Towards a Knowledge-Aware AI
A. Dimou et al. (Eds.)
© 2022 The Authors.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution License 4.0 (CC BY 4.0).
doi:10.3233/SSW220015

Plow: A Novel Approach to Interlinking Modular Ontologies Based on Software Package Management

Maximilian GOISSER, Daniel FIEBIG, Sebastian WOHLRAPP and Georg REHM
Field 33 GmbH, Berlin, Germany – firstname@field33.com

Abstract. Ontology development offers many challenges, with some of the most prominent being modularization and evolution of ontologies over time. Based on lessons learned from popular programming language package managers, we present a novel approach to package management of OWL ontologies. Most prominently we integrate a dependency resolution algorithm based on the popular SemVer versioning scheme with tooling support for dependency locking, which allows for decoupling publication and consumption of ontologies, reducing the need for coordination in ontology evolution. To complete our unified approach, we additionally provide an integrated registry, which serves as a domain-agnostic repository for ontologies (https://registry.field33.com).

Keywords. Semantics, Ontology Management, Ontology Modularization, Ontology Repositories, Knowledge Representation, OWL2, Package Management

1. Introduction and Motivation

More than a decade ago, the essay “Why Software is eating the world” [1] succinctly outlined how software is increasingly making its way into our everyday lives and how it is even being picked up in the value chains of primarily ‘analogue’ companies with business models rooted in the production of physical goods. Since 2011, this trend has continued with software growing in amount and complexity, which is why the field of software engineering has seen an increase in professionalization. With software being deeply entangled with business processes and, consequently, with business success, the requirements to reduce the risk of software projects have also grown. At the same time, threats to software production supply chains have increased, with malicious actors targeting them due to their high-value nature, using, for example, ransomware, as well as an increased reliance on third parties. To fulfill some of those requirements, new structured processes and tooling to support these processes have been established, primarily addressing the reuse of software by independent parties using programming language package management.

In the same time frame, industry adoption of ontologies and other semantic technologies has *not* seen the same level of growth, nor has it seen the same advances as other fields in software engineering in terms of robust tooling. Actual production use of ontologies is still largely relegated to a few domains in which they have demonstrated a track record to justify developing bespoke tooling, with little success in other areas. With

recent large-scale initiatives like Industrie 4.0 [2] or Gaia-X [3], interest in providing ontologies across and also as a basis for a wide variety of domains has been reignited.

When it comes to ontology engineering, the current model for package management is based on dependencies between ontology packages that are modelled solely using URIs, and package updates are handled via manual processes. This current state-of-the-art in ontology management is no longer adequate to fulfil the needs of modern software engineering workflows because the inclusion of ontologies in larger software projects can pose a significant risk to the project's success.

In this article, we present our approach, *Plow*, as a solution to bring well-established practices and tooling from programming language package management to ontology management while adapting it to the unique characteristics of knowledge representation through ontologies. By closing the gap between ontology engineering on the one hand and software engineering on the other, we are able to provide a robust foundation that projects relying on ontologies can build on and benefit from cross-fertilisation between the two fields moving forward.

The remainder of this article is structured as follows. First, Section 2 describes related work, especially with regard to modular ontologies, OWL, ontology repositories and programming language package management. Section 3 presents our overall approach, including the broad requirements we have. Section 4 describes, in detail, the implementation of Plow. Afterwards we explain the use of the tool (Section 5) and its limitations (Section 6). The article finishes with several conclusions and future work.

2. Related Work

2.1. Web Ontology Language (OWL)

The Web Ontology Language (OWL) [4] is one of the most commonly used languages for knowledge representation. Many ontology engineering tools support OWL and use the concept of an IRI-identified ontology as the fundamental unit of modularization. OWL2 [5] introduces a "version IRI" to identify a single version of an ontology.

To specify dependency relations between ontologies, the OWL specification establishes the concept of "imports" together with a number of annotation properties (e. g., `owl:imports`). Additional annotation properties allow for specifying the version of an ontology as well as specifying version order, compatibility and the incompatibility between different versions of an ontology. While these features provide a framework for expressing basic dependency relations, the specification intentionally leaves the semantics for the annotations under-specified, making them unfit as a source of truth for a fully-featured ontology management system on their own.

One area where the provided primitives are lacking is the ability to specify abstract dependency relationships. One can either express the relationship on any version of an ontology via an Ontology IRI or express the relationship on a single version of an ontology via an Ontology Version IRI. This results in a situation where one has to choose between under-specifying the compatibility to specific versions of the dependency and risk accidental upgrades or over-specifying the dependency by providing an exact version and making upgrading between versions a manual process.

OWL2 also leaves open how an ontology file should be retrieved, even though it does outline a loading mechanism. This can lead to a multitude of problems when trying

to retrieve the ontology document backing an IRI. There is no guarantee that any two requests to retrieve the ontology file by URL result in the same document due to a lack of mechanisms for testing its integrity, like, e. g., Subresource Integrity [6]. Since there is no guarantee that the machine serving the URL will be continuously online, a server going offline can result in an unretrievable ontology and an incomplete set of dependencies. A growing amount of dependencies from different sources means that the reliability of the retrieval process is subject to an increasing number of points of failure.

As a prominent standard, OWL forms the basis for ontology formalization in Plow, while supplementing its dependency management primitives with ones suitable to resolve the outlined shortcomings.

2.2. Ontology Repositories

Ontology repositories help improve the discoverability of ontologies. They vary in sophistication, with some consisting of a simple static HTML page linking to a list of Ontology IRIs and more sophisticated ones including fully interactive web applications through which users can publish ontologies in a self-service manner. A recent overview of related work in this area is provided by [7].

As Plow contains a registry component that most closely resembles existing self-publishing repositories like BioPortal, we survey the existing solutions in that space. While ontology repositories like DBpedia Archivo do not fit our model as closely, they are also of interest as inspiration on how to keep compatibility with the existing ecosystem and how the registry could be provided with an initial set of ontologies.

2.2.1. Self-Publishing Repositories

BioPortal [8] is a repository of approx. 1000 ontologies from the biomedical domain. It features a web UI for searching and discovering ontologies, through which it exposes many of the core metadata and statistics that can be of interest when evaluating an ontology for its quality and an entity explorer that can be used to explore the contents of an ontology. BioPortal incorporates access control, which can be used to restrict the visibility of an uploaded ontology to a certain set of users, and it exposes a REST API, which can be used with an API key for programmatic ontology submission and retrieval.

Ontohub [9] is a self-service ontology repository providing distributed version-control based on Git, following the standards of the Open Ontology Repository Initiative (OOR). It uses the Distributed Ontology, Modeling and Specification Language (DOL) to enable uniform support for all kinds of formal knowledge representation languages.

2.2.2. Harvesting Ontology Repositories: DBpedia Archivo

One notable recently added ontology repository is DBpedia Archivo [7]. Archivo is a repository of ontologies based upon discovery via existing ontology repositories, as well as inspecting all previously discovered ontologies for imports. It continuously crawls all known ontologies and creates new archival snapshots when a change is detected. As an additional source of ontologies, new URLs can be submitted for harvesting, which will be added to the repository for continuous checking upon a successful initial crawl. One unique feature is Archivo's support for Semantic Versioning (SemVer) [10]. As the original ontologies do not provide SemVer versions, Archivo compares the changes

between the new snapshot and a previous one. Based on the observed changes, it tries to derive an appropriate version through a set of rules that mimic the SemVer rules and provides that version as a semantic versioning "overlay".

2.3. Programming Language Package Management

Plow's dependency management is based on recent advances in tools for dependency management in programming languages. These vary greatly in sophistication and size of ecosystem. We take a look at the most recent and prominent ones, to identify the core mechanics that Plow can utilize to support a rapidly evolving ecosystem of ontologies.

2.3.1. NPM and other Package Managers

In the JavaScript/Node.js ecosystem, the Node Package Manager (npm) project provides CLI package management tooling, as well as a public repository of npm packages [11]. With almost 2 million published packages [12], it has also been the subject of research for open source software development methodologies. In terms of functionality, it reads a set of abstract dependency requirements in the form of SemVer version ranges from a manifest file (`package.json`), runs its dependency resolution algorithm and writes the resulting set of package versions into a Lockfile (`package-lock.json`).

Many recent programming language communities have recognized the importance of establishing package management tooling (as well as an integrated registry) early in the lifetime of the language ecosystem. By providing integrated tooling usually consisting of a CLI interface for package consumption and publication, a centrally hosted self-service package registry with a REST API and web UI to allow for package discovery, all core needs for package management are met. Since some features can involve components of the package management solution, having a common party maintain those components together can greatly simplify the evolution of the package management.

2.3.2. OntoMaven

OntoMaven [13] is a tool for managing transitive dependencies of modular ontologies based on the build automation tool Apache Maven [14] and its accompanying repository system. In addition to resolving transitive ontology dependencies and downloading a copy for local use, OntoMaven also includes functionality to create an OASIS XML-Catalog file, in which the local copies of external URIs are referenced. This file can be read by Protégé and other tools to allow them to load an ontology by reading it from disk instead of trying to retrieve the ontology file via HTTP. As Maven is traditionally used in software development utilizing Java (and other JVM-based languages), OntoMaven is well suited for software engineers with prior experience in that area. OntoMaven (like Maven) is only available via a command-line interface, which can make it inaccessible to ontology engineers who are used to GUI-based workflows and have little to no experience with command-line interfaces.

OntoMaven builds on top of Maven repositories, for which multiple commercial hosting offerings exist, which should provide low operating complexity when running an OntoMaven repository. In practice, the lack of an official repository to be used when starting a new ontology development project can negatively impact the solution's adoption due to the necessary effort of having to set up a repository before publishing an ontology.

3. General Approach

3.1. Simplifying Ontology Reuse

While ontology reuse is a well-established concept in general, the concrete reuse of ontologies can be challenging [15, 16]. Current primitives are not sufficient to support and automate the maintenance of ontologies that depend on a large number of other ontologies. By providing a framework that simplifies the inclusion and maintenance of external dependencies, the effort required to maintain an individual ontology can be reduced. This approach would finally make it possible to maintain a set of smaller, modular ontologies where previously, one single bigger ontology would have been used – out of sheer necessity – to reduce the overhead of manual maintenance processes. The implementation of tooling that additionally can assist in avoiding common pitfalls of ontology reuse and that streamlines the publication process can lower the barrier for enabling domain experts to provide their formalized knowledge to third-parties.

In Plow, we reduce the effort required for maintenance and consumption of ontology packages by specifying dependencies in abstract terms via SemVer ranges, which can automatically be resolved and updated through an automated version resolution mechanism. By providing this mechanism to end-users through a CLI and a GUI, we can cover a wide spectrum of use cases from deep integration into automatic software delivery pipelines facilitated by developers to the development of ontologies by ontology engineers with little or no software engineering experience.

3.2. Industry-Readiness in Consumption and Publication of Ontologies

Current solutions for building and consuming ontologies do not offer the same level of sophistication as their programming language counterparts, hindering the adoption of ontology-based systems in industry. Given that in many scenarios, ontology contents are being interpreted to guide the dynamic execution of software systems or display data to end-users, they can act as a potential delivery method for malware.

The careless inclusion of dependencies maintained by third-parties can allow for the delivery of malicious payloads without the application's maintainer being aware of this. Thus, package managers which serve as the central component for managing external dependencies, have become crucial for ensuring safety with regard to attacks against software supply chains. Furthermore, to ensure compliance with intellectual property rights, package managers have adopted functionalities to allow for the automated analysis of the licenses of all transitive dependencies of a project.

In Plow, we aim to meet these requirements by providing a dependency resolution mechanism that, at its core, is designed to be reproducible via the production of Lockfiles. Furthermore, in many aspects, Plow's design is hardened against cyber-security threats. This includes cryptographically secure integrity hashes that prevent undetectable tampering with package versions during their retrieval, which allows for delivery via third-party content delivery networks as well as using them for offline caching solutions. In addition, we integrate security features like multi-factor authentication into the service registry to prevent account takeover attacks. The availability of a Lockfile that contains the fully resolved dependency tree also allows for deeper transparency into the ontology supply chain, enabling automatic license conformance checks via SPDX 2.1 license ex-

pressions [17], which are enforced as required metadata on submission to the registry service. As ontologies created in a business context may be considered valuable intellectual property, the registry provides the option to publish private packages with access control, which by default are only available to the package maintainer.

3.3. Domain and Programming Language Agnosticism

For some of our requirements (see above), partial solutions exist in some domain-specific ontology building ecosystems, which have invested in tooling to support these requirements. As these tools are often tailored to the needs of the specific domains, they do not generalize well to other domains. Similarly, in the past, implementation efforts were mostly focused on the JVM [18] and Python [19] programming language ecosystems.

Using our approach for building a package management solution, that at its core is agnostic to domains and programming languages, the upfront cost to establish ontology building and publication in new domains and programming languages can be significantly reduced. Plow supports making ontologies of different domains available on a common platform, building ontologies that span multiple domains and simplifying the reuse of established ontologies of each of the respective domains, increasing the utility of domain-specific ontologies through reuse.

3.4. Compatibility with Existing Ecosystem

As a lot of valuable work has gone into the development of ontologies in the past, a new system should try to be compatible with existing ecosystems. In Plow, we build upon OWL and provide a solution existing ontology maintainers can migrate towards. Plow is also designed in a way that makes it easy to integrate into existing workflows.

One aspect where Plow can break compatibility with an existing ecosystem is regarding its stance to the HTTP retrieval aspect of the Linked Data principles. In Plow, ontologies are not retrieved via HTTP from their URIs (from potentially different parties) – a property that in practice many ontology maintainers struggle to uphold. Instead it relies on a single known source of truth with built-in measures for reliability.

4. Implementation

The implementation of Plow has been inspired by Cargo [20], the official package manager for the Rust programming language, and its corresponding official registry crates.io [21]. To the best of our knowledge, technical descriptions of these tools are not available, so we attempt to outline the major architectural decisions that we adopted from their system alongside our novel additions. We deviate from Cargo's architecture with regard to our implementation's support for private ontology packages, which enables a mixed public-private registry service. Our implementation of dependency resolution is also different due to the nature of the preexisting ontology ecosystem.

4.1. Ontology Package

In its current implementation, an *ontology package* is equivalent to a Turtle-serialized [22] file containing a single OWL ontology. To match the agricultural-inspired naming

of Plow, an ontology package is nicknamed a *field*. All metadata utilized for package management is added as annotations to the `owl:Ontology` entity in the ontology, with the annotation properties (Table 1) defined in our ontology[1] with the prefix `registry`.

Annotation Property	Purpose	Repeatable
`:ontologyFormatVersion`	Version identifier for the ontology format itself	No
`:packageName`	Identifier for the ontology package	No
`:packageVersion`	SemVer version for the ontology package version	No
`:dependency`	Name and SemVer version range of a package this package depends on	Yes
`:canonicalPrefix`	Canonical prefix to be used in place of the ontology IRI when importing this ontology in another ontology	No
`:licenseSPDX`	SPDX 2.1 license expression	No
`:license`	Free-form field for non-standard license information	Yes
`:author`	Name and contact information for a author of the package	Yes
`:homepage`	URL for a homepage of the ontology	Yes
`:documentation`	URL for a page with documentation for the ontology	Yes
`:repository`	URL for a VCS repository where the ontology is maintained	Yes
`:category`	Category according to the categorization system defined by the Registry	Yes
`:keyword`	Self-defined keyword to be used in discovery mechanisms of the Registry	Yes

Table 1. Overview of annotation properties defined in the `registry` ontology

Packages are identified through a combination of a *namespace* (prefixed by `@`) and a *package name*, e. g., the package `infrastructure` in the namespace `software` can be referred to by the identifier `@software/infrastructure`. Namespaces allow the grouping of packages that have a common set of authors allowed to publish packages belonging to that namespace. This property should be ensured by the registry service. Organizing packages in namespaces reduces the attack surface for "typosquatting" [23], where a malicious party publishes malicious code under a package name that is a common misspelling of a popular package, one of the most common supply chain attacks on package managers.

Ontology Package Version and Dependencies

To identify a specific version of a package, we use the *SemVer* [10] versioning scheme. Each version is assigned a *version number* that is unique across all versions of the same ontology package inside a registry (a combination of package name and version number). In SemVer, a version number follows the MAJOR.MINOR.PATCH format, where the version number sections are incremented for a newly published version based on the level of compatibility of the changes made to the package since the previously published version. This level of compatibility is derived from categorizing changes to the public API of the package to be one of "backwards compatible bugfixes" (increase in PATCH

[1] http://field33.com/ontologies/REGISTRY/

version), "backwards compatible addition of new features" (increase in MINOR version), and "breaking changes" (increase in MAJOR version).

By encoding information about the "compatibility" in the version number, this enables a package consumer to answer the question of whether it is possible to upgrade between package versions by looking only at the package version, without a need to inspect the whole set of changes that have been applied between the package versions. This simplification allows for a more streamlined approach to upgrading the versions of the packages one depends on, which was previously a labour-intensive manual process.

Programming language package managers have also been exploiting this property by using *SemVer version ranges* for specifying the acceptable set of versions for a dependency. As an example, by specifying a *dependency requirement* for the package `@foo/bar` with the version range `>=1.2.3, <2.0.0`, we can tell that the package with the version `1.7.1` falls within the specified range and satisfies the dependency requirement. Conversely, we can also tell that the version `1.1.0` does not satisfy the requirement, as according to the semantics of the MINOR position in the SemVer version number, it lacks new functionality introduced in version `1.2.0`.

4.2. *Registry Service*

A *Registry* is an abstract entity consisting of an *Index* and an *Artifact Store*. This interface can be implemented with different characteristics, fitting different use cases, e. g., for non-collaborative development purposes, a registry can be constructed where Index and Artifact Store exist in their entirety on disk on a single machine, while for collaborative development on the same Registry, the bulk of the Index and Artifact Store contents are stored on a remote server, and only the required resources are copied to the local machine. In particular, in our implementation, we developed a *Registry Service*, which runs as a remote HTTP server, to serve the requests from a GUI or CLI client. In addition to supporting the core registry functionality required for package dependency resolution, it also provides a Web UI for searching and discovering published ontology packages.

4.2.1. *Index*

The *Index* contains the metadata for all published packages and their versions that were published to the Registry (Listing 1 shows an example). By restricting itself to contain only the metadata required for dependency resolution, the total size of the Index stays small enough to fit on a local development machine easily. By storing the Index in a Git repository, we can leverage the existing synchronization mechanism of Git to efficiently synchronize the Index between machines. To support private packages, a separate Index served via authenticated REST endpoints of the Registry Service is employed. The Index records a cryptographic checksum for a package version under the `cksum` key.

Listing 1: Example of an Index metadata file for a package with a single published version and a single dependency

```
{
  "versions": [
    {
      "name": "@test/package1",
      "version": "0.1.0",
```

```
        "cksum": "ef7118...5704f04",
        "ontology_iri": "http://example.com/@test/package1/",
        "deps": [
          {
            "name": "@test/package2",
            "req": ">=0.1.0"
          }
        ]
      }
    ]
}
```

4.2.2. *Artifact Store*

The *Artifact Store*, which complements the Index, does not have any knowledge of ontology package metadata. It contains all the original copies of the published packages and makes them available for download.

4.2.3. *Additional Metadata Storage*

To support functionality related to package ownership, user authentication, access control for private packages and mapping package versions to artifacts, the Registry Service also maintains an additional metadata storage in the form of a relational database management system (RDBMS).

4.3. *Ontology Management Lifecycle Actions*

Based on the components outlined in Section 4.2, we can now define the major process steps [24] for package management. At the core is the pipeline of *extract*, *resolve* and *retrieve* (Figure 1), which is completely automated. As each of the steps is independently reproducible, the whole pipeline is reproducible given the same state of the ontology document, Lockfile and Registry. In the common case of no changes to the ontology document, an existing Lockfile, and previously retrieved ontology packages, the need for communication with the Registry is completely eliminated, enabling the same pipeline to be reused in an offline environment.

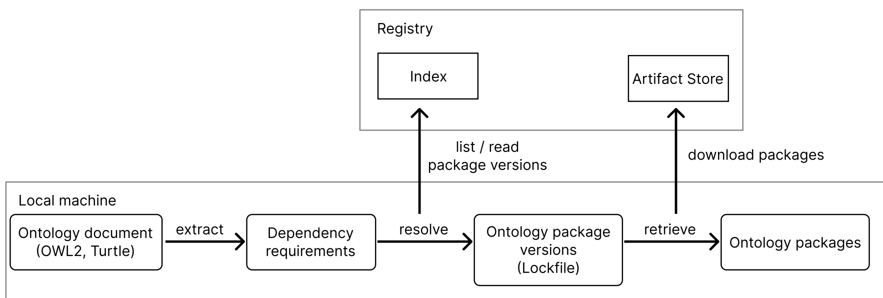

Figure 1. Outline of the package management process steps and their interaction with the registry

4.3.1. extract

The *extract* step extracts all *dependency requirements* from the ontology source file by parsing the file according to its serialization format (in our case Turtle), and reading triples with the `registry:dependency` predicate.

4.3.2. resolve

The *resolve* step takes the *dependency requirements* from the *extract* step, and computes a set of *concrete dependency versions*, which is serialized and persisted as a *Lockfile*. To do this, a dependency resolution algorithm, which has read access to the Index, is run to determine a set of dependency versions in a way that the dependency requirements for all transitive dependencies are satisfied. If the resolve step has been run previously, the existing Lockfile will be used as an additional input to the resolve step, which allows previously selected package versions to take precedence in the dependency resolution. In the case of an existing Lockfile and no changes to the dependency requirements, this results in the step being reduced to a No-Op, as all dependency requirements are already fulfilled, eliminating the need to read from the Index and making this common case fully reproducible. In the case of an existing Lockfile and a change to the dependency requirements, this results in only the required amount of (transitive) dependencies being updated to satisfy the changed dependency requirements. This increases the stability of addition, removal, and update operations to the dependency requirements, as it prevents unintended updates of dependencies unrelated to the change in requirements.

Our implementation is built on an implementation of the PubGrub version solving algorithm [25, 26]. One aspect that many dependency resolution algorithms differ in is the decision what to do about conflicting package versions in the dependency trees, with the two main options being a) allowing multiple different versions of a package to be present in the dependency tree, which allows some sets of dependency requirements to be successfully solved, or b) ensuring that only a single version of the package is present in the dependency tree.

Allowing for multiple versions of a package to be present as dependencies, is an option that is often chosen in package management for compiled languages, where at compile time the identity of an imported entity can be traced back to one specific package version and usage of that entity is restricted to only occur with other entities from the same package (e. g., type `Foo` of package version `1.2.0` can be used as an argument for function `bar` of package version `1.2.0`, but it cannot be used with function `bar` of incompatible package version `2.0.0`). As many ontology tools today rely on using the IRI (which is the same across different versions of an ontology package) for identifying an ontology, importing multiple versions of an ontology dependency is generally not supported. As we want to keep compatibility with those tools, we chose the option of only permitting a single version of an ontology package in the dependency tree for Plow.

4.3.3. retrieve

The *retrieve* step takes the *concrete dependency versions* resulting from the *resolve* step, and downloads the package versions from the Artifact Store for local usage. This step may have slight variations (or steps that run directly after it), depending on the use case. For the local editing of ontologies, we provide a variation of the step with similar functionality to the *OntoMvnImport* plugin of OntoMaven [13], where an XMLCatalog file is created with locations of the local copies of ontology packages.

4.3.4. update

With the *update* action we can update the dependencies as specified in the ontology. As this operation may result in an unresolvable set of dependency requirements, this should be modeled as a fallible operation, where a backup of a previous state with good dependency requirements is kept, which can be restored in case of failure.

4.3.5. publish

In the *publish* action, an ontology maintainer submits an ontology package version to the Registry for inclusion in the Index and Artifact Store, making it available to other users and consumers of the Registry. During submission, the Registry Service validates the presence of all metadata required for dependency management.

4.4. CLI and GUI

To make these actions available to end users, we provide the CLI command `plow`, which targets developers and continuous integration and continuous delivery (CI/CD) use cases. For less tech-savvy users we also provide a GUI application. Both support all lifecycle actions (Section 4.3).

4.5. Availability of the Tool

The code for the CLI, GUI, the library with their common logic, as well as a reference implementation of the registry service that outlines the REST API are open sourced under a permissive license and made available on GitHub.[2] We also provide a fully-featured hosted version of the registry service,[3] for which we decided not to open-source the underlying code, as it is tightly integrated with our infrastructure and relies on paid third-party services.

5. Plow in Use

Through the interaction with business clients, we are able to highlight the advantages of Plow for maintaining a set of ontology packages. By sharing our experience in building a Software as a Service (SaaS) product which integrates Plow we can also highlight how Plow enables a highly dynamic usage of ontologies in a user-facing product.

5.1. Dependency Tree Maintenance

For multiple of our clients we maintain a set of roughly 30 ontology packages related to the domain of software development, agile software development practices and related metrics. This set of packages is strongly interlinked, with the two most pathological cases being the ontologies that define the core concepts of software development and metrics, which are used by most of the other packages either directly or indirectly.

[2] https://github.com/field33/plow
[3] https://registry.field33.com

As each of the clients has separate release cycles during which the package contents are fixed, and even during a new release only a subset of the ontologies is supposed to be updated, the naive approach of just maintaining a single "current" version for each ontology was not an option, as updating that ontology could accidentally deliver changes to customers that should not receive them. To satisfy these requirements, the predecessor system to Plow was introduced, which could handle package management of ontologies with transitive dependencies, but with the limitation that only exact version requirements between ontologies could be specified.

While this system worked well, with an increasing number of ontologies, it put more strain on the ontology maintainers. Due to the fixed version requirements, even a minor change (e. g., fixing a typo) to one of the core packages that many other packages depended on became cumbersome to fix. When a new version of the core package was released, the fixed version requirement in all dependant packages had to be adjusted and a new version of them released, and then the same had to be done for the packages dependant on them, and so on. This meant that a change in one of the core ontologies triggered a cascade of manual changes and releases in the dependency tree, increasing the work the ontology maintainers had to do, and increasing the delay of new ontology-based features to the clients, as changes started to be batched into bigger releases.

Through the introduction of Plow, with its SemVer-based dependency requirements, this problem was completely resolved, allowing ontology maintainers to focus their time on expanding the scope of ontologies and focusing on breaking changes between ontology versions. Smaller bug fixes and backwards-compatible feature additions are now published under the appropriate increases of MINOR and PATCH version, and can be updated by re-running the dependency resolution.

5.2. Field 33: An Integrated Use Case

Plow is one of the central components of the Field 33 platform, a SaaS knowledge graph product. Each customer or user (in the form of an organization) maintains their own knowledge graph, with a different set of ontologies loaded for each of them. As users are not expected to have prior knowledge of semantic data technologies or ontology management, they are presented with a set of options in the form of domain packages like "Software Development", which they can load into their knowledge graph, together with a desired level of version stability towards automated updates.

With the help of Plow, this is translated into a set of SemVer-based dependency requirements, which are resolved to a set of ontology package versions that are loaded into the customer's graph (see Figure 2). To make new versions of ontologies available to customers, a curation layer is employed that builds on top of the stable identifiers for Plow package versions. By providing the Plow Registry as the interface for publication and consumption of the ontologies and reducing the involvement of our company to the role of a platform provider, the task of ontology maintenance can also be done by public contributors, contracted third-parties or customers if they have specialized staff.

6. Evaluation and Limitations

To evaluate Plow, we compare it with four similar solutions that provide state-of-the-art capabilities in their respective areas. The comparison focused upon selected features in

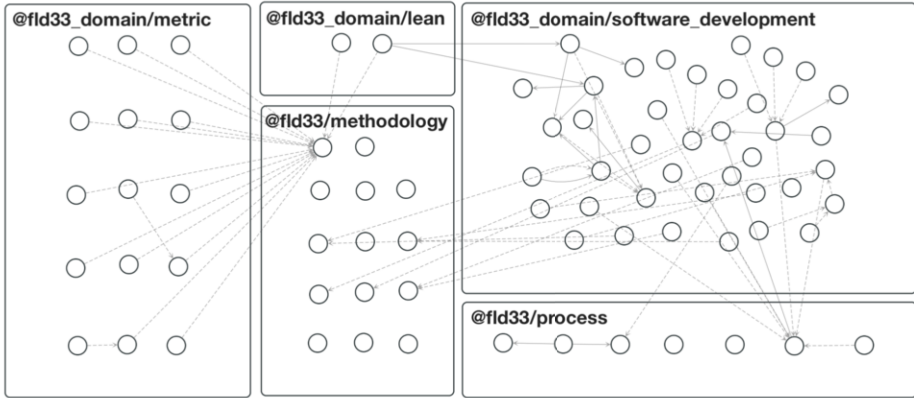

Figure 2. Plow package network visualization, showing package boundaries and contained ontology concepts, for a subset of the Software Development domain in the Field 33 product

the categories of access to ontologies and usability in industry-ready ontology engineering (see Section 3) in a comparison matrix (Table 2, adapted from [7]).

Plow is the only solution providing an integrated approach to ontology management by providing a state-of-the-art registry service and bundling it with tooling for ontology engineers to enable first-party publishing and consumption of ontology packages. While both Archivo and Plow provide unified SemVer version numbers, there is a difference in approach, where Plow relies on version numbers provided by maintainers, where Archive has to try and derive a fitting version number from the ontologies it scrapes. OntoMaven has the biggest workflow similarities when it comes to the publication and consumption of ontology packages, though OntoMaven is lacking reproducibility, as well as a hosted public instance of its repository.

Table 2. Solution comparison based on features in the areas of access and usability

Dimensions			Access				Usability			
Solution	TY	DO	SM	OV	OF	MA	MT	DR	SC	HS
Plow	R,D	I	– / ●[1]	● / ●	●	● / –	● / ●	● / ●	●	●
Archivo	R	I	● / ○	● / ●	●	○ / ●	– / –	– / –	–	●
BioPortal	R,D	S	– / ●[1]	○ / –	○	●[1] / ●	– / –	– / –	–	●
Ontohub	R,D	I	– / ●[1]	○ / –	○	●[1] / –	– / –	– / –	–	●
OntoMaven	D	I	– / ●[1]	○ / –	●	○[2] / –	● / –	● / ○[3]	–	–

● = provides property; ○ = provides property partially; – = does not provide property; **TY**: Solution type: (R)egistry/Repository/Archive, (D)evelopment platform; **DO**: ont. domain focus: (S)pecialized vs. (I)ndependent; **SM**; mode of (automated) ont. submission: *inclusion request/direct upload*; **OV**: ont. version numbers: *unified/SemVer*; **OF**: access to ont. in one unified format; **MA**: ont. metdata access via *REST API/SPARQL*; **MT**: maintainer tools: *CLI/GUI*; **DR**: *dependency resolution/including Lockfile*; **SC**: supply-chain security measures; **HS**: hosted public instance

[1] account/login required; [2] depending on used Maven repository service;
[3] can be achieved via a Maven plugin;

As we have focused on establishing the functionalities required for full end-to-end package management workflows, Plow is currently limited in the variety of supported ontology languages and serialization formats, by only supporting OWL ontologies serialized as Turtle. We also recognize that parts of the assumptions regarding programming language managers and their inherent connection to their respective ecosystems are based on subjective experience by interacting with them, which is why we would welcome additional research in these areas.

7. Conclusions and Future Work

This article first outlines the broad requirements for a state-of-the-art package management solution for ontologies and our specific approach. We then highlight details of the implementation of the individual components that make up our unified approach, Plow. By taking a look at how our tool has been used in a production setting, we illustrate its ease of use with regard to maintenance of a set of interlinked ontologies and integration into a user-facing application.

With the core functionality of our package management solution established, our next steps include the implementation for support of a wider range of established ontology formats, to allow for consuming most existing popular ontologies as Plow packages. We will also add support for consuming packages from multiple registries at the same time, as well as adding automated mirroring of indices and artifacts between registries, to be able to support a wider range of industrial use cases.

References

1. Andreessen M. Why Software Is Eating the World. 2011 Aug. Available from: https://a16z.com/2011/08/20/why-software-is-eating-the-world/
2. Grangel-Gonzalez I, Baptista P, Halilaj L, Lohmann S, Vidal ME, Mader C, and Auer S. The Industry 4.0 Standards Landscape from a Semantic Integration Perspective. *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Limassol, Cyprus: IEEE, 2017 :1–8
3. GAIA-X Open Work Package Self-Description. GAIA-X Core Ontology. Rev.: 21/12. 2022. Available from: https://gaia-x.gitlab.io/gaia-x-community/gaia-x-self-descriptions/core/core.html
4. Smith MK, Welty C, and McGuinness DL. OWL Web Ontology Language Guide. Available from: https://www.w3.org/TR/owl-guide/
5. W3C OWL Working Group. OWL2 Web Ontology Language Document Overview (Second Edition). Available from: https://www.w3.org/TR/owl-overview/
6. W3C Recommendation: Subresource Integrity. Available from: https://www.w3.org/TR/SRI/
7. Frey J, Streitmatter D, Götz F, Hellmann S, and Arndt N. DBpedia Archivo: A Web-Scale Interface for Ontology Archiving Under Consumer-Oriented Aspects. *Semantic Systems. In the Era of Knowledge Graphs*. Ed. by Blomqvist E, Groth P, de Boer V, Pellegrini T, Alam M, Käfer T, Kieseberg P, Kirrane S, Meroño-Peñuela A, and Pandit HJ. Vol. 12378. Cham: Springer, 2020 :19–35

8. Noy NF, Shah NH, Whetzel PL, Dai B, Dorf M, Griffith N, Jonquet C, Rubin DL, Storey MA, Chute CG, and Musen MA. BioPortal: Ontologies and Integrated Data Resources at the Click of a Mouse. Nucleic Acids Research 2009; 37:W170–W173
9. Codescu M, Kuksa E, Kutz O, Mossakowski T, and Neuhaus F. Ontohub: A Semantic Repository Engine for Heterogeneous Ontologies. Applied Ontology 2017 Nov; 12. Ed. by Baclawski K and Bennett M:275–98
10. Preston-Werner T. Semantic Versioning 2.0.0. Available from: https://semver.org
11. Npm – Package Manager and Registry for the JavaScript Community. Available from: https://www.npmjs.com
12. Module Counts. Available from: http://www.modulecounts.com
13. Paschke A and Schäfermeier R. OntoMaven – Maven-Based Ontology Development and Management of Distributed Ontology Repositories. *Synergies Between Knowledge Engineering and Software Engineering*. Ed. by Nalepa GJ and Baumeister J. Vol. 626. Cham: Springer, 2018 :251–73
14. Apache Maven. Available from: https://maven.apache.org
15. Shimizu C, Hammar K, and Hitzler P. Modular Ontology Modeling. Semantic Web Journal 2022. Available from: http://www.semantic-web-journal.net/content/modular-ontology-modeling-1
16. Verborgh R, ed. The Semantic Web: 18th International Conference, ESWC 2021. Lecture Notes in Computer Science 12731. Cham: Springer, 2021
17. Software Package Data Exchange (SPDX) Specification Version: 2.1. 2016. Available from: https://spdx.dev/wp-content/uploads/sites/41/2017/12/spdxversion2.1.pdf
18. Horridge M and Bechhofer S. The OWL API: A Java API for OWL Ontologies. Semantic Web Journal 2011; 2:11–21
19. Jean-Baptiste L. Ontologies with Python: Programming OWL 2.0 Ontologies with Python and Owlready2. Berkeley, CA: Apress, 2021
20. Cargo – The Rust Package Manager. Available from: https://github.com/rust-lang/cargo
21. Crates.io – Rust Package Registry. Available from: https://crates.io
22. Beckett D, Berners-Lee T, Prud'hommeaux E, and Carothers G. RDF 1.1 Turtle. W3C Recommendation. https://www.w3.org/TR/turtle/
23. Duan R, Alrawi O, Kasturi RP, Elder R, Saltaformaggio B, and Lee W. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. *28th Annual Network and Distributed System Security Symposium, NDSS*. 2021
24. Boyer S. So You Want to Write a Package Manager. 2017. Available from: https://medium.com/@sdboyer/so-you-want-to-write-a-package-manager-4ae9c17d9527
25. Weizenbaum N. PubGrub: Next-Generation Version Solving. 2018. Available from: https://nex3.medium.com/pubgrub-2fb6470504f
26. PubGrub Version Solving Algorithm Implemented in Rust. 2022 May. Available from: https://github.com/pubgrub-rs/pubgrub