j0` switch.

5 Conclusions

This work presented an experimental Ada tool, its underlying design and required infrastructure that facilitates easy reuse of third-party Ada projects. This is achieved by indexing and tagging with a semantic version their code releases in public repositories, which in turn enables the possibility of dependency resolution and easy upgrades. The whole setup requires only a recent GNAT Ada compiler and enables effortless downloading and compilation of indexed projects.

The design is based around a metadata file which is itself written in Ada and incorporated into the tool by recompilation triggered by the tool itself. This process allows users and developers of the tool alike to remain within the realm of pure Ada code.

Alire is available under an open source license to interested parties.

[15] Intel® Core™ i3-2015 (four execution threads), 16GB RAM, SSHD disk.

Aknowledgements

This work has been supported by projects ROBOCHALLENGE (DPI2016-76676-R-AEI/FEDER-UE) and ESTER (CUD2017-00). The authors thank the regulars at `comp.lang.ada` for insightful discussions.

References

1. Al-Kofahi, J., Nguyen, T.N., Kästner, C.: Escaping AutoHell: a vision for automated analysis and migration of autotools build systems. In: Proceedings of the 4th International Workshop on Release Engineering. pp. 12–15. ACM (2016)
2. Brenta, L., Leake, S.: Debian policy for Ada, `https://people.debian.org/~lbrenta/debian-ada-policy.html`
3. Clearinghouse, A.I.: Ada free tools and libraries, `http://www.adaic.org/ada-resources/tools-libraries/`
4. Colket, C.: Ada semantic interface specification (ASIS). ACM SIGAda Ada Letters (4), 50–63 (1995)
5. Dolstra, E., Löh, A.: NixOS: A purely functional linux distribution. ACM Sigplan Notices 43(9), 367–378 (2008)
6. Eisenbach, S., Jurisic, V., Sadler, C.: Managing the evolution of .NET programs. In: International Conference on Formal Methods for Open Object-Based Distributed Systems. pp. 185–198. Springer (2003)
7. Hamilton, D., Pape, P.: 20 years after the mandate. CrossTalk p. 15 (2017)
8. Matsakis, N.D., Klock II, F.S.: The rust language. ACM SIGAda Ada Letters 34(3), 103–104 (2014)
9. Newton, I., Turnbull, H.W., Scott, J.F.: The correspondence of Isaac Newton / edited by H.W. Turnbull. Published for the Royal Society at the University Press Cambridge (1959)
10. Odersky, M., Moors, A.: Fighting bit rot with types (experience report: Scala collections). In: LIPIcs-Leibniz International Proceedings in Informatics. vol. 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2009)
11. Peterson, C.: How I coined the term 'open source', `https://opensource.com/article/18/2/coining-term-open-source-software`
12. Raemaekers, S., Van Deursen, A., Visser, J.: Semantic versioning versus breaking changes: A study of the maven repository. In: Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on. pp. 215–224. IEEE (2014)
13. Reitz, K., Schlusser, T.: The Hitchhiker's Guide to Python: Best Practices for Development. O'Reilly Media, Inc. (2016)
14. Schonberg, E., Banner, B.: The GNAT project: a GNU-Ada 9X compiler. In: Proceedings of the conference on TRI-Ada'94. pp. 48–57. ACM (1994)
15. Stefik, A., Hanenberg, S.: The programming language wars: Questions and responsibilities for the programming language community. In: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. pp. 283–299. ACM (2014)
16. Taft, S.T., Duff, R.A., Brukardt, R.L., Ploedereder, E., Leroy, P., Schonberg, E.: Ada 2012 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652/2012 (E), vol. 8339. Springer (2014)
17. Thiruvathukal, G.K.: Gentoo linux: the next generation of linux. Computing in science & engineering 6(5), 66–74 (2004)
18. Tuong, F., Le Fessant, F., Gazagnaire, T.: OPAM: an OCaml package manager. In: SIGPLAN OCaml Users and Developers Workshop (2012)