Provalets

OSGi-Based Prova Agents for Rule-Based Data Access

Adrian Paschke(✉)

Freie Universitaet Berlin, Berlin, Germany
paschke@inf.fu-berlin.de

Abstract. Rule-based Data Access (RBDA) has become an active R&D topic in the recent years. In this paper we propose an easy to use agent-based rule programming model and a general component architecture for RBDA. The programming model supports rapid prototyping and reuse of existing Prova rule agents/components which are published and managed in OntoMaven repositories. We name these components Provalets. We propose a declarative component description language that is powerful enough to represent different types of Provalets, including the representation of their functional interfaces and their semantics as well as their non-functional collaboration aspects and quality of service attributes.

1 Introduction

In this publication we propose the concept of ***Provalets***, specifically for use in the context of for Rule-based Data Access (RBDA). Provalets use ideas from (mobile) software agents, automated Maven repository and build management (OntoMaven [6,7]), trusted OSGi component runtime environments and cloud infrastructures to encapsulate data intensive routines and rule-based decision and reaction logic in Prova agents [9][1]. In this paper we describe the principles of Provalets (section 2), the declarative description language for Provalets (section 3) and the OSGi-based reference implementation of Provalets (section 4). We conclude our work in section 5.

2 Principles of Provalets

Provalets are (mobile) rule-based software agents [9] that act as inference component providing rule-based data access and rule-based data processing using Prova. Prova (Prolog + Java) is both a declarative rule-based programming language and a Java-based rule engine which can be used in Prova agents. Prova provides various built-ins for rule-based data access[2]. For instance, the following example defines a rule-based data access rule which selects with the SPARQL query built-in of Prova all luxury car manufacturers from DBpedia.

[1] http://prova.ws

[2] Prova has various built-ins for rule-based data access such as Java object access, file access, XML (DOM), SQL, RDF triples, XQuery, SPARQL.

C. Debruyne et al. (Eds.): OTM 2015 Conferences, LNCS 9415, pp. 519–526, 2015.
DOI: 10.1007/978-3-319-26148-5_35

```
luxuryCar(Manufacturer) :-
 sparql_connect(Connection, "http://dbpedia.org/sparql"),
 Query="SELECT ?manufacturer ?name ?car   % SPARQL RDF Query
        WHERE {?car <http://purl.org/dc/terms/subject>
         <http://dbpedia.org/resource/Category:Luxury_vehicles> .
            ?car dbo:manufacturer ?man .
            ?man foaf:name ?manufacturer.   } ORDER by ?manufacturer",
 sparql_select(Connection, Query, QueryID),
 sparql_results(QueryID, Manufacturer).
```

Provalets have a clear REST input and output interface, specifically an input URI and an output URI. They run in a controlled and secure runtime environment that guarantees that Provalets can only read data from input resources and write data to output resources. Provalets describe their functionality in terms of pre- and post-conditions on the sets of input and output data (section 3 defines the declarative interface description of Provalets).

Provalets describe themselves using the Linked Data principles: Each Provalet has a unique URI that is resolvable via HTTP. The description contains metadata about the Provalet including runtime dependencies and permissions required on the runtime platform as well as the description of the functionality it provides in the form of statements about pre- and post-conditions over the sets of input and output data.

2.1 Input and Output Interface of Provalets

Each Provalet is configured with one input URI that it is allowed to read from and one output URI that it is allowed to write to. The runtime environment of a Provalets should not allow the Provalet to cause any other side effects. The runtime environment controls which types of data and which KB sources of data are accessible by the Provalet.

This restriction of the Provalet side effects allows to describe their functionality in a more formal way than is possible with todays web services. Provalets can be described as algebraic operators on sets of RDF data. For example, a Provalet may describe the output data as a subset or superset of the input data, assertions borrowed from set theory. A Provalet may as well restrict itself to produce only specific kinds of data as semantically defined result sets. Provalets make no other assumption about the input and output sources other than they provide and accept data.

2.2 Permissions of Provalets

Provalets described permissions they require as metadata that is read by the runtime environment during deployment. By default Provalets are solely allowed to see the data which are directly served by the configured input URI. Provalets may define additional required permission. For example to access additional static URIs or crawl URIs that are visible in the set of input data. The sources of data may be restricted by subnets, domains, protocols or even types of data a Provalets is allowed to see. Provalet may provide HTTP access credentials to the input and output resources upon request.

Provalets must request permission to use additional computing resources on the machine they are executed. A Provalet may request harddisk space to store intermediate results. Other resources include memory, CPU time, account information, access to other web services. The latter can be used by Provalet to enforce license models through trusted providers. It is the task of the runtime container of a Provalet to grant required permissions and allow access to requested resources.

2.3 Runtime Environment of Provalets

Not only the the permissions for Provalets are enforced by the runtime environment. Essentially Provalets are shielded from any platform details. By default Provalets are not allowed to access any resource on the host system. For example Provalets should not even be allowed read information about the systems time or memory consumption. Basically Provalets should only be allowed to manipulate data without making any assumptions of the system nor to collect information about it.

Container Resource. The container resource acts like a knowledge base resource and describes itself with metadata in RDF. A user or agent receives information about a container resource by sending an HTTP request to the container URI. For example the container resource describes which permissions it can grant.

To use a container resource to execute a Provalet the user sends an HTTP request adding three parameters to the container URI: the Provalet URI, the input URI and the output URI. This way it is straight forward to execute a Provalet and lookup the results afterwards, by receiving the HTTP response of the output URI.

2.4 Lifecycle of Provalets

The life cycle of a Provalet is as follows:

1. Activation request: the Provalet's URI is passed as a parameter to a container resource together with the URI of the input and the output resource.
2. Reading the Provalet's description: the resource container makes an HTTP request to the URI of the Provalet and fetches data describing the it.
3. Permission enforcement: the runtime container is configured with the requested permissions for the Provalet. If the requirements do not match the containers restrictions the request is answered with HTTP status code 200.
4. Resolving Provalet dependencies: the fetched description contains metadata that allows to compute all runtime-dependencies of a Provalet. See section 4 for technical details of our reference implementation.
5. Downloading runtime dependencies: the container resource fetches all required libraries and constructs the classpath for executing it.

6. Provalet execution: the Provalet bundle is deployed in a secured local runtime environment and the Provalet bundle is started.
7. Provalet response: the user request is answered with an HTTP status code 200 (asynchronous container), or the agent is redirected to the output URI (synchronous container).
8. Writing the result set, the data that form the result are written to the output URI.
9. Finalizing Provalet, all additionally requested resources such as file and memory are released.
10. Undeploy, this step is optional, if configured so the runtime container may erase all libraries from the system. Which will require to download them for the next call.

Our reference implementation is written in Java and is based on web standards and open source libraries.

3 Provalet Description Language

Provalets are described by a semantic component description language that is powerful enough to represent their functional interfaces and their semantics as well as their non-functional collaboration aspects and quality of service attributes. These component descriptions are based on a plug-able semantic vocabulary to model the Provalets component descriptions in a platform-independent manner. The Provalet component description follows the classification of component contracts from Beugnards et al. [4] into four layers:

1. Basic syntactic Provalet component description layer expressing the Provalet artifact characteristics and functional interfaces.
2. Behavioural Provalet description specifying Provalet component semantics.
3. Synchronisation Provalet description describing dependencies between Provalets.
4. Quality of service and licensing Provalet description describing requirements with respect to response times, quality of results etc., as well as rights and obligations with respect to security, trust and licensing (e.g. metering and accounting).

A Provalet description contains

- A set of defined types (URIs) contributed by the Provalet.
- A set of defined properties (URIs) contributed by the Provalet.
- A set of defined constraint relationships (URIs) contributed by the Provalet.
- A function (URI Rest) to load a Provalet resource given a reference and a resource type.
- A function (URI Rest) that can be used to check the properties and relationships contributed by the Provalet component.

A Provalet component description has four parts:

1. In the Provalet component section the Provalet interfaces and necessary artifact characteristics are described, such as the groupId, artifactId, version and the optionally repository dependencies of the Provalet.
2. In the input data section the input resources of the Provalet are defined. The resources defined are constants identified by name and type. The types are defined in an (external) ontology and represented by URIs. This information can be used by the Provalet component to query the RDF data resources if needed.
3. In the output data section, the resources of the Provalet are defined. This is where a Provalet component provides (computed) data resources to be consumed by a consumer. These resources are also typed.
4. In the constraints part, the pre- and post conditions, rights and obligations are specified. The semantics supports the use of standard logical connectives such as AND, OR and XOR to define complex conditions. In addition to the conditions, value properties and existence conditions are supported as well.

Types, properties and relationships are defined in plugable ontologies. The resource types are defined using the web ontology language OWL or RDFS. The component semantics is described by relations between pre- and post-conditions of the Provalet method invocations. Test cases are used for this purpose by using JUnit as the Java standard. The Provalet descriptions provide test suites as part of specifications. Furthermore, Java annotations are used in order to stipulate in the description that a Provalet has to use annotations provided by additional components. Annotations are useful if the Provalet is to take advantage of injected services.

property	semantics	verification
usesAnnotation	classes use the annotations defined	JVM, ASTanalyser
implements	classes implement the interface or extend the abstract class	JVM
extends	classes extend other classes (transitive)	JVM
isVerifiedBy	a class is verified by a test suite	JUnit test runner
tests	a test suite provides tests for an abstract type	JVM

Rule-based conditions in Provalet descriptions can be either atomic or complex. To build complex conditions, the usual rule-based logical connectives with their standard semantics can be used. Three types of atomic conditions are supported: relationships between resources, resource properties, and conditions that a resource must exist. Based on these we can now define rules describing the validity constraints, rights and obligations of a Provalet. For instance, consider the following rule:

```
extends(?x, xp1 : ProvaletInstance) :-
 parser(?x : ProvaletInterface, ?c : JavaInstantiableClass),
 implements(?c, i : JavaAbstractType),
 tests(?s : TestSuite, i),
 isVerifiedBy(?c, ?s).
```

The Provalet component has to supply an implementation class of the Provalet interface that must pass a test suite. In order to run the tests, the test runner must instantiate the implementation class, and bind the variable in the test cases within the test suite to this instance. Then the test cases are executed.

4 Implementation

As execution environment for Provalets we applied a framework implementing the OSGI standards [5]. It allows us to dynamically install, update, and uninstall a Provalet and its dependencies, in a running system. Furthermore the OSGI specification provides a security model that is capable of offering a secure execution environment. Our current implementation deploys Apache Felix [1] as OSGi framework, but can be exchanged with another one as well.

Provalets themselves are Maven OSGi artifacts deployed in an OntoMaven artifact repository using OntoMaven [6,7] for the automated management. To automatically generate a new Provalet a Maven archetype can be used. An OntoMaven generated Provalet project provides all necessary dependencies and mechanism. The included *Provalet* class extends the *AbstractProvalet* from our ProvaletCore API and must be filled with the Provalet functionalities. The *ProvaletActivator* class extending the *AbstractProvaletActivator* serves as an OSGi entrance point to the Provalet. During the Provalet development the developer has to keep attention to only specify dependencies to APIs being OSGi capable. The artifact specification can be found in the Provalet description.

To execute the Provalet on a selected input resource the user needs to call the URI of a container resource (*containerURI*) via an HTTP GET request providing the URI of the Provalet (*ProvaletURI*), the input (*inputURI*) and the output URI (*outputURI*) as parameters:

```
<containerURI>?Provalet=<ProvaletURI>&input=<inputURI>&output=<outputURI>
```

The container resource is a special resource with an assigned OSGi environment for the execution of Provalets. It handles the Provalet call and answers the HTTP request with an HTTP response message. First it resolves the Provalet characteristics by calling the *ProvaletURI* and reading the Provalet description which also includes the necessary artifact characteristics (groupId, artifactId, version and optionally the repository). With OntoMaven's dependency resolution mechanism [3,7] the Provalet artifact and all its dependencies are resolved and downloaded to a local temporary repository using the integrated Aether library [11]. The downloaded artifacts are then deployed into the OSGi framework of the container resource and the RDF representation of the *inputURI* is passed as an object to the working method of the Provalet. Finally the container resource starts the installed Provalet bundle. After execution of the working method the Provalet passed its resulting data back to the container. The container checks the contend and enforces restrictions on the Provalet execution and the output and writes the RDF representation to the `outputURI`.

Once an instantiated Provalet exists, verification of the Provalet constraints and rules can be performed using Prova's inference mechanisms. The OSGi bundle classloader is used to load the resources and instantiate a Provalet instance as OSGi component with the translated Provalet rules and ontologies describing the Provalet conditions and constraints. Furthermore, OSGi features (see RFC 125 OSGi) are used to make Provalet licensing information part of the machine

readable component meta-data. This enables the Provalet execution applications to reason about this.

Finally, the container resource is responsible for uninstalling all bundles and optionally removing all the locally installed dependencies.

Two working modes of container resources are defined. Asynchronously working containers immediately response with an HTTP response code indicating that the Provalet working method was successfully started. The user of an asynchronously started Provalet has in principal two possibilities to work with the results: (1) an agent polls the output URI after a defined time and (2) the agent uses a subscription mechanism to be informed about updates in the output URI. In the synchronous working mode of a Provalet container the agent is redirected to the output URI once the results have been successfully written to the output URI. In this working mode the user can read the result immediately after receiving the HTTP response. However if the execution of the Provalet is taking too long the server may return with a timeout.

5 Conclusions

We conclude by summarizing the advantages of the Provalet concept.

- Provalets have functional properties with a logic/rule-based semantics. They are guaranteed to produce the same output for the same input.
- Provalets can be formally described with pre- and post conditions over input and output data. This allows formalizing their functionality.
- Provalets can be formally described by their functionality, which allows for automated test generation and automated service consumption. This guarantees their quality.
- Provalets are standardized OSGi components, which execute on a local desktop, in the cloud or on a database system without the need of changing the code or the configuration.
- Provalets cannot violate data privacy constraints. Because they can move to the data and are then not allowed to communicate to the outside.
- Provalets allow simple development and testing independently of the platform they execute on. Provalets support REST protocols.
- No configuration is necessary to execute a Provalet. All steps from selection, download, deployment and execution of a Provalet is packaged in one REST URL call.
- Provalets support reuse of mobile software agents and thereby reduce cost for developing and testing software.
- Provalets dynamically benefit from scaling-out resources. Provalets may spawn themselves to more nodes and split the workload by partitioning.
- Provalets dynamically benefit from scaling-up resources. By adding more resources to a server more Provalets can run on this machine. This reduces network traffic between Provalets.

Future work on the Provalet concept include the use of the new OMG API4KP standard [2,8] for the Provalet interface descriptions and the support of aspects in the automated Aspect OntoMaven [10] deployment of Provalets.

Acknowledgments. This work has been partially supported by the "InnoProfile-Corporate Smart Content" project funded by the German Federal Ministry of Education and Research (BMBF) and the BMBF Innovation Initiative for the New German Länder - Entrepreneurial Regions.

References

1. Apache Software Foundation. Apache felix project. http://felix.apache.org/
2. Athan, T., Bell, R., Kendall, E., Paschke, A., Sottara, D.: API4KP metamodel: a meta-api for heterogeneous knowledge platforms. In: Bassiliades, N., Gottlob, G., Sadri, F., Paschke, A., Roman, D. (eds.) RuleML 2015. LNCS, vol. 9202, pp. 144–160. Springer, Heidelberg (2015)
3. Athan, T., Schäfermeier, R., Paschke, A.: An algorithm for resolution of common logic (edition 2) importation implemented in ontomaven. In: Proceedings of the 8th International Workshop on Modular Ontologies Co-located with the 8th International Conference on Formal Ontology in Information Systems (FOIS 2014), September 22, 2014, Rio de Janeiro (2014)
4. Beugnard, A., Jézéquel, J.-M., Plouzeau, N.: Making components contract aware. IEEE Computer **32**(7), 38–45 (1999)
5. OSGI Alliance. OSGi Service Platform, Core Specification, Release 4, Version 4.2. Technical report, OSGI Alliance, September 2009
6. Paschke, A.: Ontomaven API4KB - a maven-based API for knowledge bases. In: Proceedings of the 6th International Workshop on Semantic Web Applications and Tools for Life Sciences, December 10, 2013, Edinburgh (2013)
7. Paschke, A.: Ontomaven: maven-based ontology development and management of distributed ontology repositories (2013). CoRR, abs/1309.7341
8. Paschke, A., Athan, T., Sottara, D., Kendall, E., Bell, R.: A representational analysis of the API4KP metamodel. In: Cuel, R., Young, R. (eds.) FOMI 2015. LNBIP, vol. 225, pp. 1–12. Springer, Heidelberg (2015)
9. Paschke, A., Boley, H.: Rule Responder: Rule-Based Agents for the Semantic-Pragmatic Web. International Journal on Artificial Intelligence Tools **20**(6), 1043–1081 (2011)
10. Paschke, A., Schäfermeier, R.: Aspect ontomaven - aspect-oriented ontology development and configuration with ontomaven (2015). CoRR, abs/1507.00212
11. Sonatype. Aether, June 2011. http://aether.sonatype.org/