Volume 1.0

Project Deliverable 4.1

Functional workflow composer

Worker-Centric Workplaces in Smart Factories www.facts4workers.eu

FACTS4WORKERS

Factories of the Future
Public Private Partnership

Series: Heading
Published by: FACTS4WORKERS: Worker-Centric Workplaces in Smart Factories.
FoF 2014/636778

Volume 1.0: Functional workflow composer. Deliverable 4.1
Project FACTS4WORKERS: Worker-Centric Workplaces in Smart Factories

Reference / Citation

Van Herwegen J. Arndt D. Salliau F., (2016), "Functional Workflow Engine". Deliverable 4.1 Project FACTS4WORKERS: Worker-Centric Workplaces in Smart Factories

www.facts4workers.eu

1. Printing, July 2016
Layout and Setting: Florian Ott, Cooperation Systems Center Munich

Worker-Centric Workplaces in Smart Factories

E-Mail: facts4workers@v2c2.at
Internet: www.facts4workers.eu

This document is published under a
Creative Commons Attribution Non Commercial No Derives licence.
You are free to copy and redistribute the material in any medium or format. You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. You may not use the material for commercial purposes. If you remix, transform, or build upon the material, you may not distribute the modified material.
http://creativecommons.org/licenses/by-nc-nd/4.0/

About this document

Executive Summary

This document represents Deliverable 4.1 ("Functional Workflow Composer") of the H2020 project "FACTS4WORKERS - Worker-Centric Workplaces in Smart Factories" (FoF 2014/636778).

Activities in production environments are dynamic and change, based on decisions made by workers, changing circumstances (e.g. defects). A smart factory infrastructure and architecture should accommodate this adaptiveness.

At the core of such an infrastructure is the functional workflow composer, embodied by the Semantic Workflow Engine (SWE). The SWE is a framework capable of interlinking RESTful web services (REST APIs) and combining their functionality to reach goals unachievable by only using a single API.

The SWE will power the complex service composition of future factory work places.

Document authors and reviewers

The following persons have contributed directly to the document. Please note that many other people have supported our work and we thank them sincerely.

Lead Authors

Name	Organisation	Role
Joachim Van Herwegen	iMinds	WP4 Lead
Dörthe Arndt	iMinds	WP4 Lead
Frank Salliau	iMinds	WP4 Lead

Reviewers

Name	Organisation	Role
Martin Wifling	Virtual Vehicle	Project Coordinator
Gianni Campatelli	Universita di Firenze	WP3 Lead
Peter Brandl	Evolaris	WP2 Lead
Stelios Damalas	Virtual Vehicle	WP8 Lead

Table of Contents

Index of Abbreviations

API	Application Programming Interface
HMI	Human Machine Interface
JSON	JavaScript Object Notation
N3	Notation3
REST	Representational State Transfer
SWE	Semantic Workflow Engine

1 Introduction

The **mission** of the HORIZON 2020[1] project FACTS4WORKERS is to develop and demonstrate solutions that support including an increasing number of knowledge work elements into the work done on the factory floor. We see a great potential in the use of information and communication technology (ICT) to provide production employees with the information they need to perform their daily work at the right time and in an appropriate manner. These smart ICT solutions should, inter alia, improve decision making, support the search for problem solutions, and ultimately strengthen employees' position on the factory floor.

With FACTS4WORKERS, we want to contribute to the **vision** of a "smart factory" in which smart workers play a central role in the production process and ICT solutions support them in the best possible way. As the most flexible element, smart workers are the focus of attention, and their role is extended far beyond factory work's conventional automated storage routine activities. An autonomous work environment will help them continuously improve knowledge sharing and effective knowledge acquirement in the workplace.

1.1 Semantic Workflow Engine

The Functional Workflow Composer, known in the F4W project under the name Semantic Workflow Engine (SWE) is a framework capable of interlinking RESTful web services (REST APIs) and combining their functionality to reach goals unachievable by only using a single API.

It generates workflows, using metadata descriptions about the APIs, describing the steps necessary to process the data. This includes the order in which the APIs need to be called to make sure their dependencies are in order.

The SWE has several tasks, all of which are described more extensively in Chapter 3.

- Storing the descriptions of all available services.
- Generating workflows where every step is such a service.
- Calling the necessary APIs to execute the generated workflows.
- Informing the user of additional requirements or success/failure when the workflow is finished.

[1] https://ec.europa.eu/programmes/horizon2020/

1.2 Relevance to the project

Activities in production environments are dynamic and change, based on decisions made by workers, changing circumstances (e.g. defects). A smart factory infrastructure and architecture should accommodate this adaptiveness.

The semantic workflow engine is at the core of such an infrastructure. It will power the complex service composition of future factory work places.

For a full description of the architecture we refer to the relevant documents[2]. The main goal of the SWE is to connect the Human-Machine Interface (HMI) and the backend services. The HMI asks the SWE what the next step should be, of which the response is based on the available backend services and how they interlink with each other.

[2] Gerhard, Detlef; Dumss, Stefan; Rosenberger, Patrick (2016): Deliverable 5.1: Blueprint Architecture and Integration Plan. Project FACTS4WORKERS: Work-Centric Workplaces in Smart Factories.

2 Semantic Workflow Engine

In this chapter we will discuss the ideas and semantics behind the SWE without going in-depth on the implementation details, which we reserve for Chapter 3.

2.1 Workflows

In this document we are often going to talk about workflows so it's important that we explain what we mean with that word since there are several different interpretations out there. For us a workflow is a sequence of actions that can get executed. These actions can be APIs that get called or a worker that does something, such as measuring a device. All of these actions have their input and output (explicitly defined in the case of Web APIs).

Workflow diagrams are a familiar concept for many people that worked on projects. These diagrams describe how all the available actions can interact with each other and how they should be connected. Usually for every use case there is a corresponding workflow diagram that describes the steps that need to be taken to complete the objectives of that use case, possibly with several branching paths in case of situations with multiple possible outcomes. If some of the actions described can be re-used in several different diagrams, in which case they often are described in separate blocks so the necessary information only needs to be written once and can then be re-used without having to worry if a detail of the action changes.

Some of the disadvantages of these workflows are that they can be labor intensive. If new elements get introduced or certain components change a rewrite of some or more of the diagrams could be necessary to make sure that everything still fits together. Another issue is that it is not always easy, or even possible, to take all possible situations and problems into account. Especially in situations where there are many possible solutions and a multitude of possible combinations it can be hard to write everything down.

2.2 Describing the actions, not the workflow

The main goal of the SWE is to automate the process of creating workflows. The SWE is designed with a different viewpoint than normal workflow diagram creation. Whereas there you would describe how the actions are interlinked, for the SWE you completely **describe** the actions themselves: what their input is and what output they provide. The SWE then interprets all these actions and automatically finds all possible **links** between all these actions. It does not actually instantiate all these links though, since the possibilities might be too large to be useful. If a user wants to use the SWE to create a workflow, they have to provide it with a **goal**, describing to the SWE what it is they want to achieve or which data they want to acquire. Given this goal, the SWE will automatically generate a workflow, describing all the necessary steps to reach that goal.

A big difference from the workflow generated by the SWE is static workflows, is that it can change during execution; it is a live and dynamic workflow. To support the possible branching nature of a workflow, the SWE makes certain assumptions about decisions that might be taken or output that might be returned in cases where multiple solutions are possible.

To support this dynamic nature, it is necessary that the output of every action gets fed back into the SWE. It can then take that new data into account to possible update the next step, meaning that **the next step can change** based on the output of the previous step. For example: the full address of a worker is required to complete an action. The SWE knows we have a database containing that information for all workers, so it suggests a step in the workflow that corresponds to accessing that database where the next step would then be to interpret that address. If the database returns that it couldn't find the address of that worker the SWE would have to update the workflow, for example by accessing the yellow pages, before the step of interpreting the address can be executed.

The main advantage of this system is the flexibility: it is not necessary to take into account all possible combinations and situations. Only the available actions need to be described. Of course this is not without its own **challenges**: the descriptions need to be adequate and consistent so the SWE can detect all possible links. Additionally, it might not always be clear which actions are necessary before the SWE can reach the requested goals. There it is still the responsibility of the developer to make sure that the descriptions are complete.

3 Implementation details

There are several important components to the SWE that get combined in the overlapping framework to generate and execute dynamic workflows. All APIs are described independently. These descriptions then get combined in a reasoner to generate a possible path through those APIs. If user input is required the process is halted until the necessary feedback was received.

3.1 RESTdesc

In the F4W project, most of the backend data is exposed through RESTful[3] APIs, using the JSON[4] data format. The goal of the SWE is to combine all these different APIs into a bigger whole, making it necessary for the SWE to understand what these APIs are capable of.

RESTdesc[5][6] is a method to **describe the metadata** of APIs, designed specifically to solve this problem. It can be used to describe all the necessary components of an API: how to call it, what to send it, what it returns and how to interpret that result. Especially the interpretation part is important since it allows the SWE to link the results of multiple APIs together.

To describe these APIs, RESTdesc makes use of Notation3[7] (N3), which is a shorthand serialization format of semantic data, designed with human-readability in mind. A (N3) RESTdesc description of an API consists of the following 3 components:

1. The conditions that need to be fulfilled before this API can be called.
2. How to call this API.
3. How the results of this API call should be interpreted.

[3] https://en.wikipedia.org/wiki/Representational_state_transfer

[4] http://www.json.org/

[5] http://restdesc.org/

[6] Verborgh, R., Arndt, D., Van Hoecke, S., De Roo, J., Mels, G., Steiner, T., Gabarró Vallés, J., 2016. The Pragmatic Proof: Hypermedia API Composition and Execution. Theory and Practice of Logic Programming. Accepted for publication.

[7] https://www.w3.org/TeamSubmission/n3/

An example showing these 3 components can be seen below:

```
@prefix : <http://example.org/image#>.
@prefix http: <http://www.w3.org/2011/http#>.
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
{
  ?image :smallThumbnail ?thumbnail.                    1. Conditions
}
=>
{
  _:request http:methodName "GET";
            http:requestURI ?thumbnail;                 2. How to call
            http:resp [ http:body ?thumbnail ].
  ?image dbpedia-owl:thumbnail ?thumbnail.
  ?thumbnail a dbpedia:Image;                           3. Results
             dbpedia-owl:height 80.0.
}.
```

The example here shows a description for an API that works for all images of which we have a URI to their thumbnail. In this case the thumbnail will be returned and the corresponding metadata (e.g. the height is 80.0) will be stored.

Unlike plain text, N3 is a machine-readable format, allowing machines to interpret these API descriptions. If these descriptions then use the same "language" the machine can discover APIs that use the same data. For example, in the N3 above we use *dbpedia:Image* to describe that something is an image. If there is another API that takes such a *dbpedia:Image* as input then it can use the output of the example here.

3.1.1 JSON serialization

JSON is one of the most common data formats used for RESTful APIs. Since RESTdesc originally only had support for APIs returning semantic data, we extended the descriptions with the possibility to describe other formats in the API in/output, such as JSON. A (shortened) example below shows how this would look like when describing how to call an API that returns JSON:

```
_:request http:methodName "GET";
          http:requestURI "https://api/part";
          http:resp [
              http:body _:body
          ].
_:body rest:contains { rest:json
    json:type _:type;
    json:suggestion _:sug;
    json:value _:value
}.
```

The body described here corresponds to a JSON object containing 3 keys: type, suggestion, value. For example:

```
{
    "type": "width",
    "suggestion": 20.0,
    "value": 18.5
}
```

Using the *json* prefix, such as in *json:type*, alerts the framework that the data needs to be converted from JSON to N3 and back. Without going in-depth in the actual workings, the framework supports a fully bidirectional 1-to-1 translation.

3.2 Reasoning

Besides being a formalization of semantic data, N3 also introduces a logic[8] to reason over this semantic data. It is possible to describe a set of rules in N3 to generate new knowledge, extending the data we have. A simple example of such a rule is shown below.

```
{
    ?animal :eats :meat.
    ?animal :eats :plants.
}
=>
{
    ?animal a :omnivore.
}.
```

This rule tells us that if an animal eats meats and plants it is an omnivore. This data is then part of our knowledge even if there is no omnivore entry in our database, allowing us to work with more data than we actually have stored. The job of an N3 reasoner is then to go over all the data available and see which rules can be applied.

As can be seen above, the "=>" symbol is used to denote the two parts of a rule. This can also be seen in the RESTdesc descriptions: there it is used to split up the requirements from the actual API body. Since the RESTdesc descriptions are also written as rules, they can also be interpreted by a reasoner and applied to the data. In our framework we make use of the EYE reasoner[9]. The reasoner can chain these descriptions together, using the output from one as input for another to form a proof of how these descriptions can be combined to reach the requested goal. A shortened example of this can be seen below, where 3 APIs are chained together.

[8] https://www.w3.org/DesignIssues/N3Logic

[9] http://eulersharp.sourceforge.net/

```
@prefix : <http://facts4workers.eu/>
@prefix http: <http://www.w3.org/2011/http#>
@prefix r: <http://www.w3.org/2000/10/swap/reason#>

[] a r:Proof, r:Conjunction; r:component <#lemma1>;
   r:gives { :cog a :Part. :cog :validation _:sk_8. }.
```

Goal

```
<#lemma1> a r:Inference;
  r:gives {:cog a :Part. :cog :validation _:sk_8};
  r:evidence (<#lemma5> <#lemma4>); r:rule <#lemma7>.

<#lemma2> a r:Inference; r:gives {
    _:sk_0 http:methodName "POST".
    _:sk_0 http:requestURI "http://facts4workers.eu/parameters".
    _:sk_0 http:body :cog.
    _:sk_0 http:resp _:sk_1. _:sk_1 http:body _:sk_2.
    :cog <http://facts4workers.eu/parameters> _:sk_2};
  r:evidence (<#lemma5>); r:rule <#lemma10>.
```

API

```
<#lemma3> a r:Inference; r:gives {
    _:sk_3 http:methodName "POST".
    _:sk_3 http:requestURI "http://facts4workers.eu/produce".
    _:sk_3 http:body _:sk_2.
    _:sk_3 http:resp _:sk_4. _:sk_4 http:body _:sk_5.
    :cog :measurements _:sk_5};
  r:evidence (<#lemma2> <#lemma5> <#lemma6>); r:rule <#lemma9>.
```

API

```
<#lemma4> a r:Inference; r:gives {
    _:sk_6 http:methodName "POST".
    _:sk_6 http:requestURI "http://facts4workers.eu/validation".
    _:sk_6 http:body _:sk_5.
    _:sk_6 http:resp _:sk_7. _:sk_7 http:body _:sk_8.
    :cog :validation _:sk_8};
  r:evidence (<#lemma3> <#lemma5>); r:rule <#lemma8>.
```

API

```
<#lemma5>  a r:Extraction; r:gives {:cog a :Part}.
<#lemma6>  a r:Extraction; r:gives {:cogMachine a :Machine}.
<#lemma7>  a r:Extraction; r:because [ a r:Parsing ].
<#lemma8>  a r:Extraction; r:because [ a r:Parsing ].
<#lemma9>  a r:Extraction; r:because [ a r:Parsing ].
<#lemma10> a r:Extraction; r:because [ a r:Parsing ].
```

To interpret the proof generated by the EYE reasoner, as seen in the example above, a second reasoning run is executed on that proof. Using a custom set of rules designed specifically for this, the second reasoner run returns the descriptions of all APIs of which the conditions have been fulfilled, while removing all other components of the proof. In this case the resulting output would be the first API, while the other APIs have to wait on the output of the first before they can be used.

3.3 Executing the workflow

The N3 output generated by the reasoner is interpreted by the framework to know what following steps are necessary. The framework will call all APIs described by the proof output and feed their results back to the reasoner so the next steps can be found.

3.3.1 APIs failing

It is always possible that one of the APIs is (temporarily) unavailable. In that case the SWE needs to find a different path, not making use of that API. If there is such a problem, the relevant description will be removed for the remainder of the process. If no workflow is possible without that API, the SWE will report that no solution is possible. This description will be used again for every new process. If it remains a problem, it should be permanently removed to avoid unnecessary work by the SWE.

3.3 Executing the workflow

4 Updating RESTdesc for F4W

There are several components of the RESTdesc technology that had to be adapted to fit the F4W project. These are described here.

4.1 User interaction

RESTdesc was originally designed to combine several distinct APIs available on the Web. For F4W a new component was introduced in the workflow: user input. This interrupts the flow of the SWE and requires a pause while waiting for the user to provide additional data.

To simulate these calls to the user in the workflow generation we created RESTdesc descriptions of actions the user could take, such as measuring a device or writing a report. Since the reasoner us unaware if the descriptions correspond to actual existing APIs this is no issue in the workflow generation. Afterwards, when executing the workflow, this has to be taken into account by the SWE that these steps require different actions from the normal API calls.

Once the user is finished generating his response, which could take some time if much work is involved, a new call gets made to the SWE with the new information from the worker. This can then be integrated with the rest of the knowledge to generate the remainder of the workflow, just like for a normal API.

4.2 HMI screen order

An issue related to acquiring information from the user is simply displaying relevant information to the user. While the SWE does not really care about the order of actions if they can be executed in parallel, for the HMI this can be more of a problem. To prevent this problem we sometimes had to force certain user calls to precede others by adding additional restrictions to the requirements of those action descriptions.

Additionally, sometimes the user is interested in seeing some intermediate information before or after providing some input. To this end we introduced some additional RESTdesc descriptions whose sole function is sending data to the HMI. To make sure these integrated in the workflow, since they are not actually necessary to execute it, we added some additional restrictions to those actions and made them generate some status data about the workflow process. That status data was then used later on as a requirement for other actions, making sure the information update is necessary to execute the workflow.

4.3 Storage

So far the RESTdesc technology has only been used as single instances, where a client uses the reasoning technology to solve its problems locally. In F4W we needed a server architecture that can respond to the requests of multiple users simultaneously and independently. Additionally, the workflows generated could span multiple hours or even longer. Both of these issues could potentially cause problems if all data was kept in memory at all times.

To that end we introduced a database to support the data storage of the SWE. For F4W we decided to go with Redis[10]. Redis is a key/value store with some additional features relevant to the project.

For every separate instance a unique identifier gets generated. This id is used for the entire duration of that run to reference all corresponding data specific to that run. Every time the SWE requests a response from the end-user, all intermediate data gets stored in the store under the corresponding id. Once the user sends his response to the SWE, and includes this id, the data can be retrieved again, without having to keep anything in memory for that run in the meantime.

Another feature of Redis is that we can set a maximum age on the stored data. Meaning that if data does not get accessed for some time it automatically gets deleted. This is quite useful since we are working with temporary data that is irrelevant after a run is completed or the user does not continue the process before it was finished. There is no way for the SWE to know in those cases that the user is not going to return. For F4W we put the timeout on 24 hours, meaning that all data that was not used for a full day will get deleted automatically.

[10] http://redis.io/

5 SWE API details

In this chapter we will go through an explicit example showing how the SWE works for a user accessing it. We will be using the Hidria use case where the worker has to measure the size of a part so its dimensions can be compared to the expected results. We will make use of our SWE API located at http://f4w.restdesc.org/ .

The use case is quite a simple one:

- The user logs in
- The user which part he is working on
- The user measures the part produced by the machine and fills in the measurements. He also indicates if this part should count as a golden sample for all other parts or not. (For this example it should not).
- The user receives the results, indicating whether the part has the correct dimensions.

5.1 Example workflow

To start the SWE process the users sends it a message indicating which use case it wants to start up. Internally the SWE has a list of all use cases and associates with them the corresponding goals. We decided on letting the user just send us the name of the use case instead of the N3 goal for ease of use. In this case the use case is called "HIR_offset_golden".

When the SWE receives this input, it generates a possible workflow. At this point it does not know yet if the measurements are going to be made for the golden sample or not, so it just assumes one of the two options until an actual choice is made. This is what the generated workflow would look like:

1. Ask user for authorization.
2. Send authorization data to authorization API.
3. Request a list of all possible parts from the parts API.
4. Send this list to the user.

5. Ask the user which part he wants to measure.
6. Request the specific part data from the parts API.
7. Ask the user for measurements and whether this is a golden sample or not
8. Send the measurements to the golden sample API. (For now the SWE assumes it is golden sample data, this is no problem until we reach that point).
9. Send the final results back to the user.

This workflow is not fixed, it can (and will) change based on the input it receives from both APIs and worker input. Let's take a look at what happens in the first few steps.

- The user receives a request to provide authorization and inputs his username and password.
- Receiving this information, the SWE generates a **new workflow**. It will look quite similar to the one above; only step 1 will be removed since it is no longer necessary (the user authorization information is now present in the SWE).
- The first step of the new workflow is now to contact authorization API with the provided information. Assuming the information is correct and the authorization succeeds, a new workflow will again be generated where the first step is now to request all parts from the API.
- Etc.

As can be seen the SWE updates it workflow after every step, using the newly provided information to update its knowledge base and update its decisions.

For now the workflow did not really change, the steps only got removed in the order they appear. This is no longer the case in step 7 where the user inputs his measurements and the fact that he is actually **not** doing a golden sample measurement. Once the SWE receives this information the workflow it generates will look different from the previous ones:

1. Send one of the measurements to the measurement API.
2. Let the user know if the measurements were valid.

These steps are new compared to the steps 8 and 9 in the workflow example above. There is actually another assumption that the SWE makes here. The measurement API only accepts single measurements instead of a single list, so the SWE has to send them one by one. At this point the SWE assumes that it will be finished after sending 1 result, but this will be updates after every call until the list of measurements is empty.

This is only a simple example, but already shows how the SWE works with changing data while updating its workflows.

5.2 SWE API specifications

To call the SWE, we provide a restful JSON API at http://f4w.restdesc.org/next . To start the user has to POST a JSON object containing his goal. This would look as follows:

```
{
    "goal": "HIR_offset_golden"
}
```

This corresponds to the first step in the example above. Once the SWE generated the workflow, the user would receive this response:

```
{
    "http:requestURI": "authorization",
    "http:resp":
    {
        "http:body":
        {
            "contains":
            {
                "username": "_:e_sk_4_1",
                "password": "_:e_sk_5_1"
            }
        }
    },
    "callID": "d9544f82-486e-4b10-8f03-617fa21ba589",
    "data": "62f08501-8917-4eb2-bc6e-79104a9eb424"
}
```

(Some unimportant data was removed). Every field here has an important function.

- **http:requestURI** indicates the type of request. This can be used by the HMI to know what to show to the user.
- **http:resp** indicates what the SWE expects from the user. In this case it expects a JSON object containing at least a **username** and **password** field.
- **callID** and **data** are two unique identifiers used by the SWE to later on continue working the with the correct knowledge base.

It is important the SWE receives both the response from the user and the identifiers it sent, so it knows where to continue from. A response from the user back to the SWE would look like this:

```
{
    "goal": "HIR_offset_golden",
    "json":
    {
        "username": "myUserName",
        "password": "secret"
    },
    "eye":
    {
        "http:requestURI": ...
    }
}
```

A user response back to the SWE should always contain these 3 fields:

- **goal** the goal that needs to be used. This needs to be sent again since the user is free to change it during the workflow.
- **json** the data the SWE requested, in this case the JSON object containing the username and password.
- **eye** the unique identifiers the EYE reasoner needs to know where to continue reasoning. To make it easier on the user we simply request to send back the entire object they just received from the SWE. This field is a 1-to-1 copy of the SWE output in the step before.

Afterwards this process just continues. The SWE will continue sending these requests and the user keeps responding like this until the workflow is finished.

5.3 SWE API extras

There are two other APIs available besides the **/next** API call on the SWE.

/back allows the user to go back to the previous step in the workflow where user input was requested. This tells the SWE to throw away all information it gathered since that point. To call this API a POST needs to be done where the body is a JSON object containing at least a **data** field. This can be done by for example sending back the last request received from the SWE.

/clear deletes all data from this workflow from the SWE. This is useful once the user is finished with the workflow to make sure the SWE does not store too much unnecessary data. Just like the **/back** call this is done by sending a JSON object containing a **data** field.

6 Future work

6.1 Use cases

So far the SWE has only been used in a few use cases. We are currently investigating its flexibility when applying it to different use cases. The main goal is to push the limits of the SWE: see how it copes when there are a lot of solutions possible and a variety of workflows. We are also investigating if some use cases can be adapted to have more diverse workflows of the backend systems without having to include the worker, since there might be more possibilities there.

6.2 Error handling

If an API fails the SWE needs to find other paths not using that API. We are still investigating multiple solutions there. In case of a temporary outage it might be more interesting to just retry calling the API for example. The current use cases do not support alternative paths yet so this requires further investigation.

6.3 RESTdesc robustness

Not all possible situations were taken into account when RESTdesc was designed. It is possible that APIs return different formats depending on the input, or only returning a part of what was described for example. All these different situations need to be caught by the SWE and handled correctly, without negatively influencing the results.

6.4 Multiple paths

In the future there might be use cases where there are multiple paths possible for a single solution. In that case it needs to be determined which paths are optimal, if any. Here again there are multiple solutions we are investigating, such as taking the length of the path into account, assigning weights to the descriptions or taking the responsiveness of the APIs into account.

6.5 Performance

Currently the performance is not yet at the level of where we want it. If the HMI has to wait several seconds before getting a response from the SWE, the user experience is degraded. This can be improved by smarter database access and better rules for the reasoner, but it is not always easy and clear to see how and where this needs to be done. This is an issue we continue working on.

7 Conclusion

The Semantic Workflow engine greatly reduces the complexity for the end-user if there is a variety of options and possibilities available to reach an end goal. Especially if it is not clear how the available services can be combined to create a complete result.

A problem that we are currently facing is that the SWE might not be fully utilized to its strengths in the use cases so far. This due to more of a focus on the available user interfaces and less on the possible data flows in the use cases so far, which is why we are further investigating how these can be extended.

Due to the way the SWE works it is no problem to add or extend use cases, i.e. no change in the code base is required, only new descriptions need to be added. This shows how the system is use case independent and can be further utilized in the future to fit the upcoming needs that might arise.

Integrating the worker into the workflow of a system that was designed purely to handle Web APIs was a challenge, but we feel that the solution we currently have is adequate and clean. From the reasoner point of view there is no problem if there are worker action descriptions in there, and for the SWE several components are generalized so that both Web API and worker actions are handled the same, e.g. in output data integration and storage.

ABOUT THE PROJECT

The high ambition of the project FACTS4WORKERS is to create Factories of the Future with a pervasive, networked information and communication technology that collects processes and presents large amounts of data. These smart factories will autonomously keep track of inventory, machine parameters, product quality and workforce activities. But at the same time, the worker will play the central role within the future form of production. The ambition of the project is to create »FACTories for WORKERS« (FACTS-4WORKERS), to strengthen human workforce on all levels from shop floor to management since it is the most skilled, flexible, sophisticated and productive asset of any production system and this way ensure a long-term competitiveness of manufacturing industry. Therefore a serious effort will be put into integrating already available IT enablers into a seamless and flexible Smart Factory infrastructure based on work-centric and data-driven technology building blocks.

These solutions will be developed according to the following four industrial challenges which are generalizable to manufacturing in general:

- Personalized augmented operator,
- Worked-centric rich-media knowledge sharing management,
- Self-learning manufacturing workplaces,
- In-situ mobile learning in the production.

PROJECT PARTNERS

The FACTS4WORKERS project is composed of 15 partners from 8 different European countries:

Virtual Vehicle Research Center	Austria
Hidria TC Tehnološki center d.o.o.	Slovenia
Universita degli Studi di Firenze, Department of industrial Engineering	Italy
Technische Universität Wien	Austria
ThyssenKrupp Steel Europe AG	Germany
Hidria Rotomatika d.o.o., Industrija Rotacijskih Sistemov	Slovenia
iMinds VZW	Belgium
Sieva d.o.o.	Slovenia
University of Zurich, Department of Informatics	Switzerland
Thermolympic S.L.	Spain
EMO-Orodjarna d.o.o.	Slovenia
Evolaris Next Level GmbH	Austria
Itainnova - Instituto Technologico de Aragon	Spain
Schaeffler Technologies AG & Co. KG	Germany
Lappeenranta University of Technology	Finland

PROJECT COORDINATOR / CONTACT:

VIRTUAL VEHICLE Research Center
Inffeldgasse 21A
8010 Graz, AUSTRIA

Tel.: +43-316-873-9077
Fax: +43-316-873-9002
E-Mail: facts4workers@v2c2.at

FOLLOW US AT:

FACTS4WORKERS
@FACTS4WORKERS
facts4workers-project

Functional Workflow Composer

This document is entitled "Functional Workflow Composer" and represents Deliverable 4.1 of the H2020 project "FACTS4WORKERS - Worker-Centric Workplaces in Smart Factories" (FoF 2014/636778).

Activities in production environments are dynamic and change, based on decisions made by workers, changing circumstances (e.g. defects). A smart factory infrastructure and architecture should accommodate this adaptiveness.

At the core of such an infrastructure is the functional workflow composer, embodied by the Semantic Workflow Engine (SWE). The SWE is a framework capable of interlinking RESTful web services (REST APIs) and combining their functionality to reach goals unachievable by only using a single API.

The SWE will power the complex service composition of future factory work places.