H2020-ICT-2018-2-825040

Rational decomposition and orchestration for serverless computing

Deliverable D4.5

Graphical Modelling Tool I

Version: 1.0

Publication Date: 19-December-2019

Deliverable Card

Deliverable	D4.5
Title:	Graphical Modelling Tool I
Editor(s):	Michael Wurster (UST) and Vladimir Yussupov (UST)
Contributor(s):	Michael Wurster (UST), Vladimir Yussupov (UST)
Reviewers:	Giuliano Casale (IMP), Damian A. Tamburri (TJD)
Type:	Report
Version:	1.0
Date:	19-December-2019
Status:	Final
Dissemination level:	Public
Download page:	http://radon-h2020.eu/public-deliverables
Copyright:	RADON consortium

The RADON project partners

IMP	IMPERIAL COLLEGE OF SCIENCE TECHNOLOGY AND MEDICINE
TJD	STICHTING KATHOLIEKE UNIVERSITEIT BRABANT
UTR	TARTU ULIKOOL
XLB	XLAB RAZVOJ PROGRAMSKE OPREME IN SVETOVANJE DOO
ATC	ATHENS TECHNOLOGY CENTER ANONYMI BIOMICHANIKI EMPORIKI KAI TECHNIKI ETAIREIA EFARMOGON YPSILIS TECHNOLOGIAS
ENG	ENGINEERING - INGEGNERIA INFORMATICA SPA
UST	UNIVERSITAET STUTTGART
PRQ	PRAQMA A/S

The RADON project (January 2019 - June 2021) has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 825040

Executive summary

This document is the milestone version of the deliverable which presents the results of the Graphical Modeling Tool (GMT) development and draws the connections with the previous deliverables, namely D2.1 "Initial requirements and baselines" and D4.3 "RADON Models I". In addition, this document will serve as a basis for the next iteration of GMT deliverable. The major goal of this document is to provide a comprehensive description of the prototypical implementation of the concepts required to support RADON modeling approach using Eclipse Winery as a baseline. In the course of this document, we present the architectural changes to enable Eclipse Winery for modeling based on the TOSCA YAML Simple Profile specification while preserving backward compatibility with TOSCA's former standard in XML. Further, we introduce Winery extensions to support the usage of multiple TOSCA type repositories that can be versioned using Git. Finally, we present enhancements to Winery's frontend components to create and adapt TOSCA type definitions specifically to the YAML-based standard, e.g., to restrict modeling of relationships between nodes based on matching TOSCA requirements and capabilities. All GMT-related artifacts described in this document are publicly available on GitHub[1], whereas the discussed modeling constructs that were presented in D4.3 are available in the RADON particles repository[2].

[1] https://github.com/OpenTOSCA/winery/tree/project/radon
[2] https://github.com/radon-h2020/radon-particles

Glossary

CSAR	Cloud Service Archive
FaaS	Function as a Service
GMT	Graphical Modeling Tool
GUI	Graphical User Interface
IDE	Integrated Development Environment
MSA	Microservice Architecture
NFR	Non-Functional Requirement
TOSCA	Topology and Orchestration Specification for Cloud Applications
VM	Virtual Machine
WP	Work Package
Tw.n	Task n in Work Package w
Yn	Year n
YAML	YAML Ain't Markup Language
XML	Extensible Markup Language
CDL	Constraint Definition Language
IaC	Infrastructure as Code
CI/CD	Continuous Integration / Continuous Delivery

Table of contents

1 Introduction

The RADON project focuses on facilitating DevOps processes in software engineering which involve microservices, serverless and FaaS-based architecture elements, as well as data pipelines design, development, and operations. One part of such processes is to speed-up the design/assembly of desired application models and involved architecture elements. To accomplish this task, in D4.3 "RADON Models I" we introduced a set of modeling constructs allowing the representation of application topologies using a standard cloud modeling language, namely, OASIS TOSCA. These modeling constructs extend the existing types of the TOSCA standard in order to model serverless FaaS and data-driven microservice applications. To support modelers with means to use these modeling constructs in an interactive manner, a Graphical Modeling Tool (GMT) has been introduced. In this document, we elaborate on the initial version of GMT developed in the scope of WP4. This deliverable presents a first version of the GMT and extends Eclipse Winery with TOSCA YAML modeling support together with additional TOSCA-related features that were not available in Winery beforehand. This extension to support TOSCA YAML in Eclipse Winery is a major step as it enables Winery to import and export TOSCA CSARs that can be executed by the RADON orchestrator. Further, equally important, it facilitates DevOps activities such as infrastructure as code (IaC) by producing human- and machine-readable model files that can be managed using the RADON IDE. The described software in this deliverable is open source and available online on GitHub[3].

1.1 Deliverable objectives

This deliverable has the main objective to provide the foundation to support the modeling of RADON models, which can be broken down into the following goals:

1. Extend backend components of Eclipse Winery to handle RADON models based on the TOSCA Simple Profile standard in YAML.
2. Extend frontend components of Eclipse Winery to provide graphical modeling support to create and adapt RADON models based on TOSCA YAML.
3. Preserve backward compatibility with TOSCA's former XML-based standard.
4. Provide graphical modeling support to compose serverless FaaS and data-driven microservice applications based on the released RADON particles.
5. Provide the foundation to generate executable blueprints for consumption in the RADON orchestrator.

[3] https://github.com/OpenTOSCA/winery/tree/project/radon

1.2 Overview of main achievements

By this first-year deliverable, the main achievements have been to provide the foundation to graphically maintain RADON applications using the TOSCA Simple Profile standard in version 1.3. Having Eclipse Winery as a baseline and by extending it with the respective YAML-based modeling features, we publish a comprehensive modeling tool to graphically (i) create and adapt reusable modeling entities, such as TOSCA Node Types and Policy Types, (ii) compose RADON application structures in the form of TOSCA Service Templates, (iii) enrich existing RADON applications with test-related and performance-specific attributes using TOSCA Policies, and (iv) export a portable archive containing all information to execute the deployment by using the RADON Orchestrator. This forms the overall baseline for RADON to model serverless FaaS, microservices, their architectural composition, and data pipelines. Further, with this deliverable, we introduce a graphical TOSCA modeling tool that is the first of its kind supporting TOSCA's former XML standard as well as the newer one: TOSCA Simple Profile 1.3 in YAML.

1.3 Structure of the document

The current report is structured as follows: **Section 1** provides an overview of this deliverable by presenting the overall objective. **Section 2** gives an overview of requirements guiding this deliverable while **Section 3** introduces the final architecture of the GMT and provides further technical details. **Section 4** showcases the prototype based on the "thumbnail generation" use case scenario and, finally, **Section 5** concludes the report and reflects on the requirements by discussing their level of compliance. Lastly, **Appendix 1** presents the baseline of Eclipse Winery and highlights features that must be improved in order to achieve the requirements.

2 Requirements

Released in June 2019, the deliverable D2.1[4] presented an extensive analysis of initial RADON project requirements and use case scenarios that play the main role in guiding the technical, tooling-specific activities. In this section, we briefly summarize the requirements that have been relevant to guide this deliverable.

Table 1. RADON requirements relevant to the GMT.

ID	Category	Title	Description
R-T4.3-1	Must have	Integration into IDE	The graphical modeling tool (GMT, Winery) must provide a GUI which is integrated into the IDE.
R-T4.3-2	Must have	Navigation to business logic	In the GMT, it must be possible to navigate to the respective workspace where the source code is maintained.
R-T4.3-3	Should have	Navigation to deployment logic	In the GMT, it should be possible to navigate to a workspace where the deployment logic is located.
R-T4.3-4	Must have	Definition of constraints	In the GMT, a user must be able to define a set of constraints (based on the CDL) for one or more components.
R-T4.3-5	Could have	Import existing models	A user could be able to import existing models that can then be reused when creating new ones.
R-T4.3-6	Should have	Integration with other RADON tools	A user should be able to trigger certain tools from the modeling tool.
R-T4.3-7	Should have	Present verification result	Given a RADON model which does not comply with a set of hard constraints, the graphical modelling tool should be able to graphically represent the explanation generated by the verification tool.
R-T4.3-8	Must have	Test case specification	In the GMT, it must be able to use predefined or to create new test case specifications that a user can use to annotate modeled components.

[4] http://radon-h2020.eu/public-deliverables

R-T4.3-9	Must have	Modeling lots of elements	In the GMT, it must be possible to model an amount of up to two hundred elements (i.e., nodes, relations).
R-T4.3-10	Could have	Group modeling elements	The GMT could provide a feature to group or abstract certain elements in order to reduce the visual complexity of tens or hundreds of FaaS components.
R-T4.4-1	Must have	Export executable deployment model	The bundle which is exported from the modeling tool must be processeable by the RADON orchestrator.
R-T4.4-2	Could have	Export of different deployment model formats	The GMT could provide an option to export a blueprint in different formats to use other orchestration tools, such as OpenTOSCA or Terraform.
R-T4.4-3	Could have	Import of model in different format	The GMT could provide the possibility to import different output formats produced by the integrated RADON tools.

This deliverable report addresses partially the requirements from Table 1. We refer to this table later in this document and highlight when a particular requirement is met. The remaining requirements will be considered in a future deliverable.

3 GMT architecture and implementation

In this section, we outline the implementation roadmap and explain the architectural decisions made to fulfill project requirements and elaborate on the technical aspects of accomplished tasks.

3.1 Implementation goals and roadmap

The high level goal of the implementation is to support modeling of RADON applications using Winery, including microservice architectures, serverless and FaaS-based applications as well as data pipelines. Based on the requirements recapitulated in Section 2, we can highlight several distinct sub-goals, namely (i) modify Winery to enable modeling based on TOSCA YAML Simple Profile specification, (ii) support usage of so-called RADON particles that were introduced in the deliverable D4.3 “RADON Models I”, and (iii) support new graphical modeling workflow imposed by the YAML specification, e.g., TOSCA type implementations are no longer used and relations are modeled using a combination of requirements and capabilities.

To reach these goals, we make the following contributions by this first deliverable:

- We introduce an adapter-based Model Conversion component to translate Winery’s domain model into TOSCA YAML or TOSCA XML; preserving backward compatibility with TOSCA’s former standard in XML.
- A new YAML specific repository implementation is presented that can be used to (de)serialize Winery’s internal domain model into compliant TOSCA YAML files, which facilitates the DevOps methodology embraced by the project.
- We extend Winery’s repository implementations to support versioning using Git together with the YAML-based datastore.
- We present an extension to work with multiple TOSCA repositories versioned by different Git remote locations, e.g., the RADON particles and a self-managed type repository.
- We introduce a “YAML” feature flag providing the possibility to enable or disable certain parts of the backend and the frontend.
- Enhancements to Winery’s frontend components to create and adapt TOSCA type definitions according to the YAML-based standard, e.g., property, requirement, and capability definitions.
- Enhancements to Winery’s Topology Modeler component to restrict modeling of relationships between nodes based on matching TOSCA requirements and capabilities.

Essentially, the aforementioned subgoals target different parts of Winery architecture. For example, YAML support requires reconsidering the repository implementation in Winery, whereas the graphical modeling workflow modifications affect mainly the frontend components, i.e., management UI and topology modeler UI. In its turn, enabling convenient usage of RADON particles requires adding support for integrating multiple separate repositories in Winery. In the next subsections, we describe the specifics of each sub-goal starting with the support of TOSCA Simple Profile in YAML specification.

3.2 YAML support implementation choices

There are multiple possible options of how YAML-based repositories can be supported in Winery. For example, one possible, more intrusive option is to reimplement the underlying repository and all affected interfaces to work with YAML specification only. In contrast, another, less intrusive option we analyzed is based on introducing adapters for transparent conversion of internal data models without significant modifications in the existing interfaces. In the following we describe, why the final decision was made to choose a less intrusive modification.

Option 1: YAML-only reimplementation of the underlying repository. When choosing this option, multiple modifications will be required in Winery. Firstly, the *FilebasedRepository* class has to be changed to work with the YAML data models. In its turn, this decision will result in a need to reimplement serialization and deserialization of JSON data received from the frontend components for REST API's resources as well as modifying the utility classes responsible for processing the models data. One of the main advantages of this approach is a YAML-native repository in Winery, which resonates with the modern infrastructure as code [Mor16] approaches. Performance-wise it is also more efficient to avoid internal format conversion. In contrast, the largest drawback of choosing this option is that there will be no backward compatibility. This will result in a need to reimplement all research plugins Winery currently has. Moreover, since Winery is a part of OpenTOSCA ecosystem which includes, e.g., its own orchestration engine consuming XML-based TOSCA models, ignoring backward compatibility issues is not possible.

Option 2: Standalone YAML- and XML-based repository implementations. Another possibility is to introduce two standalone repository implementations: one for XML-based and one for YAML-based TOSCA specifications. This approach would also require having a REST API that supports both data models. This, for example, can be achieved if a language-independent superset of models can be derived and used for refining the existing REST API. The advantages of this approach are (i) support of both TOSCA specification types, (ii) modeling flexibility, and (iii) full backward compatibility. While in the long run, this option provides the best flexibility and is better performance-wise, achieving it is a long-running task due to several reasons. Firstly, deriving a superset of XML- and YAML-based TOSCA specifications is a labor-intensive process as both specifications cannot always be mapped directly due to difference in modeling approaches. For instance, TOSCA YAML specification no longer has such constructs as Node Type Implementation or Relationship Type Implementation. Introduced changes would also impose implementation requirements on Winery's frontend components, that would need to look and behave differently depending on which type of repository is used. For example, in case a YAML-based repository is used, capabilities and requirements will need to be defined differently since both specifications have varying constructs and approaches for this task. This option seems beneficial and can be considered an ideal implementation goal. Therefore, in the next option we describe a less intrusive approach to deal with RADON's requirements and which can be considered as an intermediary step before reaching the final goal of having two completely independent repository types.

Option 3: Adapter-based model translation. The third option we considered is a more relaxed version of Option 2. The general idea is also to implement a separate YAML-based repository type, which relies on the YAML data models. However, instead of reimplementing the interfaces and data processing utilities in frontend and backend components, a set of internal adapters is introduced to perform conversion of YAML files into internal XML-based data models and vice versa. As a result, Winery will work as-is for all user-based interactions, since the internally-used models remain unchanged. At the same time, the actual output obtained when modeling components and applications are TOSCA-compliant YAML files which, e.g., can be exported as CSARs and consumed by the RADON orchestrator. This option has several advantages: it reduces implementation complexity and at the same time sets the stage for moving towards Option 2, as the interfaces and frontend components can be gradually modified to support both implementations up to the point where no internal conversion is needed.

Decision outcome: Adapter-based model translation. Looking at the advantages and drawbacks of the aforementioned options, the Option 3 looks as a good candidate since it offers a good trade-off between fulfillment of the project's requirements in reasonable time and providing a good baseline implementation for evolving Winery towards a full-fledged YAML-based repository type working together with a standalone XML-based repository. In addition, the interfaces and involved components can be incrementally updated to address new requirements and provide new feature support. The main challenge at this point is, therefore, a definition of specification mapping together with conversion rules and respective architectural decision. In the next section, we elaborate on the details about the format conversion.

3.3 Winery model conversion

Winery in its current state comprises a Java domain model for TOSCA's XML (TOSCA XML) standard and for the TOSCA Simple Profile standard in YAML (TOSCA YAML) separately. This means, Winery has separate packages expressing Java model classes to cover each of the standard. However, as highlighted in Section A1.1, the Java model to reflect TOSCA YAML was only made to export a TOSCA XML model in TOSCA YAML format. This resulted in Java classes only considering how the TOSCA XML standard can be expressed in TOSCA YAML. Therefore, several TOSCA YAML modeling constructs are not supported at all, e.g., attributes or data types, and, therefore, were not part of the Java domain model.

For this use case, to export a TOSCA XML model in TOSCA YAML, Winery uses a Model Conversion component that is able to perform conversions between these two domain models. As this provides a solid baseline and due to the fact that Winery heavily uses the TOSCA XML model in its internal components, e.g., the research plugins and the REST API over HTTP, we decided to extend to converter as well as the TOSCA XML and TOSCA YAML model to support the complete set of features as suggested by the standard.

In the following, we present an excerpt of the conversion strategy and rules to convert (i) TOSCA entities in general, (ii) TOSCA node types, and (iii) TOSCA relationship types.

Conversion of TOSCA entities. A TOSCA entity comprises a set of common attributes. Other TOSCA types, such as TOSCA node types, extend this definition. Figure 1 shows the conversion rules of general TOSCA entities side by side. The TOSCA XML entity type is depicted on the left-hand side while the TOSCA YAML entity type is depicted on the right-hand side. The attributes "name", "derived_from", and "properties[Definitions]" exist in both domains and can be directly mapped. Notably, the "properties[Definitions]" attribute is a complex type and must be converted with a special care. Special conversion rules are used to make the mapping between the domains possible. However, the conversion rules are rather simple as these attributes are very similar. The two attributes "tags" and "metadata" have the same purpose in both domains. The list of key-value pairs, containing additional information, can be directly converted. Further, the attributes "abstract", "final" and "targetNamespace" exist only in TOSCA XML. However, these attributes can be stored as key-value pairs, since they data type will only be string or boolean. Therefore, there are special rules to map these attributes to the "metadata" attribute in TOSCA YAML. YAML entities offer a "version" attribute. At the moment, there is no equivalent in TOSCA XML and since Winery employs its own versioning approach (cf. Section A1.4) this attribute is not supported.

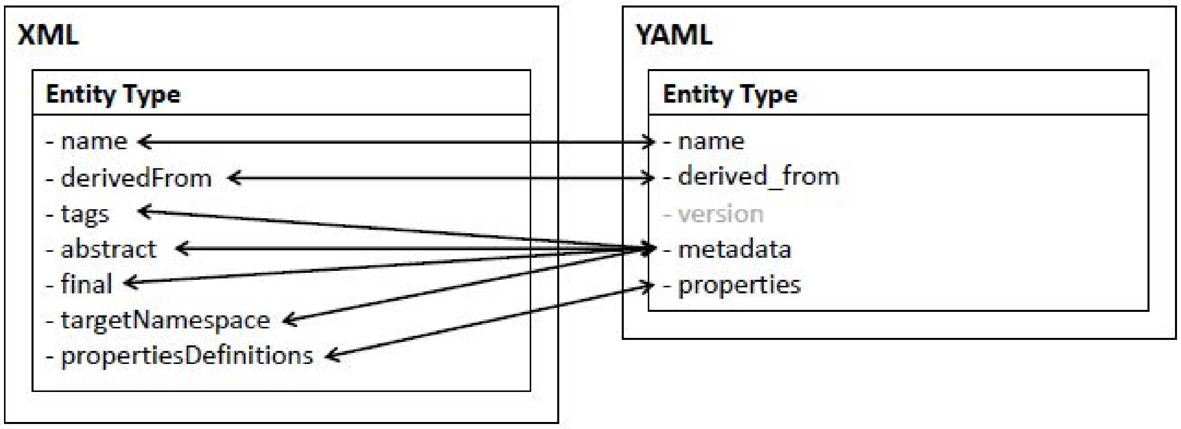


Figure 1. Conversion of general TOSCA entities.

Conversion of TOSCA node types. The node type definition of TOSCA YAML has been reworked and contains now considerably more attributes compared to TOSCA XML. The reason for this is that a node type in TOSCA YAML contains the complete information about properties, interfaces, node type implementations, and artifact templates in a condensed manner. In TOSCA XML, the base information of a node type (e.g., name, properties, etc.), the respective node type implementation, and the corresponding artifact templates are represented in three different domain objects that are wired in a loosely coupled way by referencing IDs.

On top of the attributes of general TOSCA entities, attributes regarding “requirements”, “capabilities, and “interfaces” can be mapped directly to the respective TOSCA YAML domain model. However, due to the reasons above, there are special conversion rules to transform the respective “Artifact Template” and “Node Type Implementation” domain classes to TOSCA YAML and back, as depicted in Figure 2. The result, since TOSCA YAML does not have corresponding domain classes, attributes such as “targetNamespace” are ignored and cannot be mapped. When converting back from TOSCA YAML to TOSCA XML, reasonable defaults are used by Winery’s converter in order to create valid TOSCA XML domain objects.

Conversion of TOSCA relationship types. Similar to node types, the relationship type definition in TOSCA YAML has been reworked and simplified. TOSCA XML defines instead of a combined domain class, three separate ones that are loosely connected by ID references. Figure 3 shows the conversion of relationship type in TOSCA XML to TOSCA YAML and vice versa.

The converter comprises special rules to translate the TOSCA XML “interfaces” attribute to TOSCA YAML while merging the content of the related domain classes respectively: “Relationship Type Implementation” and “Artifact Template”. In turn, the conversion from TOSCA YAML to TOSCA XML tries to provide reasonable defaults for attributes such as “targetNamespace”. Further, the converter provides auto-generated names, i.e., for the “name” attribute in created “Relationship Type Implementation” and “Artifact Template” objects.

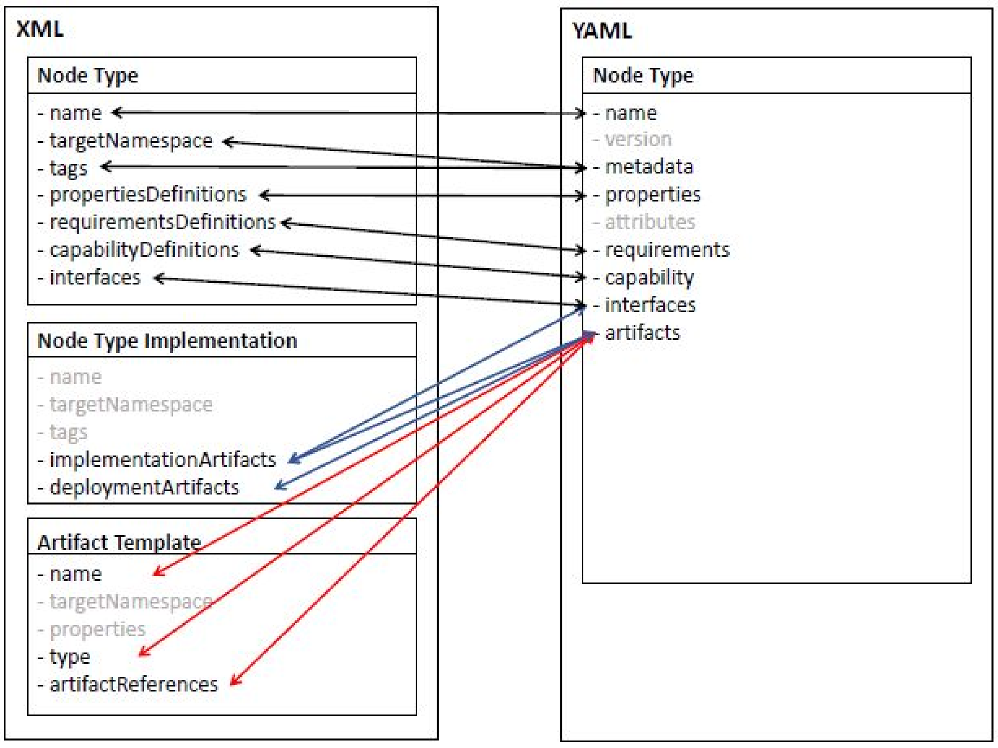


Figure 2. Conversion of TOSCA node types.

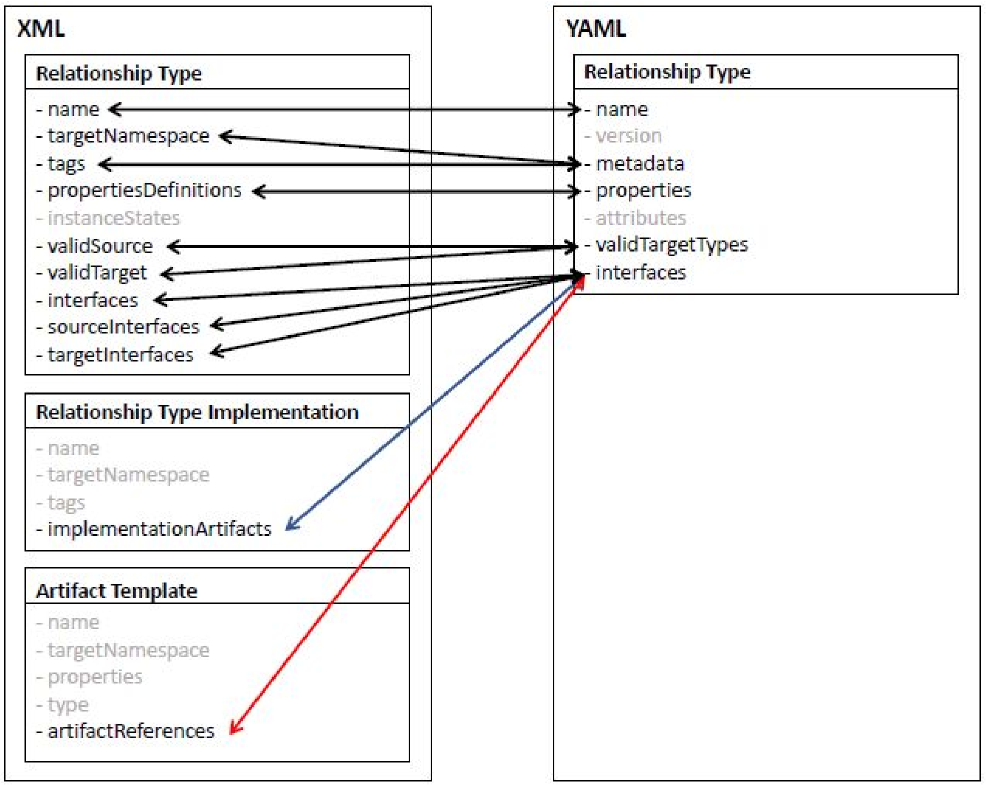


Figure 3. Conversion of TOSCA relationship types.

By extending the existing Model Converter component we can support any TOSCA YAML specific feature required to fulfill the requirements of the RADON project. Further, existing Winery components, such as the REST API over HTTP, are still functioning based on the TOSCA XML domain model and do not require any adaptations. This lets us focus on more important development tasks to provide an improved user experience in modeling TOSCA application using the new TOSCA Simple Profile standard.

3.4 Winery repository architecture

Part of RADON's DevOps methodology is to adopt infrastructure as code (IaC) as a core concept. This requires that Winery have to produce valid output compliant with the TOSCA Simple Profile (YAML) standard in version 1.3. Therefore, if a RADON user uses Winery to model a microservice application, including containerized and FaaS-based components together with respective data flows, the output will be serialized to YAML and stored in Winery's file-based datastore. From here, an advanced user can make use of the RADON IDE to adapt the model "as code" and push the changes to a Git remote to trigger a Git-based CI/CD workflow.

To achieve this, and to keep compatibility with the TOSCA XML standard, we provide a new repository implementation that can be used to (de)serialize Winery's internal TOSCA XML model into compliant TOSCA YAML. Therefore, we refactored Winery's repository implementation class hierarchy to fulfill the following goals:

- A user must be able to choose between a TOSCA XML and a TOSCA YAML based datastore by supplying a configuration at Winery startup.
- A user must be able to use Winery's "GitBasedRepository" implementation together with both datastore strategies.
- A user must be able to use multiple datastores in Winery, probably hosted on different Git remote locations.

To achieve the first goal, we refactored and combined the repository interface definition into one interface named "IRepository" (cf. Figure 4). It now contains all required operations to maintain TOSCA definitions. Further, we implemented an abstract class (FilebasedRepository) containing common logic that applies to all file-based datastores. Finally, we implemented to strategies to be used as file-based datastores: XmlBasedRepository and YamlBasedRepository. Following the best practices of the well-known strategy pattern, which enables the selection of a specific algorithm or implementation at runtime, we are now able to select the respective implementation based on a specific configuration parameter in Winery's core settings.

For the second goal, we refactored the existing "GitBasedRepository" to work with both strategies. Essentially, we applied the decorator pattern which is mainly used to add behavior to an individual object without affecting the behavior of other objects from the same class. Thus, the "GitBasedRepository" now have to be instantiated by referencing a respective file-based strategy.

From here, the “GitBasedRepository” decorates the initial implementation and enriches it with additional features regarding versioning with Git.

In the former version, Winery was capable to work only with a single datastore (or repository), meaning, type definitions and application blueprint are maintained in only one single repository. However, the last goal, to use multiple repositories that are stored separately and are even versioned on different Git remote locations, is fulfilled by the “MultiRepositorySupport” implementation. The “MultiRepositorySupport” class applied the composite pattern. In software engineering, the composite pattern describes a group of objects that are treated exactly like a single instance of the same object type. The purpose of a composite is to "compose" objects into a tree-like structure where the actual implementation of the composite pattern enables to treat individual objects and compositions in a consistent manner. By this extension, we can now use separate repository strategies, even versioned with Git, to use TOSCA’s normative type definitions, the RADON particles, and any user-provided repository to model and compose serverless FaaS and data-driven microservice applications using TOSCA’s Simple Profile standard in YAML.

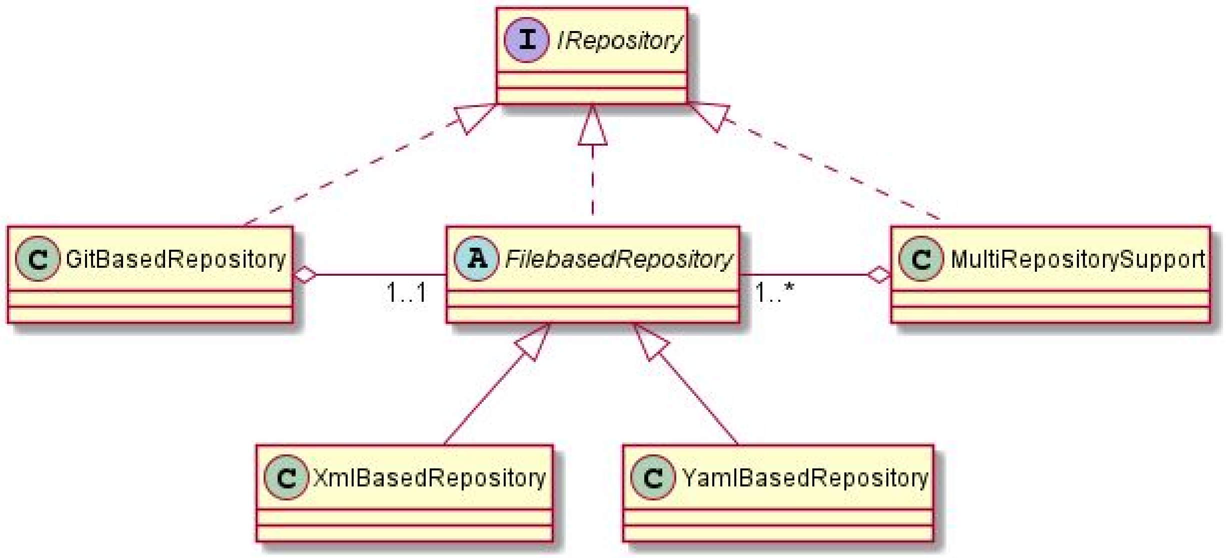


Figure 4. Repository hierarchy showing the new YAML-based strategy; together with the adaptations to use the strategies with the Git-based decorator and the multi-repository composite.

Since internal components in Winery, e.g., the REST API over HTTP, basing upon the TOSCA XML domain model, the implementation of the “XmlBasedRepository” remain simple. It just deals with the (de)serialization of objects to and from the actual location on the filesystem. Instead, the implementation of the “YamlBasedRepository” is more complex. More complex in the sense that the internal TOSCA XML domain model needs to be converted into the TOSCA YAML model before serializing the content to the filesystem; and vice versa when loading content from the filesystem. Here, the Model Converter comes into play employing the respective conversion rules, as elaborated in Section 3.2. The Model Converter is whenever a save or load request is made to the underlying repository. Figure 5 depicts how the Model Converter is used by the “YamlBasedRepository” implementation. Essentially, the Converter Component acts as an adapter between the TOSCA XML and TOSCA YAML domain model.

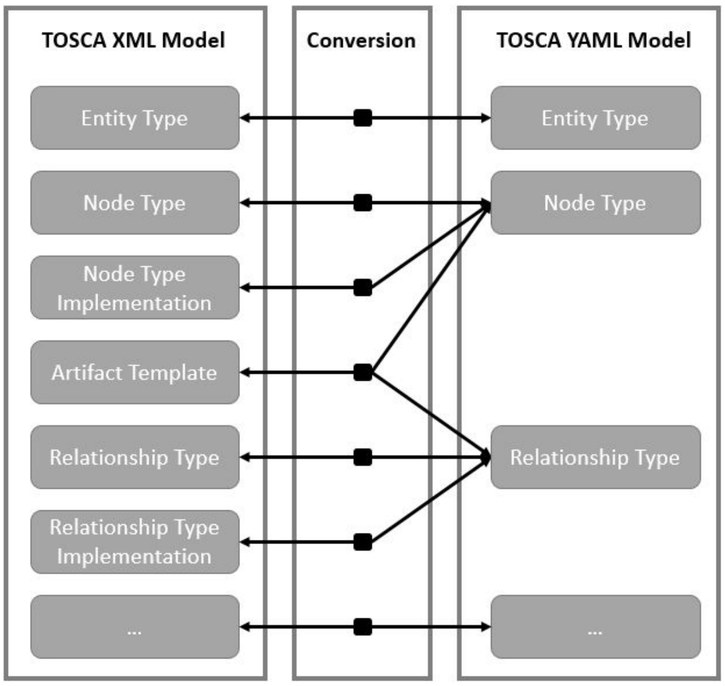


Figure 5. Model conversion in YamlBasedRepository implementation.

3.5 Final architecture

Resulting architecture of Eclipse Winery, which forms the baseline in RADON to model serverless FaaS and data-driven microservice applications, is depicted in Figure 6. An important aspect was that Winery has separate domain models, one for TOSCA XML and another one for TOSCA YAML. This has been introduced in former version of Winery to handle the import and export of XML and YAML based CSAR files. However, as elaborated in Appendix 1, the TOSCA YAML domain model was extended in order to support modeling features beyond importing and exporting.

Therefore, a new YAML-based repository was introduced. This implementation acts as a new repository strategy and uses the enhanced Model Converter component to (de)serialize respective YAML files and to translate it to the TOSCA XML domain model. Further, Model Converter becomes a dedicated component Winery as it is now prominently used inside Winery to support the conversion between the two domain models.

To switch between the repository strategy, Winery now comprises an extensible configuration component used to specify what filesystem provider (“yaml” or “xml”) shall be used. Depending on this setting, Winery is initialized respectively on startup. Further, this configuration is also used to specify if the underlying file-based repository shall be versioned with Git. If so, Winery is initialized with the Git-based repository decorator; decorating the respective file-based repository implementation. Further, in Winery's management UI a user can define to maintain multiple repositories and, afterwards, Winery uses the multi-repository implementation respectively. We showcase this feature in Section 4.

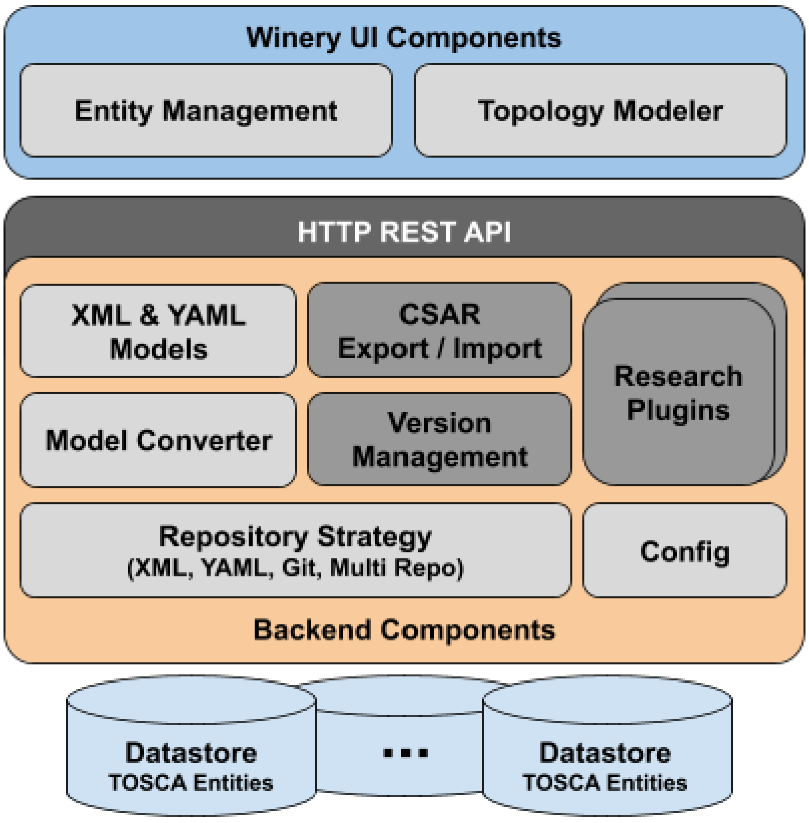


Figure 6. Final component architecture of Eclipse Winery used in RADON.

A major achievement of this deliverable is that we can fully reuse the existing TOSCA XML domain based for Winery's internal components. This means that existing and specialized research plugins as well as the REST API over HTTP didn't require adaptations and saved a lot of development effort, which resulted in more time to adapt Winery's frontend layer to enhance modeling support using TOSCA YAML.

Feature-based UI adaptations. Winery uses Angular as a framework for the frontend part. Angular is an open-source web application framework based on TypeScript introduced and maintained by Google. There is a huge open-source community providing lots of usable third-party components. As we made it possible to reuse the backend source code in most places, we have done the same for the frontend layer. Winery in its former versions already provides a very good baseline and user-experience when it comes to modeling an application based on TOSCA. However, many UI parts are specialized the modeling features of TOSCA XML. This means we developed additional UIs, extended and enhanced existing ones in order to tackle the modeling features of TOSCA YAML. The frontend components were already able to enable or disable certain UI features. Mainly, these feature flags were used in the past to activate special features developed to showcase research plugins in the course of UST's general research activities in the field of TOSCA. For RADON, and to fulfill the requirements towards a full-fledged Graphical Modeling Tool, we introduced a new feature flag indicating if the backend has been initialized with a YAML-based file repository. That gave us the opportunity to reuse existing Angular directives, pipes, and components to influence what UI section shall be visible or hidden. For example, this comes into play when working with TOSCA properties and relations. TOSCA properties in TOSCA YAML can only be key-value pairs having a default value, required flag, and description. Properties in TOSCA XML, in contrast, could have been expressed using arbitrary XML. Further, the modeling of relations between nodes has changed in TOSCA YAML compared to the older standard. To define a relation, respective TOSCA requirements and capabilities have to be defined beforehand. These kinds of modeling features are respectively enabled if Winery is started in the context of RADON. Moreover, in Section 3.2 we noted that a representation of a TOSCA node type has changed. This resulted in a rather simplified user-interface by hiding irrelevant parts such as the TOSCA node type implementation dialog.

Next, we show how the current prototype can be used to model a serverless FaaS application.

4 Prototype overview and usage

In this section, we elaborate on how to get the prototype running and demonstrate the process of modeling the RADON toy application example.

4.1 Build and start the prototype

At this stage of the project, it is recommended to build Winery from its sources as we add further enhancements as we go. However, there is also a published Docker image available that can be used to try-out Winery as GMT for RADON.

Checkout the source code:

```
git clone https://github.com/OpenTOSCA/winery.git
git checkout project/radon
```

Build the project using Docker:

```
docker build -t radon/modeling-tool .
```

Run the GMT using Docker:

```
docker run -it -p 8080:8080 \
              -e WINERY_REPOSITORY_PROVIDER=yaml \
              -v /var/workspace:/repository \
              radon/modeling-tool
```

Alternatively, you can start Winery from a pre-built Docker image:

```
docker run -it -p 8080:8080 \
              -e WINERY_REPOSITORY_PROVIDER=yaml \
              -v /var/workspace:/repository \
              opentosca/radon-modeling-tool
```

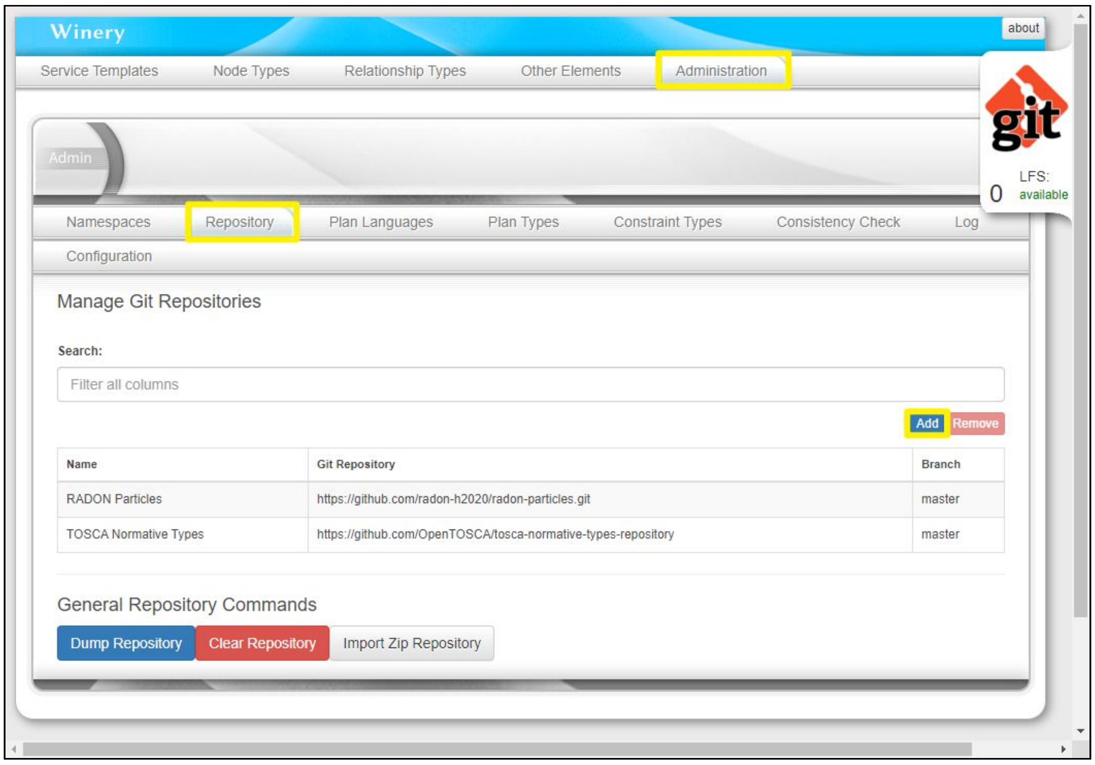
Afterwards, one could launch a browser window and navigate to:

```
http://localhost:8080
```

The prototype starts the screen showing current application blueprints (TOSCA service templates). From here, an application developer could start creating a new application blueprint based on the reusable types managed by Winery. However, to base an application on such reusable types, the GMT needs to be set up correctly, which we present next.

4.2. Set up multiple modeling repositories

As discussed in Section 3, supporting multiple distinct repositories is one of the features needed for enabling modeling applications with the model definitions stored in RADON particles repository. From the user's perspective, however, this feature is not intended to influence the modeling workflow significantly, i.e., configuring multiple repositories must rather be transparent for modelers. Therefore, interaction with this functionality is implemented as a part of Winery administration process. More specifically, Winery can be configured to use multiple distinct repositories in an *Administration* tab of Winery's management UI frontend component as shown in Figure 9. After the configuration is set up, modelers will be able to access and reuse the constructs stored in the specified repositories, e.g., RADON particles repository, TOSCA normative types repository, or any custom repositories. For example, a custom repository can comprise modified types inheriting from RADON particles or represent completely new, company-specific type definitions grouped separately. Figure 7 demonstrates Winery with two configured repositories, namely RADON particles and TOSCA normative types repositories. Both these repositories are Git-based meaning that a particular branch must be referenced at configuration time. Using the corresponding buttons, a repository can be added or deleted. Moreover, a sump of the chosen repository can be created by clicking the corresponding button. This dump can further be imported into Winery in case the exact same state of the repository is needed. As an additional option, using a respective button an existing repository can be emptied.



Manage Git Repositories

Name	Git Repository	Branch
RADON Particles	https://github.com/radon-h2020/radon-particles.git	master
TOSCA Normative Types	https://github.com/OpenTOSCA/tosca-normative-types-repository	master

General Repository Commands

Figure 7. Manage Git repositories having reusable modeling types.

4.3. Browse and adapt reusable modeling types

After the repositories support is configured, modelers are able to browse and reuse the modeling constructs available in the defined repositories. This interaction also happens by means of Winery's management UI, with all related TOSCA modeling constructs being available in a corresponding self-titled tab. For instance, Figure 8 demonstrates the tab listing all available node types. For more details, modelers are able to go into a detailed view window for every available modeling construct, e.g., node, relationship, or policy type, simply by clicking on the desired element. The detailed view window provides information relevant for the chosen element type, e.g., capability and requirement definitions will be shown for node and relationship types.

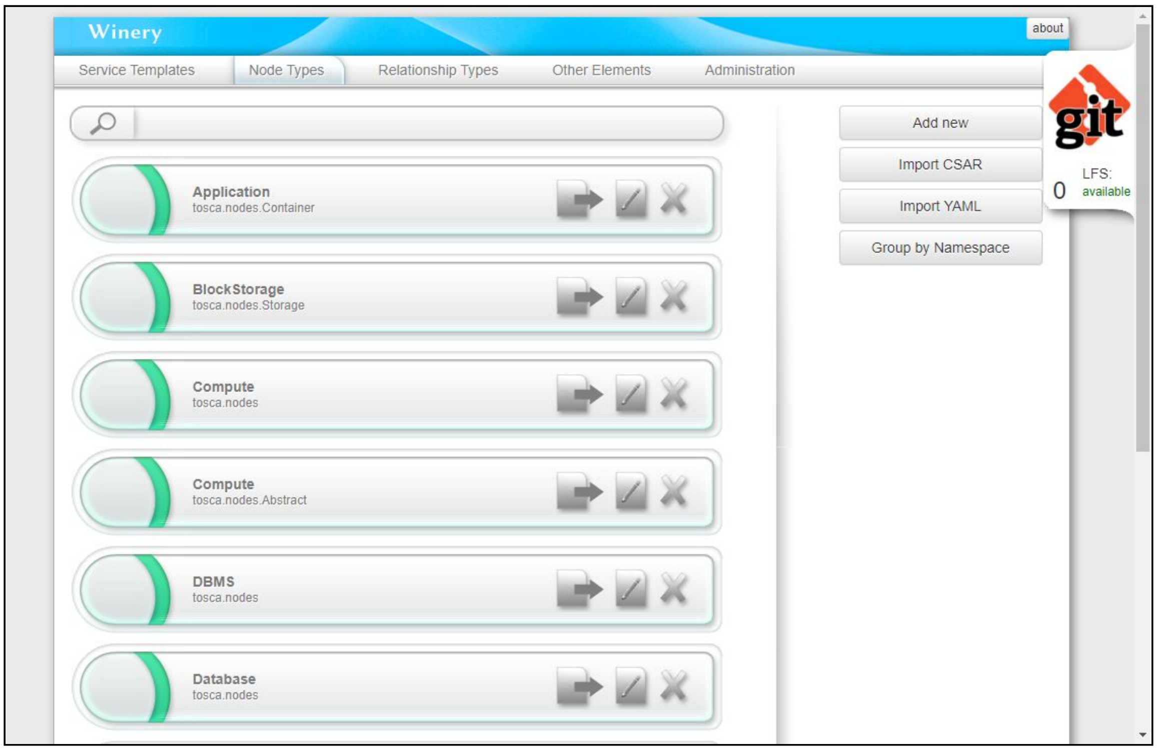
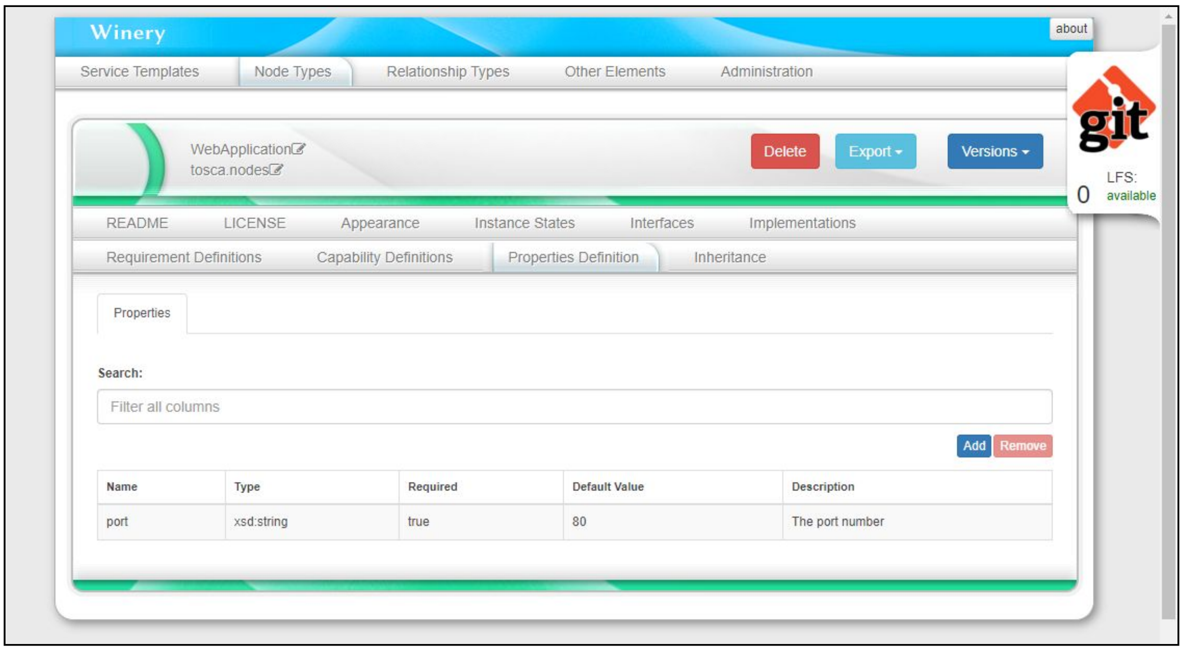


Figure 8. Reusable TOSCA node types.

For instance, Figure 9 shows the detailed view window for a WebApplication node type. More specifically, it depicts the currently defined properties for this type. The existing set of properties can be modified using the respective UI buttons for adding or removing properties. Apart from just modifying the properties of modeling constructs, it is also possible to export a chosen element as a CSAR, completely delete it, or control its versions. However, the shown WebApplication node type is deserialized from Winery's file-based repository and presented to adapt in a user-friendly manner. Whenever a user changes a type, e.g., adds properties or capabilities, Winery will save these changes to its data store compliant to the TOSCA standard. Listing 1, for example, shows the resulting YAML syntax that is produced by Winery.



Name	Type	Required	Default Value	Description
port	xsd:string	true	80	The port number

Figure 9. Definition of TOSCA fields using a unified user experience.

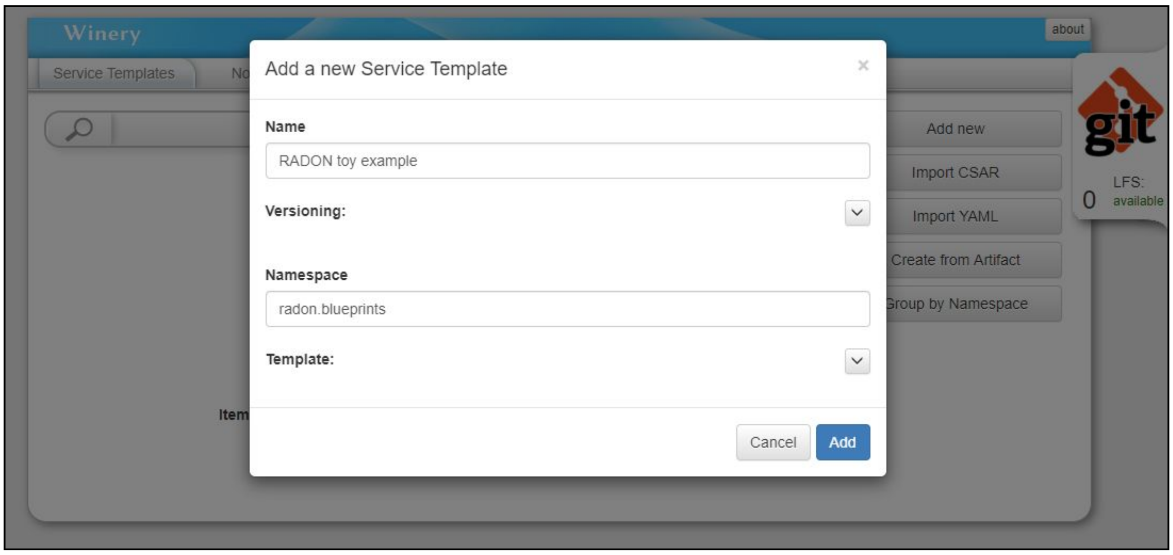
```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  tosca.nodes.WebApplication:
    derived_from: tosca.nodes.Root
    properties:
      port:
        type: string
        required: true
        default: 80
        description: The port number
    capabilities:
      app_endpoint:
        type: tosca.capabilities.Endpoint
    requirements:
      - host:
          capability: tosca.capabilities.Compute
          node: tosca.nodes.WebServer
          relationship: tosca.relationships.HostedOn
```

Listing 1. Resulting TOSCA YAML definition produced by Winery.

4.4 Create RADON toy example

In this section, we model the RADON toy example implemented using AWS, which generates thumbnails of images uploaded to an Amazon S3 bucket and stores the thumbnails on another S3 bucket. The application consists of three distinct components, namely a Lambda Function, a S3 Bucket, and the Amazon Platform itself. Each component is expressed as a separate TOSCA node type and is made available for modeling by the RADON particles repository.

An application developer can create a TOSCA service template to describe the cloud application and the involved components, or rather its topology. As depicted in Figure 10, the service template is created with a name and a specification of a namespace in which the blueprint should reside.



Name: RADON toy example

Versioning:

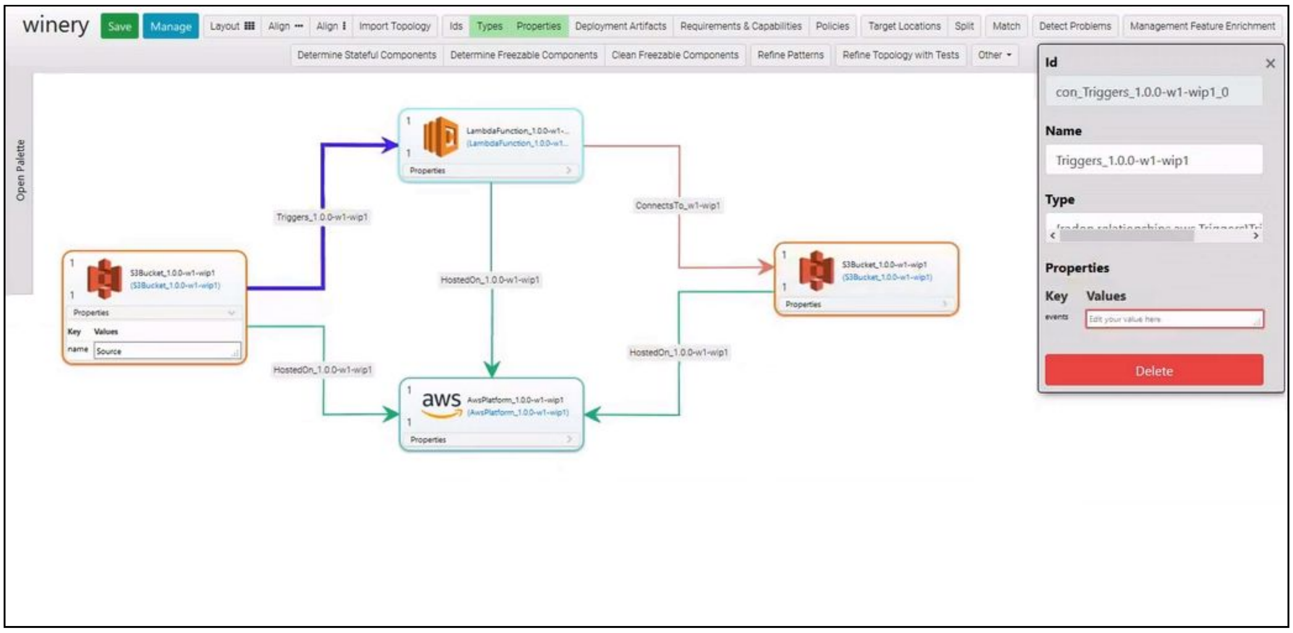
Namespace: radon.blueprints

Template:

Figure 10. Creation of a new RADON toy example blueprint.

To graphically model the topology, a user opens Winery's Topology Modeler. Users can drag-and-drop components from the palette on the right-hand side of the editor to model their intended application structure. To model the RADON toy example, we need an AWS Platform node which hosts a Lambda Function and two S3 buckets, one that hosts the source images and one as target bucket for the generated thumbnails. Figure 11 depicts the graphical structure of the RADON toy example. There the components are already connected using a respective relationship. For this application we used three distinct types of relations. One to express that the buckets and the function are hosted on AWS (cf. respective "HostedOn" relationships), another one to describe the behavior that the function connects to the target bucket (cf. "ConnectsTo" relationship between the function and the target bucket), and one "Triggers" relationship expressing the notion that the Lambda function is triggered by a S3 bucket event, which is emitted after new images were uploaded to the corresponding bucket. To model such relationships, a user can select the desired relationship and connect the component by clicking and dragging the relationship from the source component to the target component.

Using the Topology Modeler, a user also defines the properties of the respective components to describe them. For example, the S3 bucket on the left-hand side (cf. Figure 11) has a “name” property which reflects that this is the “source” bucket. Further properties can be set similarly, which are then used by the orchestrator to deploy the application.



Id

con_Triggers_1.0.0-w1-wip1_0

Name

Triggers_1.0.0-w1-wip1

Type

Properties

Key	Values
name	Source

Key	Values
events	Edit your value here

Figure 11. Modeling RADON toy example using Winery.

In terms of the event binding between the source bucket and the function, a user can specify the event related information as a property of the “Triggers” relationship. For example, a user could specify the respective S3 events that should trigger the function using AWS’ specific syntax. Finally, by clicking the “Save” button the service template will be stored in Winery’s repository. Like any other TOSCA entity, service templates are also stored in YAML format. This enables that a user can view and adapt the resulting blueprint using the RADON IDE and facilitates CI/CD by employing IaC.

5 Conclusion

In this document, we provide the implementation details of modifications in Eclipse Winery targeting the RADON project requirements. Firstly, to facilitate understanding of the introduced enhancements, the overview of Winery's relevant components and architectural decisions was presented. To accomplish the overall goal of modeling RADON applications in Winery, a set of subgoals has to be achieved, including (i) adding the support for TOSCA YAML specification, (ii) supporting the usage of RADON-specific modeling constructs, or so-called RADON particles, introduced in D4.3, and (iii) enhancing the graphical modeling workflow resulting from the usage of YAML-based specification. As a result, this document provides the descriptions of implementation details and architectural decisions made to accomplish these sub-goals. Additionally, the document provides an overview of how the modified version of Winery can be used to produce YAML-based models of RADON toy application example.

Next, we reflect on the requirements relevant to the GMT implementation in the scope of RADON. Table 2 shows an overview of the level of fulfillment for each of the agreed requirements. The labels specifying the "Level of compliance" are defined as follows:

(i) ✗ (unsupported): the requirement is not fulfilled by the current version,

(ii) ✔ (partially-low supported): a few of the aspects of the requirement is fulfilled by the current version,

(iii) ✔✔ (partially-high supported): most of the aspects of the requirement is fulfilled by the current version,

(iv) ✔✔✔ (fully supported): the requirement is fulfilled by the current version). Afterwards, we discuss for each requirement briefly how it has been addressed by publishing this deliverable.

Table 2. Achieved level of compliance to GMT's requirements.

Id	Requirement Title	Priority	Level of compliance
R-T4.3-1	Integration into IDE	Must have	✗
R-T4.3-2	Navigation to business logic	Must have	✗
R-T4.3-3	Navigation to deployment logic	Should have	✗
R-T4.3-4	Definition of constraints	Must have	✔
R-T4.3-5	Import existing models	Could have	✔
R-T4.3-6	Integration with other RADON tools	Should have	✗

R-T4.3-7	Present verification result	Should have	✗
R-T4.3-8	Test case specification	Must have	✔✔
R-T4.3-9	Modeling lots of elements	Must have	✔✔
R-T4.3-10	Group modeling elements	Could have	✔
R-T4.4-1	Export executable deployment model	Must have	✔✔
R-T4.4-2	Export of different deployment model formats	Could have	✔
R-T4.4-3	Import of model in different format	Could have	✗

At this stage, Winery supports modeling and attachment of policies, a TOSCA constructs which can be used to define non-functional requirements, also including constraint definitions required by R-T4.3-4. As described in Section 4, after modification of Winery it is now possible to model YAML-based policies representing the desired constraints. However, this feature has to be improved in the next iteration and synchronized with the latest updates in constraint definition language and verification tool. We are not fully compliant with R-T4.3-5 as this feature has to be tested more and improved where needed. However, import of existing models is possible either via manual manipulation with the repository or using the import feature, which still requires more testing and refinements. We are mostly compliant with the test cases specification required by R-T4.3-8. Similar to R-T4.3-4, test cases can be modeled using policies. In contrast to constraint definitions, RADON particles repository already offers a set of test-related policy types, which can be reused within Winery. To fully comply with this requirement, the test specification will be further refined and synchronized with the updated version of the continuous testing tool.

We are mostly compliant with R-T4.3-9, as it is possible to create complex application topologies in Winery with the RADON particles being imported into it. However, to fully comply with this requirement, modeling of microservice-based, grouped applications has to be improved together with the increase in the number of available RADON particles. We barely comply with the requirement R-T4.3-10, as support for grouping in models hs to be improved. We are mainly compliant with R-T4.4-1, as export of modeled application YAML-based TOSCA specification is available. To fully comply with the requirement, the export functionality has to be refined and tested more as well as synchronized with the updates from the RADON orchestrator. As described in Section 4, the employed conversion mechanism which allows exporting models in YAML and XML formats. We barely comply with R-T4.4-2, as the supported model formats are XML and YAML. In the next iteration, we plan to improve the quality of the exported models as well as to employ additional testing. Using an existing set of models available in particles repository, it is possible to model an initial version of RADON use cases. To fully comply with their requirements, the corresponding testing and synchronization functionalities have to be revised and refined.

Winery mainly complies with the requirements referring to basic modeling, as it is possible to graphically model application topologies and edit them using the set of available functionalities.

Future work. In the next project period we finalize and refine features based on the review validation results using RADON's envisioned demo lab applications. Specifically, we will work in the following areas to fulfill the remaining requirements:

1. **Integration with RADON IDE.** The ability to switch between the graphical representation and a code-based representation is yet to be implemented in Winery. This feature is going to be addressed in the next iteration of the deliverable.
2. **Integration of other RADON tools.** The integration with other tools can be achieved using Winery's pluggable architecture and available REST API. This activity is going to be guided for tools that are intended to be integrated with Winery; probably using non-TOSCA data models for integration.
3. **Usage of CDL and Verification Tool.** The usage workflow of CDL and verification tool is going to be finalized in the next iteration of the deliverable. This would also help fulfilling the corresponding requirements described previously.
4. **Data pipeline modeling.** Winery modeling workflow might require to be adapted based on the concepts introduced in the deliverable D5.5. The synchronization with these concepts is going to happen in the next iteration of the deliverable.
5. **Validation based on demo lab applications.** The scenarios described in the deliverable D6.1 intend to serve as an additional validation layer for RADON tools. Further validation of Winery using the example applications described in D6.1 is planned for the next iteration of the deliverable.
6. **Showcasing the modeling of industrial use cases.** Industrial use cases presented by responsible project participants are intended to serve as a final frontier for validating the introduced concepts. Modeling of given industrial application topologies using Winery and showcasing the supported features and integration with other tools, RADON IDE and RADON orchestrator in particular are planned for the next version of this deliverable.

References

[Kop01] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. „Service-Oriented Computing: 11th International Conference“. In: Springer Berlin Heidelberg, 2013. Kap. Winery – A Modeling Tool for TOSCA-Based Cloud Applications, S. 700–704. doi: 10.1007/978-3-642-45005-1_64

[Mor16] K. Morris, Infrastructure As Code: Managing Servers in the Cloud. “Oreilly & Associates Incorporated”, 2016. [Online]. Available: https://books.google.si/books?id=kOnurQEACAAJ

Appendix 1: GMT baseline implementation overview

As Eclipse Winery is chosen as a basis for GMT implementation, it is crucial to understand which existing functionalities are affected by RADON requirements and which new functionalities have to be implemented from scratch. To facilitate the understanding of taken design and implementation decisions, in this section we elaborate on the original architecture properties and feature sets of Eclipse Winery relevant in the scope of RADON project.

Eclipse Winery (Winery further in text) is a web-based environment for modeling cloud application topologies TOSCA cloud modeling language [Kop01]. Using Winery's management graphical user interface (GUI), modelers can specify different application components using corresponding TOSCA constructs such as nodes, relationships, policies, capabilities, and requirements. Subsequently, these components can be graphically assembled into service templates with the help of a topology modeling GUI. By switching between these GUIs, modelers are able to modify existing component definitions, introduce new versions, and export the final application models in a form of so-called Cloud Service Archives (CSAR). Moreover, the modeling process can be optimized by reusing a repository of predefined component definitions. Eclipse Winery is based on TOSCA XML specification v1.0. In the following, we describe the relevant details about Winery's component architecture.

A1.1 High-level architecture of Winery

Figure A1 depicts a simplified component architecture of Winery without naming irrelevant application components. Essentially, Winery provides the user interface layer that comprises Angular-based Management GUI and Topology Modeler applications. These components communicate with the business layer by means of a RESTful HTTP API. The backend which comprises business logic components is implemented using Java programming language.

As one of the most relevant features in the context of project requirements, Winery provides an extensible mechanism to implement supported repository types. In the original Winery version, file system is used as a repository type. All defined types and templates are stored in a repository folder which has a predefined format. For example, node types are grouped by their namespace and stored in a folder dedicated only for node types. Since TOSCA XML specification is used for modeling, all stored model files are in XML format, which makes it impossible to use Winery as-is. To support TOSCA modeling, backend provides separate sets of Java domain models representing both TOSCA XML v1 specification and YAML Simple Profile v1.1. The latter set of models is used for exporting into YAML format. However, this feature is experimental and cannot be used as-is since the produced YAML-based CSARs are not compliant with TOSCA Simple Profile specification because (i) multiple mappings between XML and YAML are absent, and (ii) several modeling constructs are not supported at all, e.g., modeling of attributes. Winery also provides version management component, which allows defining versions to modeled components.

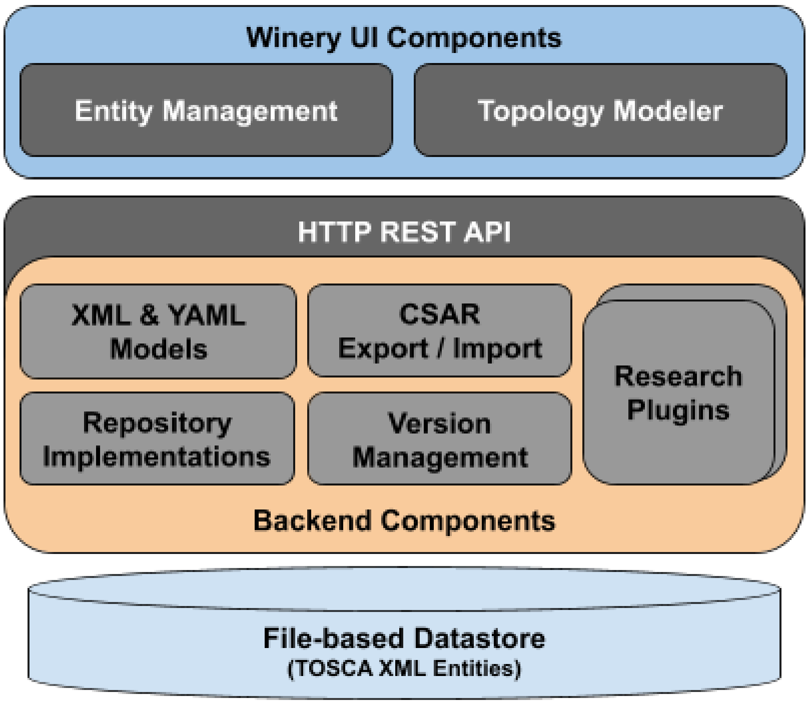


Figure A1. Simplified component architecture of Eclipse Winery.

Moreover, a dedicated component is responsible for packaging, exporting, and importing of CSARs from- and into the repository. In addition, Winery provides multiple components implementing various research concepts, e.g., application topology splitting or model compliance checking, which, however, are not directly related to RADON's requirements yet have to be taken into consideration for backward compatibility reasons. In the following we provide the details on some of the depicted components.

A1.2 File-based datastore

In the original version, Winery provides, essentially one kind of datastore (in the following also called "repository"): a file-based repository working with TOSCA XML v1.0 specification. As an addition, there is a repository implementation that extends the initial file-based version with Git support. This implementation facilitates the version control of defined models and artifacts, since the underlying repository becomes a Git repository and can be managed according to the standard Git workflow. Figure A2 demonstrates the class hierarchy in the original version of Winery. The overall initialization process then looks as follows: at startup, a file-based repository is initialized, which means that a local folder that is defined in Winery's configuration is treated as a TOSCA models repository of a predefined structure. The structure of the repository represent distinct sets of TOSCA constructs grouped by their namespaces, which are also used as a part of the REST resource names. More details about the architectural decisions in Winery can be found on Winery's web site[5].

[5] https://eclipse.github.io/winery/adr

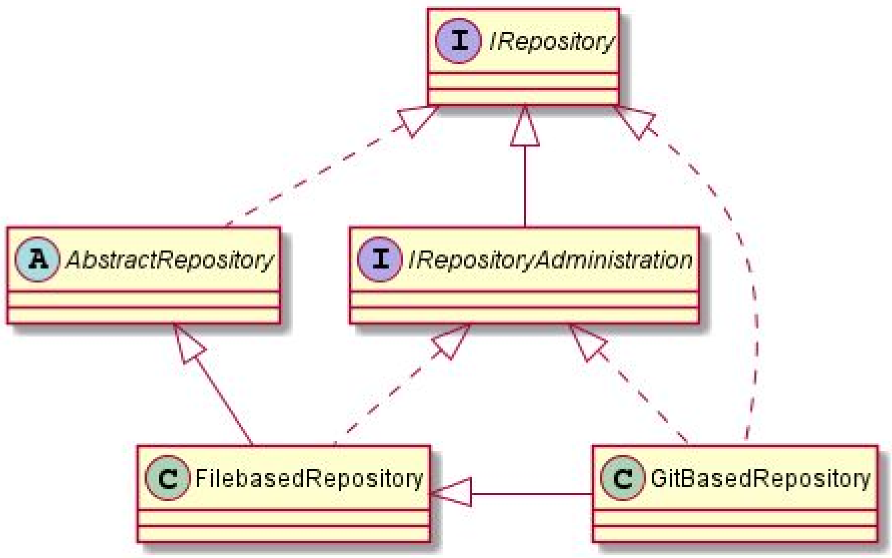


Figure A2. Repository implementation class hierarchy.

As the main interaction point with the repository, Winery's REST API is used. Apart from the frontend components, this API can be used for integrating Winery with other RADON tools. However, the interface is also coupled with TOSCA XML data models, which would require modifying serialization and deserialization of JSON data obtained from the frontend components, as well as internal utility classes responsible for processing model data.

A1.3 CSAR export and import

To support orchestration of modeled applications' deployment, TOSCA introduces the concept of CSAR, a self-contained archive that groups together all artifacts resulting from the modeling process, including component definitions, deployment scripts, and non-functional requirements. This archive has a predefined format, also including a meta file that lists all entities added to the archive. Winery allows exporting available models as CSARs as well as importing existing CSARs into the repository for modification and reuse. At export time, Winery traverses all linked component definitions and adds them to the archive to keep it self-contained, following the CSAR structure described in the TOSCA XML specification. In addition to the XML-based CSAR, Winery provides an option to export a YAML-based CSAR using the existing set of YAML models, which, however, are not executable and do not entirely comply with the standard. For import of CSARs, Winery analyzes the structure if the given archive and splits it into components, to store them in the repository separately. This decision was made to avoid redundancy, when multiple service templates using the same component types. While being a good baseline, these export and import functionalities cannot be reused as-is, since the conversion rules from XML to YAML are not always correct. As a first step, precise conversion rules have to be defined, and then the newly-implemented converters can be used inside existing importer/exporter components to extend them with proper YAML-based CSAR import/export functionalities.

A1.4 Versioning in Winery

Winery provides two mechanisms to version modeled application components. As introduced in Section 3.2, Winery's file-based datastore can be used together with Git to facilitate the version control of model entities. Whenever Winery's datastore is initialized as a Git repository, the web-based user interface can be used to visually show the detected changes on the filesystem. Further, the user interface supports to create a new Git commit to record the current detected changes as a new version. Having this, the produced files by Winery can be versioned and maintained in a DevOps-enabled environment using a Git-based workflow. It also enables collaboration with other teams and tools by synchronizing the datastore with a Git remote to track and record changes on a larger scale.

On top of that, Winery comprises an own versioning feature which can be used independently of the underlying datastore. All maintainable TOSCA entities can be individually versioned in a TOSCA compliant way by automatically adding a versioning suffix to the names of the respective entities. Winery uses three version identifiers to support different use cases and purposes: (i) a component version, (ii) a Winery version, and (iii) a work-in-progress (WIP) version. The resulting name of the entity is encoded using the following pattern whereas the version identifier uses a Debian-like naming scheme:

```
<name>_[<componentVersion>-]<wineryVersion>[-<wipVersion>]
```

A component version allows a user to optionally specify external versions for a TOSCA entity, e.g., to reference a specific version of a MySQL database. Following a component version, the Winery version defines the actual version of a given entity. For bug fixes or other advancements, the Winery version will be incremented. Further, to distinguish stable versions form unstable ones, a WIP identifier can be used. As this versioning style was implemented having the XML standard in mind, we will revise and adapt this to be used with YAML-based TOSCA models.