# Self-Sustaining Platforms: a semantic workflow engine*

Sam Coppens, Ruben Verborgh, Erik Mannens, Rik Van de Walle

Ghent University – iMinds – Multimedia Lab
Gaston Crommenlaan 8 bus 201, B-9050 Ledeberg-Ghent, Belgium
firstname.lastname@ugent.be

**Abstract.** In this paper, we provide a novel semantic workflow system, based on semantic functional service descriptions and a rule file. The workflow engine follows a three-step process. First, it determines for all the resources in its knowledge base the functionality they need to progress in the workflow. This uses a phase-functionality rule file which binds phases of the workflow to functionalities. During a second phase, the functionalities are mapped to REST service calls using RESTdesc functional descriptions. During the third step, the engine executes the generated service calls and pushes the resource it acted on to the next phase in the workflow using a phase-transition rule file. The main advantage of this approach is that each step can be influenced by external information from the Linked Open Data cloud. It exploits the fact that Linked Open Data and RESTful Web services and APIs are resource-oriented. Moreover, the workflow rule file makes the system easily adaptable and extensible to achieve new functionalities or to obey changing company policies. Finally, the separation between functional descriptions and service descriptions supports easy management over the fast-changing services at hand.

## 1 Introduction

Today, applications on the Web increasingly rely on Linked Open Data [2] and RESTful services [3]. Both have a resource-oriented architecture that exploit the links between these resources. The increasing speed at which Linked Open Data, Web services and APIs are being deployed, demands an intelligent, expandable workflow engine grows in various domains, such as *factories* and *Smart Cities*. Process control in factories is often hard-coded into the control software of the production machines. Whenever they want to introduce a new sensor to steer the process, software needs to be adapted, recompiled, and re-deployed on every machine. An even bigger problem arises for Smart Cities, which use millions of wireless sensors. Existing workflow engines try to make this manageable, yet they are mostly manually created. Moreover, the intelligence of a platform is often distributed over the software code and the workflow engine, making the management of such a system a huge burden. Novel workflow engines should be able to integrate resources of any kind, i.e., Linked Open Data and RESTful Web services, to even become resource-agnostic. The only difference is that resources describing RESTful Web services and APIs get a functionality attached to them.

* The research activities as described in this paper were funded by Ghent University, iMinds, the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research-Flanders (FWO-Flanders), and the European Union.

In the solution we envision, the software intelligence and the workflow engine are *unified* into one system. Its first cornerstone is that workflows do not require manual composition anymore, since they are composed automatically using *semantic service descriptions* and *workflow description files* that captures the intelligence of the platform. By relying on functional service descriptions, services offering the same functionality become interchangeable. The declarative workflow rule files steer the reasoning engine by describing the whole process in terms of functionalities. This way, services become interchangeable and the platform becomes a loosely coupled system. Adaptations in the services or the workflow rule files are straightforward, without the need to adapt software code.

The second cornerstone of the envisioned platform is the *automated execution* of generated workflows. Together with the automatic workflow composition, the whole system becomes self-sustaining, easily adaptable and extensible, and highly portable. Workflows are dynamically recomposed when needed and automatically triggered when needed.

This unique combination enables our system to form the backend platform for a broad spectrum of applications, ranging from home domotic systems, long-term preservation platforms, process control software for factories to large Smart Cities. The only manageable assets are the service descriptions and the worfklow description rule files. At any moment new services can be described, which will automatically be incorporated into the newly generated workflows. Even the whole process can be changed, by making alterations to the workflow description rule files. All this happens in a declarative way, eliminating the need to redesign, recompile, and redeploy the software, keeping the system more manageable. These two characteristics, i.e., manageability of the services and speed-up of development process or extensibility of the system, are key to our solution and distinguish it from traditional workflow engines.

## 2 Concept and Architecture

When composing a workflow, our platform basically needs to following information, as shown in Figure 1:

**Knowledge Base** The knowledge base contains actually the information to be acted on. This is actually Linked Open Data, coming from a local triple store or optionally the Linked Open Data cloud.

**Phase-Functionality rule file** A phase-functionality rule file is a declarative file, stating how the different phases of a workflow are mapped to functionalities to move the resource to the next phase in the workflow. Rules are described in N3 [1], as will be discussed in Section 3.1.

**Phase-Transition rule file** This file is actually responsible for routing the different phases (and hence functionalities) of a workflow such that it complies to a certain overall functionality or company policy.

**Functional service descriptions** These service descriptions describe the individual services the workflow engine can make use of to compose its workflows. RESTdesc [10] describes services, as will be detailed in Section 3.2.

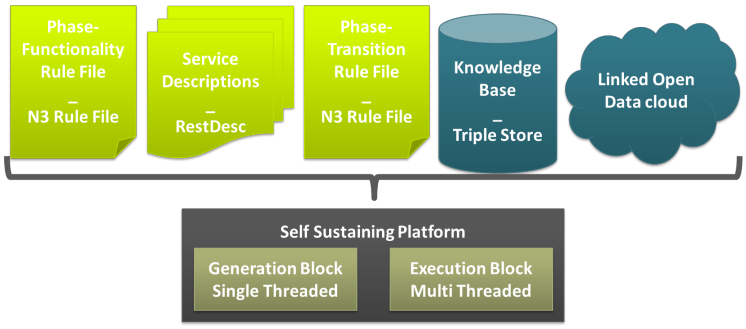


**Fig. 1.** Basic building blocks of the self-sustaining platform.

The phase-functionality rule file, the phase-transition rule file and the service descriptions in combination with the knowledge base, and optionally even external knowledge bases from the LOD cloud will generate the needed service calls by means of reasoning. This happens by the workflow generator. Because we rely on monotonic rule reasoning, this is a single threaded process. The generated service calls will be executed by the workflow executor. Of course, this can be a multi-threaded process.

The operation behind the self-sustaining platform actually consists out of three phases, as illustrated in Figure 2:

1. In the **functionality generation phase**, the phase-functionality rule file will tell for each resource which functionality it needs to get to the following phase of the workflow. This will happen by reasoning with the rule file over the knowledge base. This will entail triples describing a functionality the resource needs to get to the following phase of the workflow.
2. In the **service call generation phase**, these functionalities are mapped to service calls using the service descriptions. This is also done by reasoning over the entailed triples of the previous phase. The entailed triples of this reasoning cycle will describe service calls to be executed during the following phase. Splitting up the functionalities of service calls has a great advantage for the management of the services and generated workflows.
3. In the **service call execution phase**, the generated service call is being executed. If the service executed successfully, the phase-transition rule file is used to move to the next phase within the workflow.

Determining the needed transition is performed by means of reasoning over the entailed from the previous phase. The entailed triples of this phase will be stored to the knowledge base and a whole new reasoning cycle from the first phase is started. The next sections, will discuss the different steps in detail.

To demonstrate our idea, we will focus on the use case of publishing Linked Open Data. Publishing Linked Open Data typically consists of four steps: harvesting the data, mapping the harvested data, reconciling the mapped data, enriching the reconciled data, and finally, publishing the enriched data.

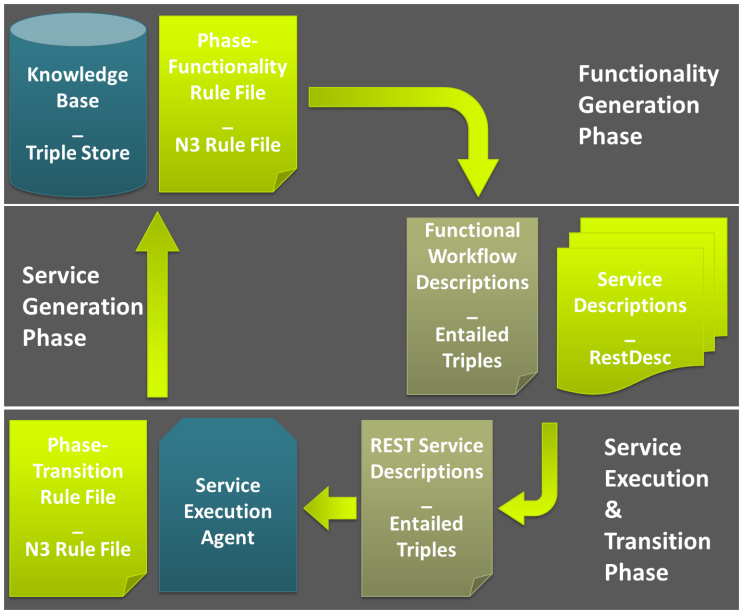


**Fig. 2.** Schematic operation of the self-sustaining platform.

## 3 Operational Phases

### 3.1 Functionality Generation Phase

This phase-functionality rule file will split up the workflow into different phases and it will list for each phase the functionality it needs to get to the following phase. If we take back the example of publishing Linked Open Data, then its workflow consists of the following ordered list of functionalities: harvesting, mapping, reconciliation and publication. Each of the functionalities give rise to a new phase. An example is schematically shown in Figure 3. Every phase is characterised with a functionality to complete the phase.


**Fig. 3.** Example workflow for publishing LOD.

The phases, and the functionalities connecting them, form a functional workflow description. This is a composition of functionalities that need to be performed in order to fulfil a certain policy or achieve a certain goal. Such a goal or policy can be described using a rule file and a rule-based reasoner. In this rule file, the phases and functionalities need to denoted with URIs. Taking back our example of LOD publisher, the rule file looks like the following.

```
@prefix wf: <http://example.org/workflow#>.
@prefix fn: <http://example.org/functionalities#>.

{?object0 wf:hasStarted wf:phase_0} =>
                          {?object0 fn:harvestRecord ?object1.
                           ?object1 wf:hasCompleted wf:phase_0}.

{?object1 wf:hasStarted wf:phase_1} =>
                          {?object1 fn:map ?object2.
                           ?object2 wf:hasCompleted wf:phase_1}.

{?object2 wf:hasStarted wf:phase_2} =>
                          {?object2 fn:reconcile ?object3.
                           ?object3 wf:hasCompleted wf:phase_2}.

{?object3 wf:hasStarted wf:phase_3} =>
                          {?object3 fn:enrich ?object4.
                           ?object4 wf:hasCompleted wf:phase_3}.

{?object4 wf:hasStarted wf:phase_4} =>
                          {?object4 fn:publish ?object5.
                           ?object5 wf:hasCompleted wf:phase_4}.
```

This phase-functionality rule file actually describes the different phases of a resource in a workflow and for each phase its functionality. It doesn't tell how the different phases should be coupled in order to achieve a certain workflow. These transitions are described in the phase-transition rule file, discussed in Section 3.3. During the second step, this functional workflow, actually described as entailed triples, is being materialised to explicit service calls. Assume we have an URL of a resource we want to publish as LOD. e.g., `http://foo.org/resource/1234`. If we add the following triple to our triple store/knowledge base, we get the publishing procedure starting:

```
<http://foo.org/resource/1234> wf:hasStarted wf:phase_0.
```

Reasoning over this triple with our example rule file yields the following triples:

```
<http://foo.org/resource/1234> fn:harvestRecord ?object1.
?object1 wf:hasCompleted wf:phase_0.
```

These triples are passed to the next step, where functionalities are translated into REST service calls. During this step another reasoning cycle will happen. All the entailed triples will be added to the knowledge base after the successful execution of this phase of the workflow during the last step. After this step, a new reasoning cycle is fired. Another reasoning cycle over these triples would yield the following triples, which in turn are passed to the next step:

```
?object1 fn:map ?mappedResult.
?mappedResult wf:hasCompleted wf:phase_1.
```

Eventually, after several reasoning cycles, the entire workflow will be executed.

### 3.2 Service Generation Phase

As explained, this step is going to turn the result of the functionality generation phase into a service call description, which will be fed to the service execution and transition phase. The result of the functionality generation phase is actually a set of entailed triples, as shown in the previous section. For each phase in the workflow, we have a triple denoting the functionality it needs for the next phase. These triples should actually result in a description of the of a REST service, which serves the functionality.

For the service description, we rely on RESTdesc. RESTdesc is both a description and a discovery method targeting restful Web services, with an explicit focus on functionality. It consists of well-established technologies such as HTTP and RDF/Notation3 and is built upon the concepts of hyperlinks and Linked Data. Its goal is to complement the Linked Data vision, which focuses on static data, with an extension towards Web services that focus on dynamic data. All RESTdesc descriptions are:

- *self-describing*: using Notation3 semantics;
- *functional*: explaining exactly what the operation does;
- *simple*: descriptions are expressed directly using domain vocabularies.

Since RESTdesc entails the operational semantics of Notation3, it allows for versatile discovery methods. We can indeed use the power of Notation3 reasoners to determine whether a service satisfies at set of conditions. Even more advanced reasoning is possible to decide on service matching, and to create complex compositions of different services. We see this as an important prerequisite for services in order for them to contribute to the future Web of Agents, since new functionality can only be obtained by on-demand compositions tailored to a specific problem. Such a RESTdesc description of a service that harvests a record, looks like this:

```
@prefix fn: <http://example.org/functionalities#>.
@prefix http: <http://www.w3.org/2011/http#>.

{ ?url fn:harvestRecord ?record. }
=>
{ _:request http:methodName "GET";
            http:requestURI ?url;
            http:resp [ http:body ?record ].}.
```

It actually says: If you have a URL from which you want to harvest a metadata record, then you can do this by an http GET request on the URL and the response will contain the record in its body. POST requests are also described in this way. The following mapping service is a good example of a POST request.

```
{ ?record fn:map ?mappedRecord.}
=>
{ _:request http:methodName "POST";
            http:requestURI "http://example.org/service/map/";
            http:body ?record;
            http:resp [ http:body ?mappedRecord ].}.
```

Thus, during this second step, we only need to reason with the RESTdesc service descriptions over the entailed triples of the first step in order to get the service call descriptions that need to be executed during the next step. If we take back the example of Section 3.1, the entailed triples from the first reasoning cycle being feed to this step are:

```
<http://foo.org/resource/1234> fn:harvestRecord ?result.
?result wf:hasCompleted wf:phase_0.
```

Reasoning over these triples with the service description of the harvest service, inferences the following triples that are fed to the workflow execution step:

```
_:request http:methodName "GET";
          http:requestURI <http://foo.org/resource/1234>;
          http:resp [ http:body ?record ].
```

If you have in your repository with RESTdesc service descriptions multiple services covering the same functionality, this phase would yield multiple REST service calls being described. Not all these service descriptions need to be fed to the service execution phase, only one needs to. For this reason, this phase stops after the first RESTdesc service description that entails a service call description. This separation of functionalities and services that cover these functionalities greatly enhances the management of the services and the workflows. It makes services with the same functionality interchangeable. Of course, this service selection step can also be influenced, as will be discussed in Section 4.3.

### 3.3 Service Execution and Transition Phase

In this step, the entailed triples of the previous step are processed. The entailed triples of the previous step actually describe a REST service call, explaining the HTTP GET request can be done on the URL `http://foo.org/resource/1234` and the record's metadata will be returned into the response's body. Thus, an execution agent can process these entailed triples, describing a service call. If the agent executes the service call successfully, the phase transition rule file is used to progress to the next phase of the workflow. This phase-transition file actually routes all the different phases of the workflow. This is done by reasoning over the entailed triples of the previous steps. For our LOD publication example, this phase transition file looks like this:

```
@prefix wf: <http://example.org/workflow#>.
@prefix fn: <http://example.org/functionalities#>.
{?object wf:hasCompleted wf:phase_0} =>
                         {?object0 wf:hasStarted wf:phase_1}.

{?object wf:hasCompleted wf:phase_1} =>
                         {?object0 wf:hasStarted wf:phase_2}.

{?object wf:hasCompleted wf:phase_2} =>
                         {?object0 wf:hasStarted wf:phase_3}.

{?object wf:hasCompleted wf:phase_3} =>
                         {?object0 wf:hasStarted wf:phase_4}.
```

For our example, these entailed triples fed to this step are:

```
<http://foo.org/resource/1234> wf:hasCompleted wf:phase_0.
  _:request http:methodName "GET";
            http:requestURI <http://foo.org/resource/1234>;
            http:resp [ http:body ?record ].
```

The service-executing agent detects the service call description and executes it. If the execution was successful, the reasoning with the phase-transition file is started, yielding the following triples:

```
<http://foo.org/resource/1234> wf:hasStarted wf:phase_1.
```

After this last reasoning cycle, all the entailed triples of the three steps are being ingested into the knowledge base and a new functionality generation phase is started, as shown in Figure 2. If the execution of the service call was unsuccessful, the operation breaks up, storing nothing to the knowledge base, because you cannot state the phase has already finished. Thus, the triples being added to the knowledge base are:

```
<http://foo.org/resource/1234> wf:hasCompleted wf:phase_0.
_:request http:methodName "GET";
            http:requestURI <http://foo.org/resource/1234>;
            http:resp [ http:body ?record ].
<http://foo.org/resource/1234> wf:hasStarted wf:phase_1.
```

## 4 Advanced Features

In this section, we are going to discuss some advanced features of the workflow engine. Until now, only the basic operation has been discussed. By adapting the phase-functionality, file, the phase-transition file and the service descriptions, some more advanced features are possible in this framework. The advanced features discussed in this section are: how to integrate feedback loops, cron jobs, and nested functionalities. The advanced features are not limited to these ones. Other features are, e.g., the possibility to include Linked Open Data into the workflow, such that workflow are adapted or triggered by external data from the Linked Open Data cloud. SPARQL endpoint results can be integrated into the workflows. For this, SPARQL requests are modeled as RESTful service calls. The results from these endpoints can then be used within the rule files make the workflows adapting to this external data. Another feature that can be integrated into the workflow engine is basic workload balancing. For this, we can include counters attached on the service endpoints, such that the load for a functionality is shared across the different services, serving the same functionality. A last feature to mention is that it becomes easy to integrate authorisation into the workflows. To integrate this in the workflows, a new phase must be integrated into the workflow, e.g., in our example, for validating the generated enrichments. The transition from this phase is then actually steered by an external REST service, which is triggered by the authority. In the remainder of this section, we will discuss in detail feedback loops with a cronjob (time-based task), nested functionalities, and how to influence the service selection.

### 4.1 Feedback Loops

Feedback loops can be built into the workflow. In our example, we can introduce a update process by introducing a recurring reharvest of the record. This is achieved by adapting the first rule and adding a rule to the phase-functionality rule file:

```
@prefix dc: <http://purl.org/dc/elements/1.1/>.
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix time: <http://www.w3.org/2000/10/swap/time#>.

{?object0 wf:hasStarted wf:phase_0} =>
                 {?object0 fn:harvestRecord ?object1.
                  ?object1 wf:hasCompleted wf:phase_0.
                  ?object1 dc:source ?object0.
                  ?object1 dcterms:modified time:localTime.}.

{?object5 wf:hasStarted wf:phase_5.} =>
                 {?object5 dc:source ?url.
                  ?url fn:reHarvestRecord ?object6.
                  ?object6 wf:hasCompleted wf:phase_5.}.
```

With these rules, we define an extra phase for the workflow, i.e., `wf:phase:5`, which is coupled to the functionality `fn:reHarvestRecord`. Next, this phase needs to be integrated into the phase-transition rule file. The phase-transition file will get the following extra rule:

```
@prefix func: <http://www.w3.org/2007/rif-builtin-function#>
@prefix pred: <http://www.w3.org/2007/rif-builtin-predicate#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

{?object wf:hasCompleted wf:phase_4;
         dcterms:modified ?lastHarvest.
 (time:localTime ?lastHarvest) func:subtract-dateTimes ?difference.
 (?difference "P14D"^^xsd:dayTimeDuration)
    pred:dayTimeDuration-greater-than true.} =>
                  {?object wf:hasStarted wf:phase_5}.

{?object wf:hasCompleted wf:phase_4} =>
                  {?object wf:hasStarted wf:phase_1}.
```

### 4.2 Nesting

The functionalities can be nested using multiple rules. In our previous example, the reharvest functionality consists actually of a new delete and an already existing harvest functionality. The following rules for the phase-functionality rule file shows reharvesting:

```
{?object fn:reHarvestRecord ?result} =>
                    {?object wf:hasStarted wf:phase_6.}.

{?object wf:hasStarted wf:phase_6} =>
                    {?object fn:delete ?delete.
                     ?object wf:hasCompleted wf:phase_6}
```

And the following rules are added to the phase-transition file:

```
{?object wf:hasCompleted wf:phase_6} =>
                    {?object wf:hasStarted wf:phase_0.
                     ?object wf:isReharvest true.}

{?object wf:hasCompleted wf:phase_0} =>
                    {?object wf:hasStarted wf:phase_1.}

{?object wf:isReharvest true.
 ?object wf:hasCompleted wf:phase_0} =>
                    {?object wf:hasCompleted wf:phase_5.}
```

This needs a little explanation. There is no service providing the reharvest functionality, there are services for the delete and harvest functionality. Whenever our semantic workflow engine cannot trigger a rule with the entailed triples from the functionality generation phase during the service generation phase, these entailed triples are taken to the service execution and transition phase. This reasoning cycle, only defines phase transitions without delivering a service call description. During the subsequent reasoning cycle, our newly defined rules, bringing our object from the deadlocked reharvest state to the delete state for which it has a service description. In turn, the delete phase will give rise eventually to the harvest phase and this is how functionalities can be nested in our platform. After this harvest, the nested loop needs to be closed. This is done with the last rule from the phase-transition file, which says that if the harvest is finished and is part of a reharvest operation, the reharvest operation is finished.

### 4.3 Service Selection

In Section 3.2, we described what happens if several service descriptions fulfil the same needed functionality. Via the workflow rule file this service generation phase can be influenced to obey certain rules for selecting the appropriate service. A use case for this would the selection of the right mapping service. One can image having several mapping services in place, each covering other mapping formats as input and output. Then one can have the following rules in the workflow rule file to select the appropriate mapping service during this service generation phase:

```
{?object wf:hasStarted wf:phase_0} =>
    {?object fn:harvestRecord ?result.
     ?result wf:hasCompleted wf:phase_0.
     ?result ex:vocabulary <http://www.loc.gov/MARC21/slim>}.

{?object wf:hasStarted wf:phase_1} =>
    {?object fn:map ?result.
     ?result wf:hasCompleted wf:phase_1.
     ?result ex:vocabulary <http://purl.org/dc/terms/>}.
```

In this workflow rule file, we specified in the first rule the vocabulary used to describe the record harvested, i.e., MARC XML. In the second rule, we specify that everything needs to be mapped to the vocabulary dcterms. In combination with specifying the mapping vocabularies, you can steer the selection of the services used for a functionality:

```
{ ?record ex:vocabulary <http://www.loc.gov/MARC21/slim>.
  ?record fn:map ?mappedRecord.
  ?mappedRecord ex:vocabulary <http://purl.org/dc/terms/>.}
=>
{ _:request http:methodName "POST";
            http:requestURI "http://example.org/service/map";
            http:body ?record;
            http:resp [ http:body ?mappedRecord ].}.
```

This way service descriptions can be manipulated to also include, e.g., a trust rating or a quality rating. The workflow rule file can then be adapted to take always the most trustworthy service or the most qualitative one.

## 5 Related Work

Our self-sustaining platform relies on two basic technologies: Semantic service descriptions and automatic workflow composition. "Semantic Web Services: a RESTful approach" [4] has a RESTful grounding ontology that maps OWL-S [8] and WADL [6]. OWL-S is a W3C submission for Semantic Web service descriptions. Web Application Description Language (WADL) is a W3C submission for describing REST over HTTP Web services. The Semantic Automated Discovery and Integration (SADI [11]) offer Web service Design-Patterns, an API and a reference Implementation that simplify the publication of services such that they can easily be discovered and integrated.

When it come to automatic workflow composition, Data-Fu, a language and interpreter for interaction with read/write Linked Data, is related to our research. Data-Fu is a declarative rule language for specifying interactions between web resources. Another platform, supporting automatic workflow composition is the work of Krummenacher [7]. In this work, he investigates the composition of RESTful resources in a process space. The work is based on resources described by graph patterns. Similar work is carried out by Speiser and Harth [9], which also relies on graph patterns for RESTful Linked Data Services. In this context, it is also worth mentioning WINGS, (Workflow INstance Generation and Specialization [5]). WINGS is a semantic workflow system that consists of the WINGS Portal, which is its user interface, and the WINGS Semantic Workflow Reasoner, which contains its constraint reasoning and propagation algorithms.

For our workflow composition, we rely on rule-based reasoning. All the intelligence of an application is gathered by three rule files. This centralisation of the application's intelligence allows easy management of the services, because services become interchangeable if they provide the same functionality. At the same time, it allows easy adaptation and extensibility of the application through the adaptation of the three rule files. There is no need anymore to recompile, and redeploy the software.

## 6 Conclusions

In this paper, we presented a novel workflow engine, based on Linked Data, RESTful Web services and rule-based reasoning. Our platform will generate workflows described in terms of phases and functionalities connecting the phases.

In a later phase, these functionalities are mapped into service call descriptions, which are finally executed. A main feature of our platform is that it is easily adaptable and expandable. The whole system captures the application intelligence through three N3 rules files. One for describing the different phases of a workflow and for each phase the functionality it needs. Another describing the phase transitions, and finally one describing the RESTful Web services the platform relies on. These files can be adapted at runtime, but at the same time aggregate the platform's intelligence. Another main feature is the explicit separation between functionalities and services. By this, services serving the same functionality become interchangeable. This makes both the management of the workflows and the services a lot easier. Our generated workflows can be steered by external Linked Data, so that the platform can act on this external data. A last feature of the platform is its ability for error handling. These features together make the platform self-sustaining.

## References

1. Berners-Lee, T. and Connolly, D. Notation 3, 2006. Available at `http://www.w3.org/DesignIssues/Notation3`.
2. Bizer, C.;Heath, T. and Berners-Lee, T. Linked Data – the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.
3. Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 407–416, New York, NY, USA, 2000. ACM.
4. Otávio Freitas Ferreira Filho and Maria Alice Grigas Varella Ferreira. Semantic Web Services: A RESTful Approach. In *IADIS International Conference WWWInternet 2009*, pages 169–180. IADIS, 2009.
5. Yolanda Gil, Varun Ratnakar, Jihie Kim, Pedro A. González-Calero, Paul T. Groth, Joshua Moody, and Ewa Deelman. Wings: Intelligent workflow-based design of computational experiments. In *IEEE Intelligent Systems*, 2011.
6. Marc J. Hadley. Web application description language (wadl). Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2006.
7. Reto Krummenacher, Barry Norton, and Adrian Marte. Towards linked open services and processes. In Arne-Jørgen Berre, Asunción Gómez-Pérez, Kurt Tutschku, and Dieter Fensel, editors, *FIS*, volume 6369 of *Lecture Notes in Computer Science*, pages 68–77. Springer, 2010.
8. David Martin, Mark Burstein, Erry Hobbs, Ora Lassila, Drew Mcdermott, Sheila McIlraith, Srini Narayanan, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic markup for web services. Technical report, W3C, November 2004.
9. Sebastian Speiser and Andreas Harth. Integrating linked data and services with linked data services. In *ESWC (1)*, volume 6643 of *Lecture Notes in Computer Science*, pages 170–184. Springer, 2011.
10. Ruben Verborgh, Thomas Steiner, Davy Van Deursen, Jos De Roo, Rik Van de Walle, and Joaquim Gabarró Vallés. Capturing the functionality of Web services with functional descriptions. *Multimedia Tools and Applications*, (2):365–387, 2013.
11. Mark Wilkinson, Benjamin Vandervalk, and Luke McCarthy. The semantic automated discovery and integration (sadi) web service design-pattern, api and reference implementation. *Journal of Biomedical Semantics*, 2(1):8, 2011.