Where `impact` got *Go*ing

Michael Tiller[1] Dietmar Winkler[2]

[1]Xogeny Inc., USA, michael.tiller@xogeny.com
[2]Telemark University College, Norway, dietmar.winkler@hit.no

Abstract

This paper discusses the `impact` package manager. The primary goal of this project is to support the development of a healthy eco-system around Modelica. For many other languages, the existence of an easy to use package manager has made it easier for people to explore and adopt those languages. We seek to bring that same kind of capability to the Modelica community by incorporating useful features from other package managers like `bower`, `npm`, *etc.*

This paper is an update on the status of the `impact` package manager which was discussed previously in (Tiller and Winkler 2014). This latest version of `impact` involves a complete rewrite that incorporates a more advanced dependency resolution algorithm. That dependency resolution will be discussed in depth along with many of the subtle issues that arose during the development of this latest version of `impact`. Along with a superior dependency resolution scheme, the new version of `impact` is much easier to install and use. Furthermore, it includes many useful new features as well.
Keywords: Modelica, package management, GitHub, dependency resolution, golang

1 Introduction

1.1 Motivation

The motivation behind the `impact` project is to support two critical aspects of library development. The first is to make it very easy for library developers to *publish* their work. The second is, at the same time, to make it easy for library consumers to both *find* and *install* published libraries.

We also feel it is important to reinforce best practices with respect to model development. For this reason, we have made version control an integral part of our solution. Rather than putting users in a position to have to figure out how to make `impact` work with a version control system, we've build `impact` around the version control system. Not only do users not have to find a way to make these technologies work together, `impact` actually nudges those not using version control toward solutions that incorporate version control. In this way, we hope to demonstrate to people the advantages of both `impact` and version control and establish both as "best practices" for model development.

By creating a tool that makes it easy to both publish and install libraries, we feel we are creating a critical piece of the foundation necessary to establish a **healthy ecosystem** for model development.

1.2 History

Earlier, we mentioned that `impact` has been completely rewritten. In fact, the very first version of `impact` was just a single Python script for indexing and installing Modelica code (Tiller 2013). It eventually evolved into a multi-file package that could be installed using the Python package management tools.

2 Requirements

After building the original Python version, we gave some thought to what worked well and what didn't work well. One issue we ran into almost immediately was the complexity of installing the Python version of `impact`. Python is unusual in that it has two package managers, `easy_install` and `pip`. It comes with `easy_install`, but `pip` is the more capable package manager. So in order for someone to install `impact`, they first needed to install Python, then install `pip` and then install `impact`. This was far too complicated. So we wanted to come up with a way for people to install `impact` **as a simple executable** without any run-time or prerequisites.

Another issue we ran into with the Python version was the fact that there are two different and incompatible versions of Python being used today (*i.e.,* 2.x and 3.x). Trying to support both was an unnecessarily inefficient use of resources. We also had some difficulties in the Python version with support for SSL under Windows (StackOverflow 2010). Because we were doing lots of "crawling" (more on this shortly), we needed a platform that provided **solid HTTP client support**. For these reasons, we felt we needed to move away from Python altogether.

Although most Modelica users run their development tools and simulations under Windows, there are several tools that support OSX and Linux as well as Windows. So as to not neglect users of those tools and to support more cross-platform options, we also wanted to be able to **compile `impact` for all three major platforms**.

Furthermore, we wanted to provide a simple executable for all platforms without having to have actual development machines for each of these different platforms. For this reason, **cross compilation** between different platforms was an important consideration as well.

Of course, we also wanted to have **good performance**. For most package management related functions, the speed of the internet connection is probably the biggest limiting factor. So CPU performance wasn't that high on the list. But, as we shall discuss shortly, the computational complexity of the dependency resolution algorithm we implemented could lead to some computationally intensive calculations for complex systems of dependencies.

For these reasons, we ultimately rewrote `impact` in Go (Go-Developers 2014). Go is a relatively new language from Google that stresses simplicity in language semantics but, at the same time, provides a fairly complete standard library. You can think of Go as being quite similar to C with support for extremely simple object-oriented functionality, automatic garbage collection and language level support for CSP-based concurrency. With Go, we were able to satisfy all the requirements above.

3 Version Numbering

Before we dive into all the details associated with crawling, indexing, resolving and installing, it is useful to take a moment to briefly discuss versioning. Modelica supports the notion of versions through the use of the `version` and `uses` annotations. These two annotations allow libraries to explicitly state what version they are and what versions of other libraries they use, respectively.

But there is one complication to the way Modelica deals with versions. In Modelica, a version is simply a string. This by itself isn't a problem. But it becomes a problem, as we will discuss in greater detail shortly, when you need to understand relationships between versions. In particular, there are two important things we would like to determine when dealing with version numbers. The first is an unambiguous ordering of versions. In other words, which, of any two versions, is the "latest" version? The second is whether a newer version of a library is "backwards compatible" with a previous version. These are essential questions when trying to resolve dependencies and the current string based approach to versions in Modelica is not semantically rich enough to help us answer either of these.

This issue is not unique to the Modelica world. These same questions have been asked for a very long time and various approaches have been invented to deal with answering these questions. One recent and widely used approach is to employ what is called **semantic versioning** (Preston-Werner 2014). Semantic versioning is pretty much what it sounds like, an approach to defining version numbers where the version numbers have very explicit meanings associated with them.

A very simple summary of semantic versioning would be that **all** versions have exactly three numerical components, a major version number, a minor version number and a patch. A semantic version must have all of these numbers and they must be `.`-separated. For this reason, the following versions are not legal semantic version numbers: `1`, `1a`, `1.0`, `1.0-beta5`, `4.0.2.4096`. Each of the three numbers in a semantic version means something. If you make a non-backward compatible change, you must increment the major version. If you make a backward compatible version, you must increment the minor version. If you make a change that should be completely compatible with the previous version (*e.g.,* doesn't add any new capability), you increment only the patch version.

There are additional provisions in semantic versioning to handle pre-release versions as well as build annotations. We will not discuss those semantics here, but they are incorporated into our implementation's treatment of version numbers.

Our use of semantic versioning is aligned with our goal of strongly encouraging best practices. It is important to point out that the use of semantic versions is completely legal in Modelica. In other words, Modelica allows a wider range of interpretations of version numbers. By using semantic versions, we narrow these interpretations but we feel that this narrowing is much better for the developer since it also provides meaning to the version numbers assigned to a library.

However, because Modelica libraries are free to use nearly any string as a version number, we need to find a way to "bridge the gap" between past usage and the usage we are encouraging moving forward. Although internally `impact` understands **only** semantic versions, it is still able to work with nearly all existing Modelica libraries. This is achieved through a process of "normalizing" existing versions. When `impact` comes across versions that are not legal semantic versions, it attempts to create an equivalent semantic version representation. For example, a library with a version string of `1.0` would be represented by the semantic version `1.0.0`.

For this normalization to work, it is important to make sure that the normalization is performed *both* on the version number associated with a library and on the version numbers of the libraries used. In other words, it must be applied consistently to both the `version` and `uses` annotations.

DOI 10.3384/ecp15118725

4 Indexing

As mentioned previously, there are two main functions that `impact` performs. The first is making it easy for library developers to publish their libraries and the other is making it easy for consumers to find and install those same libraries. Where these two needs meet is the library index. The index is **built** by collecting information about published libraries. The same index is **used** by consumers searching for information about available libraries.

Building the index involves crawling through repositories and extracting information about libraries that those repositories contain. In the following section we will discuss this crawling process in detail and describe the information that is collected and published in the resulting index.

4.1 Sources

Currently, `impact` only supports crawling GitHub (GitHub 2014) repositories. It does this by using the GitHub API (GitHub-Developers 2014) to search through repositories associated with particular users and to look for Modelica libraries stored in those repositories. We will shortly discuss exactly how it identifies Modelica libraries. But before we cover those details it is first necessary to understand which *versions* of the repository it looks into.

Each change in a Git repository involves a *commit*. That commit affects the contents of one or more files in the repository. During development, there are frequent commits. To identify specific versions of the repository, a tag can be associated with that version. Each tag in the repository history that starts with a `v` and is followed by a semantic version number is analyzed by `impact`.

4.2 Repository Structure

For each version of a repository tagged with a semantic version number, `impact` inspects the contents of that version of the repository looking for Modelica libraries. There are effectively two ways that `impact` finds Modelica libraries in a repository. The first is to check for libraries in "obvious" places that conform to some common conventions. For cases where such conventions are insufficient, `impact` looks for a file named `impact.json` to explicitly provide information about the repository.

4.2.1 Conventions

With respect to `impact`, the following is a list of "obvious" places that `impact` checks for the presence of Modelica libraries:

- **`./package.mo`** The entire repository is treated as a Modelica package.
- **`./<dirname>/package.mo`** or **`./<dirname> <ver>/package.mo`** The directory `<dirname>` is presumed to be a Modelica package.
- **`./<filename>.mo`** or **`./<filename> <ver>.mo`** The file `./<filename>.mo` is a file containing a Modelica library.

In all cases, the name of the library is determined by parsing the actual Modelica `package` definition and is not related to the name of the repository. As can be seen from these conventions, only files and directories that exist at the root level are checked for Modelica content.

4.2.2 `impact.json`

For various reasons, library developers may not wish to conform to the repository structure patterns discussed previously. Furthermore, there may be additional information they wish to include about their libraries. For this reason, a library developer can include an `impact.json` file in the root of the repository directory that provides additional information about the contents of the repository. For example, a repository may contain two or more Modelica libraries in subdirectories. The `impact.json` file allows information about the storage location of each library in the repository to be provided by the library developer. Furthermore, the author may wish to include contact information beyond what can be extracted from information about the repository and its owner. These are just a few use cases for why an `impact.json` file might be useful for library developers. A complete schema for the `impact.json` file can be found later in Section 4.4.2.

4.3 Handling Forks

The Modelica specification implicitly assumes that each library is uniquely identified by its name. This name is used in both the `version` and `uses` annotations as well as any references in Modelica code (e.g., `Modelica` in `Modelica.SIunits`). This assumption works well when discussing libraries currently loaded into a given tool. But when you expand the scope of your "namespace" to include all libraries available from multiple sources, the chance for overlap becomes possible and must be dealt with.

Previously, we mentioned the importance of supporting best practices in model development and the specific need to accommodate version control as part of that process. Up until now, we have leveraged version control to make the process of indexing and collection libraries easier. However, version control does introduce one complexity as well. That complexity is how to deal with *forks*.

Forks are common in open source projects and typically occur when there are multiple perspectives on how

development should progress on a given project. In some cases, rather than reconciling these different perspectives, developers decide to proceed in different directions. When this happens, the project becomes "forked" and there are then (at least) two **different** libraries being developed in parallel. Each of these libraries may share a common name and perhaps even the same version numbers but still be fundamentally different libraries.

A fork can arise for another, more positive, reason. When someone improves a library they may not have permission to simply fold their improvement back into the original library. On GitHub in particular, it is extremely common for a library to be forked simply to enable a third-party to make an improvement. The author of the improvement then sends what is called a *pull request* to the library author asking them to incorporate the improvement. In such a workflow, the fork is simply a temporary measure (akin to a branch) to support concurrent development. Once the pull request is accepted, the fork can be removed entirely.

Regardless of why the fork occurs, it is important that `impact` accommodates cases where forking occurs. This is because forking is a very common occurrence in a healthy eco-system. It indicates progress and interest and we should not do anything to stifle either of these. The issue with forking is that the same name might be used by multiple libraries. In such cases, we need a better way to uniquely identify libraries.

For this reason, `impact` records **not only the library name, but also a URI associated with each library**. In this way, the URI serves as a completely unambiguous way of identifying different libraries. While two forks may have the same name, they will never have the same URI.

4.4 Schema

We've mentioned the kinds of information `impact` collects while indexing as well as the kind of information that might be provided by library developers (via `impact.json` files). In this section, we will provide a complete description of information used by `impact`.

4.4.1 `impact_index.json`

As part of the indexing process, `impact` produces an index file named `impact_index.json`. This is a JSON encoded representation of all the libraries found during indexing. The root of an `impact_index.json` file contains only two elements:

`version` A string indicating what version of impact generated the index. The string is, of course, a semantic version.

`libraries` The libraries field is an array. Each element in the array describes a library that was found. **The order of the elements is significant**. Libraries that occur earlier in the list take precedence over libraries that appear later. This is important in cases where libraries have the same name.

For each library in the `libraries` array, the following information may be present:

`name` The name of the library (as used in Modelica)

`description` A textual description of the library

`stars` A way of "rating" libraries. In the case of GitHub, this is the number of times the repository has been starred. But for other types of sources, other metrics can be used.

`uri` A URI to uniquely identify the given library (when it shares a common `name` with another library)

`owner_uri` A URI to uniquely identify the owner of the library

`email` The email address of the owner/maintainer of the library

`homepage` The URL for the library's homepage

`repository` The URI for the library's source code repository

`format` The format of the library's source code repository (e.g., Git, Mercurial, Subversion)

`versions` This is an object that maps a semantic version (in the form of a string) to details associated with that specific version

The details associated with each version are as follows:

`version` A string representation of the semantic version (*i.e.,* one that is identical to the key).

`tarball_url` A URL that points to an archive of the repository in `tar` format.

`zipball_url` A URL that points to an archive of the repository in `zip` format.

`path` The location of the library within the repository.

`isfile` Whether the Modelica library is stored as a file (`true`) or as a directory (`false`)

`sha` This is a hash associated with this particular version. This is currently recorded by `impact` during indexing but not used. Such a hash could be useful for caching repository information locally.

`dependencies` This is an array listing the dependencies that this particular version has on other Modelica libraries. Each element in this array is an object with one field, `name`, giving the name of the required library and another field, `version`, which is the **semantic version** of that library represented as a string (see previous discussion on normalization in 3).

4.4.2 `impact.json`

As mentioned previously in Section 4.2.2, each directory can include a file named `impact.json` that provides explicit information about Modelica libraries contained

DOI 10.3384/ecp15118725

in that repository. The root of the `impact.json` file contains the following information:

`owner_uri` A link to information about the libraries owner

`email` The email address of the owner or maintainer

`alias` An object that whose keys are the names of libraries and whose associated values are the unique URIs of those libraries. This information can, therefore, be used to disambiguate between dependencies where there may be multiple libraries with that name.

`libraries` This is an array where each element is an object that contains information about a library present in the repository.

For each library listed in the `libraries` field, the following information may be provided:

`name` The name of the library

`path` The path to the library

`isfile` Whether the entity pointed to by `path` is a Modelica library stored as a file (`true`) or as a directory (`false`).

`issues_url` A link pointing to the issue tracker for this library

`dependencies` An explicit list of dependencies for this library (if not provided, the list will be based on the `uses` annotations found in the package definition).

Each dependency in the list should be an object that provides the following information:

`name` Name of the required library

`uri` Unique URI of the required library

`version` **Semantic version** number of the required library (represented as a string)

5 Installation

The previous section focused on how `impact` collects information about available libraries. The main application for this information is to support installation of those libraries. In this section, we'll discuss the installation side of using `impact`.

5.1 Dependency Resolution

5.1.1 Background

To understand the abstract problem behind the concept of a dependency, we refer to the formal study undertaken in (Boender 2011). There, a repository is defined as a triple (R, D, C) of a set of packages R, a dependency function $D : R \to \mathscr{P}(\mathscr{P}(R))$, and a *conflict relation* $C \subseteq R \times R$. At that level, version numbers have been abstracted to (distinguishable) packages: Every version yields a distinctive package $p \in P$.

The dependency function D maps a package p to sets of sets of packages $d \in D(p)$, where each set represents a way to provide one required feature of p. In other words: If for each $d \in D(p)$ at least one package in d is installed, it is possible to use p.

Currently, there is no way to express *conflicts* directly in a Modelica package. However, due to the existence of external libraries (which could conflict in arbitrary ways), it is likely that such a need will arise in the future. Additionally, current Modelica makes it impossible to refer to two different versions of a library from the same model. Hence, we consider different versions of the same package conflicting.

The dependency resolution of `impact` fits into Boender's model. Therefore, the conclusions drawn in (Boender 2011) can be applied to `impact` as well:

The set of packages `impact` installs for a given project needs to fulfill two properties, Boender calls *abundance* and *peace*. Informally, abundance captures the requirement that all dependencies be met while peace avoids packages that are in conflict with each other. A set of packages that is peaceful and abundant is called *healthy* and a package p is called *installable* w.r.t. a given repository if and only if there exists a healthy set I in said repository such that $p \in I$.

The problem of finding such an installable set is however a hard one. In fact, Boender proves by a simple isomorphism between the boolean satisfiability problem and the dependency resolution that finding such a set is NP-hard. Fortunately, for the current typical problem size, this isn't really an issue.

5.1.2 Resolution Algorithm

The indexing process collects quite a bit of information about available libraries. Most of the complexity in implementing the installation functionality in `impact` is in figuring out **what** to install. And most of that complexity is in finding a set of versions for the required libraries that satisfy all the dependency relations. This process is called dependency resolution.

The resolution algorithm starts with a list of libraries that the user wants to install. In some cases, this may be a single library but, in general, the list can be of any length. For each library in the list, the user may specify a particular version of the library they wish to install, but this isn't mandatory. One important point here is that we refer to this as a *list*, not a set. Order is significant here. The libraries that appear first are given a higher priority than those that appear later.

Let's explain why this priority is important. Consider a user who wishes to install libraries `A` and `B`. If the user has not explicitly specified what version of each library they are interested in, `impact` assumes the user wants

the latest version, if possible. But what if the latest version of *both* cannot be used? To understand this case, consider the following constraints:

`A:1.0.0` uses `B:2.0.0`

`A:2.0.0` uses `B:1.0.0`

where `A:1.0.0` means version `1.0.0` of library `A`. This example is admittedly contrived, but the underlying issue is not. We can see here that if we want the latest version of `A`, we cannot also use the latest version of `B` (and *vice versa*) while still honoring the constraints above. The ordering of the libraries determines how we "break the tie" here. Since `A` appears first, we assume it is more important to have the latest version of `A` than to have the latest version of `B`.

Let's take this extremely simple example to outline how the resolution algorithm would function in this case. In later sections, we'll introduce additional complexities that must be dealt with.

If a user asks for libraries `A` and `B` to be installed, the question that the dependency algorithm has to answer is **which versions** do we use. Assuming that each library has a version `1.0.0` and `2.0.0`, then each "variable" in this problem has two possible values. The following table essentially summarizes the possibilities:

Version of `A`	Version of `B`
`1.0.0`	`1.0.0`
`1.0.0`	`2.0.0`
`2.0.0`	`1.0.0`
`2.0.0`	`2.0.0`

This is a simple enumeration of the possibilities. But *remember*, we assume the user wants the most recent version *and* we assume `A` is more important than `B`. Semantic versioning provides us with a basis for determining which version is more recent. Given these we reorder these combinations so that the most desirable combinations appear first and the least desirable appear last:

Version of `A`	Version of `B`
`2.0.0`	`2.0.0`
`2.0.0`	`1.0.0`
`1.0.0`	`2.0.0`
`1.0.0`	`1.0.0`

Now we see the `impact` of the dependency constraints. Specifically, the first (most desirable) combination in this table does not satisfy the dependency constraints (*i.e.,* `A:2.0.0` does not work with `B:2.0.0`). If we eliminate rows that violate our dependency constraints, we are left with:

Version of `A`	Version of `B`
`2.0.0`	`1.0.0`
`1.0.0`	`2.0.0`

In summary, we order the combinations by their desirability (considering both the relative priority of the libraries and their version numbers) and then we eliminate combinations that don't satisfy our dependency constraints.

This gives an overview of how the algorithm works conceptually. But, as you may have guessed, the problem is not quite this simple. Consider now a slightly more complex case with the following dependencies:

1	`A:3.0.0`	uses	`B:1.2.0`
2	`A:3.0.0`	uses	`C:1.1.0`
3	`B:1.2.0`	uses	`C:1.2.0`
4	`A:2.0.0`	uses	`B:1.1.0`
5	`A:2.0.0`	uses	`C:1.0.0`
6	`B:1.1.0`	uses	`C:1.1.0`
7	`A:1.0.0`	uses	`B:1.0.0`
8	`A:1.0.0`	uses	`C:1.0.0`
9	`B:1.0.0`	uses	`C:1.0.0`

Now we have three variables we need to solve for, `A`, `B` and `C`. For each variable, we have three possible values. As we've already described, newer versions are preferred over older versions while searching. This means that the first combination we will consider will be...

`A:3.0.0`, `B:1.2.0` and `C:1.2.0`

...and the last combination we will consider will be...

`A:1.0.0`, `B:1.2.0` and `C:1.2.0`

There are several interesting things to notice about this case. First, although the problem is not particularly large (3 libraries with 3 versions each), the number of combinations to check is significant (*i.e.,* $3 \cdot 3 \cdot 3 = 27$). Of these 27 combinations, only the last one to be considered (*i.e.,* the least desirable) satisfies the dependency constraints. There is nothing we can really do about the fact that the oldest version of each of these libraries must be used (this is dictated by the dependencies themselves and has nothing to do with the algorithm). But the complication is that we must consider all of them (in this contrived case) before finding the one we want.

In reality, we would not actually enumerate all possibilities *a priori*. Instead, we would simply consider each "variable" one at a time and loop over all possible versions. If, at any point, we find a conflict with our constraints, we simply break out of the inner most loop. This is referred to as *backtracking*. In Modelica pseudo-code, the algorithm (for this specific case) might look like this:

```
for A in ["3.0.0", "2.0.0", "1.0.0"] loop
  for B in ["1.2.0", "1.1.0", "1.0.0"] loop
    if not are_compatible(A,B) then
       break;
    end if;
    for C in ["1.2.0", "1.1.0", "1.0.0"] loop
      if not are_compatible(B,C) then
        break;
      end if;
      if not are_compatible(A,C) then
        break;
      end if;
      //If we get here, we have a solution
    end for;
  end for;
end for;
```


DOI 10.3384/ecp15118725

Using this backtracking, we can more efficiently traverse the possibilities by eliminating lots of cases that we know are a dead end (especially in larger problems). Any search based on backtracking is vulnerable to poor performance under certain (typically pathological) conditions. We'll return to this point later when we talk about performance of our current implementation.

There is one last complication we must deal with when resolving dependencies. Consider the following simple set of dependencies:

1	`A:2.0.0`	uses	`B:1.2.0`
2	`A:2.0.0`	uses	`C:1.1.0`
3	`B:1.2.0`	uses	`C:1.2.0`
4	`A:1.0.0`	uses	`B:1.1.0` or `B:1.0.0`
	(*i.e.,* `A` can use `B:1.1.0` or `B:1.0.0`)		
5	`A:1.0.0`	uses	`D:1.1.0`
6	`B:1.0.0`	uses	`C:1.1.0`
7	`C:1.2.0`	uses	`D:1.0.0`
8	`B:1.1.0`	uses	`C:1.2.0`

We can also represent this set of dependencies graphically as shown in Figure 1. Graphically, we have a box to represent each library and that box contains the different versions available. These versions are connected by the constraints shown in the table above.

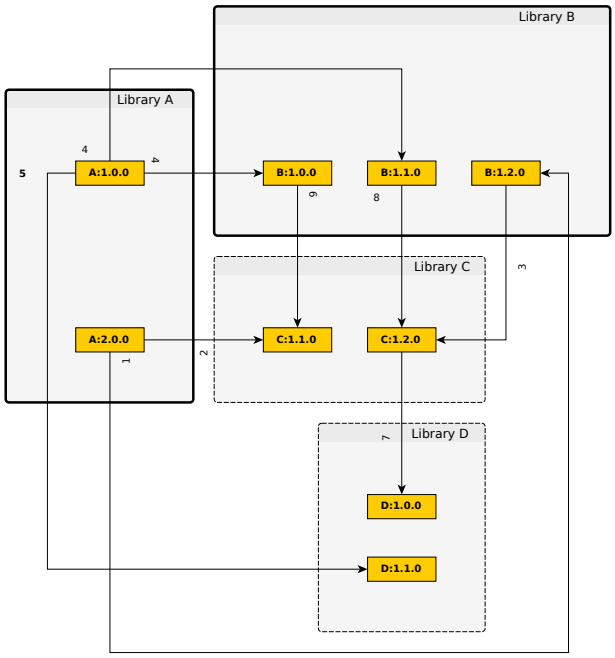

Figure 1. Graphical representation of package dependencies

Given these dependencies and the fact that the user wishes to install both `A` and `B`, what are the variables in our dependency resolution algorithm? Obviously, we must consider all the versions of both `A` and `B` (*i.e.,* we must pick a version from the box for library `A` and `B` in Figure 1). But what about `C` and `D`? It makes no sense to enumerate all combinations of versions for these four libraries because in many cases `D` isn't even required. Furthermore, what is their relatively priority (*i.e.,* if a choice is required, is it more important to have the latest version of `C` or `D`?)

When resolving dependencies, we only introduce new libraries when necessary (*i.e.,* if they are needed by our current choices of existing libraries) and their relative priority is determined by the relative priority of the library that introduced them.

To understand how the resolution works in this case, first consider the case of `A:2.0.0`. This version cannot be chosen. This is because `A:2.0.0` wants `C:1.0.0` while `B:1.2.0` wants `C:1.1.0`. So no choice for `C` is valid. Furthermore, we don't even consider `D` because it isn't required in any of these cases.

Now if we move to the case of `A:1.0.0`, things are more complicated. Now we **do** need to consider both `D` and `C`. However, note that because `A:1.0.0` depends directly on `D`, we consider `D` more important. This is important because when considering `A:1.0.0` we have two versions of `B` that are compatible[1] (*i.e.,* `B:1.1.0` and `B:1.0.0`). Given that we are considering `A:1.0.0` and we've already ruled out `B:1.2.0`, we are left with the following combinations:

Version of `B`	Version of `D`	Version of `C`
`1.1.0`	`1.1.0`	`1.2.0`
`1.1.0`	`1.1.0`	`1.1.0`
`1.1.0`	`1.0.0`	`1.2.0`
`1.1.0`	`1.0.0`	`1.1.0`
`1.0.0`	`1.1.0`	`1.2.0`
`1.0.0`	`1.1.0`	`1.1.0`
`1.0.0`	`1.0.0`	`1.2.0`
`1.0.0`	`1.0.0`	`1.1.0`

Notice the ordering of the columns? Since the user originally asked for both `A` and `B`, `B` comes first. But when it comes to `C` and `D`, having a more recent version of `D` is more important than having a more recent version of `C`.

As mentioned previously, we don't construct every combination. Furthermore, we don't always consider all libraries. The best way to understand how this search proceeds is to enumerate the partial combinations that our search generates and the point at which backtracking occurs. In such a case, we can think of the search as proceeding as follows:

[1] It is not possible to express this kind of "or" dependency currently in Modelica, but it is supported by `impact`. This capability exists in `impact` both to support anticipated future capabilities in Modelica (Tiller 2012) and/or to support cases we will discuss shortly that consider cases of compatibility implicit in semantic versions.

A:2.0.0	👍
A:2.0.0 & B:1.2.0	👍
A:2.0.0 & B:1.2.0 & C:1.2.0	✘→#2
A:2.0.0 & B:1.2.0 & C:1.1.0	✘→#3
A:2.0.0 & B:1.1.0	✘→#1
A:2.0.0 & B:1.0.0	✘→#1
A:1.0.0 & B:1.2.0	✘→#4
A:1.0.0 & B:1.1.0	👍
A:1.0.0 & B:1.1.0 & D:1.1.0	👍
A:1.0.0 & B:1.1.0 & D:1.1.0 & C:1.2.0	✘→#7
A:1.0.0 & B:1.1.0 & D:1.1.0 & C:1.1.0	✘→#8
A:1.0.0 & B:1.1.0 & D:1.0.0	✘→#5
A:1.0.0 & B:1.0.0 & D:1.1.0 & C:1.2.0	✘→#6
A:1.0.0 & B:1.0.0 & D:1.1.0 & C:1.1.0	✔

This elaboration of the search shows the role that the relative priority of libraries and versions has on the search order but also how a particular library is not even considered until a dependency on that library is introduced by choosing a particular version that depends on it.

It should be noted that there are a variety of other special cases we also deal with like self dependency and cyclic dependency. But these are constraints like any other and don't really impact the algorithm in any significant way.

The actual algorithm is implemented by the `findFirst` method on the `LibraryGraph` type found in the `github.com/xogeny/impact/graph` package. The inputs to this function are:

`mapped` Any existing decisions about specific versions of each library (initially empty)

`avail` The set of all (remaining) possible versions for each library (initially all versions of all libraries)

`rest` A list of libraries that are required based on existing decisions but for which no version choice has yet been made (initially the libraries the user wants installed in the order specified by the user)

The algorithm then proceeds as follows:

1. Is `rest` empty? If so, we are done and we have a solution (*i.e.,* `mapped`)
2. Consider the first library in `rest`
3. Loop over available versions (based on `avail`)
   (a) Add this choice to `mapped`
   (b) Find any new library dependencies resulting from this choice
   (c) If incompatible decisions have already been made about these new dependencies, backtrack
   (d) Update `avail` to include version of new dependencies that are compatible with our previous choices
   (e) If there are no possible versions for any library we depend on, backtrack
   (f) Return the result of calling this function again recursively using updated values for `mapped`, `avail` and `rest`.

5.1.3 Formulating Constraints

The default assumption is that dependencies will come from the `uses` annotation in Modelica. There is a proposal to extend the `uses` annotation to allow multiple compatible versions to be listed (vs. only a single compatible version today). As mentioned previously, such an `or` relationship is already supported by `impact`. So this change would not impact the resolution algorithm used by `impact`.

Although it hasn't yet been implemented, one proposed fallback mode for `impact` is to ignore the explicit dependencies contained in Modelica code and instead rely on the dependency relationships **implicit** in semantic versions. In other words, if a library `B` has two versions, `1.1.1` and `1.1.2`, and those versions strictly follow semantic versioning conventions, then we know that any library that depends on `B:1.1.1` must also be compatible with `B:1.1.2`. Such a fallback mode could be employed when `impact` is unable to find a solution using explicit constraints.

6 Go Implementation

We've created an implementation of `impact` using Go. This implementation includes different sub-packages for dealing with crawling repositories, resolving dependencies, parsing Modelica code and managing configuration settings. It also contains a sub-package for implementing the command-line tool and all of its sub-commands. This structure means that `impact` is not only a command-line tool, but also a Go library that can be embedded in other tools.

The Go implementation includes the following commands:

`search` Search library names and descriptions for search terms.

`install` Install one or more libraries and their dependencies.

`index` Build an index of repositories.

`version` Print out version and configuration information.

For each command, you can use the `-h` switch to find out more about the command and its options.

Earlier we described our requirements. The main reason we moved to Go from Python was Go's support for cross-compiling between all major platforms and the fact

DOI 10.3384/ecp15118725

that it generates a statically linked binary that doesn't depend on any runtime. The Go compiler includes a complete implementation of HTTP for both the client and server. In fact, the standard library for Go is fairly complete. At the moment, the only third party dependencies for `impact` are a Go implementation of the GitHub v3 API and an implementation of semantic versioning.

The performance of compiled Go code is quite good. In Section 5.1.2 we described how the algorithm we are using could, in a worst case scenario, search every potential combination before finding either a solution or failing. We constructed several test cases with n variables where each variable had 2 possible values. The result is that there will be 2^n possible combinations. These cases were contrived so that the least desirable combination was the only one that would satisfy the dependency constraints. We tested the time required for find a solution for different values of n and we got the following performance results:

n	Time (ms)
10	45
12	141
14	646
20	52,000

It is important to keep in mind that this is a **contrived** case to demonstrate the worst possible case for resolution. There may very well be other algorithmic approaches that will find identical solutions but search more efficiently. But given what we know about Modelica libraries and their dependencies, we found this performance more than sufficient for our application.

One last point worth making about the implementation of `impact` has to do with security. In order to generate an index from GitHub repositories, it is necessary to crawl repositories. In order to accomplish this, many API calls are required. GitHub will only allow a very limited number of "anonymous" API calls. This limit will be reached very quickly by `impact`. In order to increase the number of allowed API calls, GitHub requires an "API key" to be used. Such a key can be provided to `impact` but it cannot be provided via a command line option or a configuration file. This is to avoid this sensitive information being inadvertently recorded or exposed (*e.g.,* by committing it to a version control repository). Instead, such tokens must be provided as environment variables.

The `impact` source code is licensed under an MIT license and is hosted on GitHub. The GitHub repository (Xogeny 2015) includes a `LICENSE`, `README.md` and `CONTRIBUTING.md` which provide a detailed license, introductory documentation and instructions for contributors, respectively. We've linked the GitHub repository to a continuous integration service so that each commit triggers tests and emails out build status to the maintainers.

7 `impact` on library developers

What does all this mean for library developers wanting to make their library accessible via `impact`? Let us first have a look at the past "sins" that were restricting the development work on Modelica libraries.

7.1 Observations

1. We noticed that the `MODELICAPATH` concept is not properly understood by the users and often gets in their way. Therefore we should not rely on it but rather work with all of our files collected into a *working directory* (which should always part of the `MODELICAPATH` and made first priority for the look-up in the tool).

2. If we go away from having to collect all Modelica libraries in the `MODELICAPATH` then there is no longer a need to store the version number with the library folder name. I.e., simply "`<PackageName>`" is sufficient and no need for "`<PackageName> <Version>`".

3. Until now, we advised the lib developers to keep the current development version in a `master` branch and merge `master` into a `release` branch where the directory structure can be changed (e.g., into "`<PackageName> <Version>`" and any generated content can be added). Finally, developers should then place a tag on the release branch. This was done for the following reasons:

   - The link to the tag provided a (tar)zip file that contained the library with the "`<PackageName> <Version>`" format ready to be used with `MODELICAPATH`. However, we no longer need to rely on `MODELICAPATH` any more we don't need to add the `<Version>` identifier to the folder anymore.
   - If we would like to see what stage a certain release in `master` was at, then we needed to either inspect the `git` history (following backwards from the release tag) or use an additional tag (e.g., "`1.2.3-dev`") which is rather cumbersome and seems unnecessary.

   But since GitHub also supports new alternatives (see below) there is no longer a need for a specific `release` branch. That is to say, libray developers can still use it if they think it useful but they don't have to anymore.

7.2 Repository structure recommendations

There are new features/mechanisms made available both by GitHub and `impact`:

- GitHub's support for assets (GitHub-Blog 2013) allows us to upload additional files to tagged releases
- `impact` does not use the `MODELCAPATH` model but rather uses a "one working directory per project" approach where (one version) of all required libraries and their dependencies live in one (working) directory.

We recommend that library developers make the most out of the new features above and change the structure in which they organize their library repositories.

1. Get rid of the `release` branch as long as it was only for the sake of providing a download-able zip-file with a customized structure or providing additional generated files. Instead use the new GitHub Releases (GitHub-Blog 2013) which allows uploading of additional assets for download.
   - E.g., rather than adding HTML documentation to the `Resources` sub-folder and committing this to the `release` branch and then tagging it, tag the `master` branch and then generate a zip-file which contains that state and add the generated files to the tagged release. GitHub also provides some information on "Creating Releases" (GitHub-Help 2015a) and there exists, for example, the `aktau/github-release` tool (Hillegeer 2015) to help automating that process.
   - Another benefit of the release assets is that the GitHub API (GitHub-Developers 2014) allows you to get the download count for your releases (GitHub-Help 2015b). This was not possible for the simple taggged-zip-ball downloads.
2. Get rid of the `<PackageName> <Version>` formatted folder names. The version number does not belong in the `master` (i.e., development) branch anyway and the version annotation is contained in the `version` annotation which tools will happily display for you. When you install a package with `impact` it will strip that version number in any case.

7.3 Changes for the library listing

The listing of Modelica libraries on `https://modelica.org/libraries` is generated by parsing the GitHub API and creating a static HTML file that contains all information with links. Currently it is a standalone *Python* script but we are thinking of adding this functionality as a sub-module to `impact` itself.

Up to May 2015 the listing pointed directly to the (tar)zip-ball URL of the latest tag of a library. This worked fine if the library used the old `release` branch model where the "ready to install" version was placed. Clicking on that coloured version link resulted in a direct file download.

This has now been changed in such a way that if one clicks on the listed "Last Release" button one will get redirected to the "Releases" page of that project showing the last release. This has the advantage that one does not immediately download the (tar)zip ball but gets to see proper release notes first *and* is given a choice of what version of a release to download (e.g., pure source distribution of that tag, customized version with additional files, different platform dependent versions with pre-compiled binaries).

7.4 Which license is best for your library

The *Modelica Standard Library* (`Modelica`) (Modelica Association 2013) is licensed under the "Modelica License Version 2.0" (Modelica Association 2008). So in order to stay compatible with the `Modelica` library most user libraries chose the same license. This seemed like a natural choice. However, there is one problem which is not immediately apparent to most library developers. This is that the "Modelica License Version 2.0" contains the section "4. Designation of Derivative Works and of Modified Works" which says that:

"*. . . This means especially that the (root-level) name of a Modelica package under this license must be changed if the package is modified . . .* ".

This clause makes perfect sense for a main library like the `Modelica` library that is developed and maintained by a major group centrally and wants to protect its product name. But what does this mean for open-source projects that no longer are hosted centrally but rather decentralized on platforms like GitHub and GitLab but were contributions no longer are made by committing directly into **one** central repository? In the de-centralized case contributions are given by first "forking" (i.e., generating a copy of the original repository), modifying that fork and then sending the contribution back via a "pull-request" (i.e., offering the originating project to accept the changes made on the fork). The problem is that the very first step of "forking" the library generated a copy with the identical "(root-level) name" and at a different location. One could argue that this alone is already a violation of the terms of the "Modelica License Version 2.0".

So what should the library developer do? The simplest solution is to **not** use the "Modelica License Version 2.0" for libraries but rather go for standard licenses (Open Source Initiative 2015) that are more compatible with open source, community driven development (*e.g.,* MIT or BSD licenses). Interestingly, the old "Modelica License Version 1.1" is still suitable for user libraries since it does not contain the restrictions of having to change the package name.

So what about "copyleft style" licenses? The most

DOI 10.3384/ecp15118725

famous copyleft license is the GNU General Public License. People might think this would be a good choice for a license in order to protect parts of their library from being used inside proprietary libraries without any bugfixes and improvements being fed back to them as "upstream" developers. Unfortunately the GPL also forbids that any other non-GPL library (even the *Modelica Standard Library*) uses the GPL licensed library and is distributed that way. So what about the LGPL, this allows the usage and distribution *alongside* with other non-gpl libraries. The problem here is that it does not allow static linking. Something that typically happens when one creates a compiled version of a simulation model that uses different Modelica libraries. A typical example would be the generation of an FMU (Modelica Association 2015). A way out of this is the "Mozilla Public License" which is very much alike the LGPL but allows generated code to be statically linked together with non-GPL licensed code.

In conclusion, libraries should, if possible, avoid the "Modelica License Version 2.0" as this was primarily designed for the requirements of the *Modelica Standard Library*. Perhaps there will be a future revision that is adapted to current open-source development models. But until then, we suggest the use of standard licenses along the lines of BSD/MIT or MPL.

8 Future Development

8.1 Dependency Constraints

As already mentioned, there is currently no way to express conflicts between different packages. However, it is highly likely that such conflicting pairs will exist as more and more packages are published. For instance, two Modelica models might depend on different, specific versions of an external library that cannot be linked or loaded at the same time, an already published package might contain known bugs etc. Hence, `impact` could be extended by the means to express conflicts as well.

Boender introduces the notions of *strong dependencies* and *strong conflicts* to optimize the handling of very large repositories. This kind of optimization might not be necessary in the Modelica ecosystem right now, but could provide helpful performance enhancements in future versions of `impact`.

8.2 Crawling

At the moment, `impact` is only able to crawl GitHub repositories. There is nothing particularly special about GitHub and/or its APIs. The authors are confident that indices could be constructed for many different storage types. The most obvious next steps for crawling support would be to add support for GitLab and Bitbucket (Mercurial and Subversion) repositories. Pull requests to introduce such functionality are welcome.

On a related note, we anticipate there will be many use cases where `impact` could be useful for closed source projects that involve private repositories. We think this is an important use case and we hope to provide support for crawling such repositories. This would, for example, allow model developers at companies that have made a significant investment in building Modelica related models and libraries to use `impact` to search and install these proprietary libraries via their corporate intranet.

8.3 Project Details

We have already created a number of issues that require users to provide more explicit information about how they want impact to function on a per project basis. For example, when working with forked libraries (where the index contains multiple libraries with the same name), it is useful to use the URIs associated with each library in the index to disambiguate which particular library to use. Furthermore, there may be cases where the user is actually interested in doing development work on the dependencies as well. In such cases, those dependencies shouldn't simply be installed, they should be **checked out** from their repository to make modifying and recommitting easier.

For these and other project related features, we feel there is a need to introduce another file to provide such additional information that is project specific.

8.4 Web Based Search

Other package managers often provide a web site where users can search for a specific package through the web, read documentation, log issues and/or even download the packages. Because `impact` is organized into libraries (and not just a command line tool), we feel this kind of functionality could be added in the future.

8.5 Installers

Finally, when installing software, it is common for developers to distribute "installers" (*i.e.,* executables that, when run, unpack and install the software). Another potential extension of `impact` could be to generate such installers. In this case, we could once again leverage Go's static executable generation to build such installers from the index. Instead of installing the needed files locally, the installer could simply bundle them up and attach them to an installation program using one of the many Go extensions (Riemer 2015; Tebeka 2015) for concatenating static content onto executables or simply downloading some pre-specified libraries over the network.

9 Conclusion

In conclusion, `impact` leverages information already available in Modelica source code along with some common conventions in order to help users find and install Modelica libraries. It does this by crawling repositories and indexing their contents. An index of publicly available libraries created by `impact` is hosted on `modelica.org` for use by the `impact` command line tool.

If present, the `impact` command line tool is already used by OpenModelica to help find and install dependencies. By making the `impact` executables available across platforms and providing a version of the source code that can also be embedded as a library, we hope the Modelica community will benefit from having first class package management capabilities, just like other software eco-systems.

10 Acknowledgements

The authors would like to thank Christoph Höger of Technische Universität Berlin, Martin Sjölund of Linköping University, Francesco Casella of Politecnico di Milano and Peter Harman of ESi Group for their contributions to this project.

References

Boender, Jaap (2011). "A formal study of Free Software distributions". PhD thesis. Université Paris-Diderot-Paris VII.

Go-Developers (2014). *The Go Programming Language Specification*. URL: `http://golang.org/ref/spec`.

GitHub (2014). *Build software better, together.* URL: `https://github.com/`.

GitHub-Blog (2013). *Release Your Software*. URL: `https://github.com/blog/1547-release-your-software`.

GitHub-Developers (2014). *GitHub API v3*. URL: `http://developer.github.com/v3/`.

GitHub-Help (2015a). *Creating Releases*. URL: `https://help.github.com/articles/creating-releases/`.

– (2015b). *Getting the download count for your releases*. URL: `https://help.github.com/articles/getting-the-download-count-for-your-releases`.

Hillegeer, Nicolas (2015). *aktau/github-release*. URL: `https://github.com/aktau/github-release`.

Modelica Association (2008). *Modelica Licence Version 2.0*. URL: `https://modelica.org/licenses/ModelicaLicense2`.

– (2013). *Modelica - Free library from the Modelica Association*. URL: `https://github.com/modelica/Modelica`.

– (2015). *Functional Mock-up Interface*. URL: `https://fmi-standard.org`.

Open Source Initiative (2015). *Licenses*. URL: `http://opensource.org/licenses/`.

Preston-Werner, Tom (2014). *Semantic Versioning 2.0.0*. URL: `http://semver.org/`.

Riemer, Geert-Johan (2015). *go.rice*. URL: `https://github.com/GeertJohan/go.rice`.

StackOverflow (2010). *How to install Python ssl module on Windows?* URL: `http://stackoverflow.com/questions/2261866/how-to-install-python-ssl-module-on-windows`.

Tebeka, Miki (2015). *nrsc - Resource Compiler for Go*. URL: `https://bitbucket.org/tebeka/nrsc`.

Tiller, Michael (2012). *Modelica Change Proposal For Package Handling*. URL: `https://trac.modelica.org/Modelica/raw-attachment/ticket/573/Package-Proposal_asMCP.doc`.

– (2013). *Gist of first version of impact.py*. URL: `https://gist.github.com/xogeny/fac3ea9174e74275e7fe`.

Tiller, Michael and Dietmar Winkler (2014). "impact - A Modelica Package Manager". In: *Proceedings of the 10th International Modelica Conference, March 10-12, 2014, Lund, Sweden*. Ed. by Hubertus Tummescheit and Karl-Erik Årzén. Modelica Association. Linköping University Electronic Press, pp. 543–548. URL: `http://www.ep.liu.se/ecp/096/057/ecp14096057.pdf`.

Xogeny (2015). *impact code repository on GitHub*. URL: `https://github.com/xogeny/impact`.