STORY

added value of STORage in distribution sYstems

# Deliverable 3.7 Interoperability platform

| | |
|---|---|
| Revision............... | 1 |
| Preparation date .. | 31-10-2019 (M54) |
| Due date .............. | 30-04-2019 (M48) |
| Lead contractor.... | UCL |
| Dissemination level | PU |

**Authors:**
Paul Valckenaers – UCL

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

## Table of contents

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Project STORY - H2020-LCE-2014-3

## Executive summary

In-depth interoperability is about the software not adding constraints that don't exist in the world-of-interest. In-depth operability also is about recognising which software unavoidably adds constraints to its world-of-interest and, subsequently, avoiding to build up inertia for these constraints.

Because such software does not introduce constraints, or ensures constraints are easy to deactivate, developers are able to use them regardless of the system requirements. The software will be knowledgeable about the world-of-interest. Thus, their services will save development time, prevent errors, reduce the demand for human expertise. etc.

In contrast and inherently, in-depth interoperable software cannot deliver 'final solutions' as real-world systems must introduce constraints when approaching the time to deliver their services (i.e. when the time is now, everything is fixed/decided in a pragmatic sense).  That's why it is (only) a platform on which to build solutions.

The in-depth interoperability platform solely comprises the former: software that does not add constraints. Decision-making constraint-introducing software constitutes an application running on the platform. Triggered by discussing a draft version of this deliverable, the scope has been extended to include a generic design to limit the inertia build-up of decision making within applications that execute on the in-depth interoperability platform. The effort was shifted from covering more real-world counterparts in the energy domain toward revealing a key mechanism to contain the impact of decision making elements within an application running on the platform.

Research identified digital twins of parts of reality as the key opportunity. This implies that platform software modules mirror a corresponding real-world counterpart in their world of interest (e.g. a pump). As the world of smart energy comprises a significant and continuously-growing number of such real-world counterparts, the list of candidate digital twins is impractically large without a concrete installation drawing a boundary. Therefore, this deliverable comprises software modules that "lead by example". The modules show how it is done while maximising accessibility by avoiding non-contributing functionality.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

This document provides guidance to understanding the platform and its software modules. In particular, it discusses the connection to the real world by means of device drivers. Here, the innovation is to recognize the importance of application independence for the device drivers.

Moreover, the document outlines the range of configurations that can be realized when following the guidance from this deliverable. These configurations include normal deployment, simulation and embedded simulation (cf. figures 3.1, 3.2 and 3.3). All configuration use the same software (e.g. the digital twin of a pump); there is no need to have multiple versions and no need to keep them consistent. The embedded simulation, when shared by multiple controllers, generates a prediction of what will happen without any explicit coordination among these controllers.

The deliverable is this document and the source code of the modules.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

# 1 Introduction

This document provides guidance through the software modules of deliverable D3.7 of the STORY project.

## 1.1 What is the in-depth interoperability platform?

This deliverable comprises software modules that are digital twins of real-world resources (e.g. pumps, metering devices, fluid in circuits) and real-world activities (e.g. circulating a fluid through a circuit). These digital twins constitute the in-depth interoperability platform on top of which applications execute. These twins are used directly by application software and/or are aggregated into performers.

Performers are applications on the in-depth interoperability platform (e.g. a climate control for an office building). Performers combine digital twins of resources and activities with decision making. The decision making implies that performers are not part of the in-depth interoperability platform (because decisions are potential sources of conflict).

In contrast, conflicts with digital twins are conflicts with their real-world counterpart, not with design choices made by the digital twin developers. When solving a conflict, the digital twins remain unaffected (especially when they mirror any modifications in their real-world counterpart).

In the discussion of draft versions of the deliverable, the viewpoint from industry within the consortium pointed out that answers for access control, authorization, etc. are relevant. Accordingly, the deliverable comprise software modules that support resource allocation and authorization explicitly. In the context of in-depth interoperability, this represents a generic design to limit the inertia build-up for the decision making within performers.

Next to the normal user, D3.7 introduces a super-user who controls resource allocation as well as the authorization to access services from performers. The super-user is able to de-allocate resources, overruling the performer that aggregates them, and thus disable any decision-making (to address a conflict). Overruling is likely to break some services provided by the performer, and may need to be re-implemented by the overruling party, but allows to undo decisions (e.g. add new services).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

As the platform comprises digital twins of its world-of-interest, the number of software modules in real-world installations scales with the corresponding reality. In a technical world, the number of modules will be overwhelming unless the developers focus on a specific installation or equipment range. Accordingly, D3.7 provides software modules to show how to elaborate digital twins and performers for a given installation while keeping these example modules as accessible/simple as possible. In other words, the in-depth interoperability platform guides by example where the ultimate platform emerges from real-world installations, each covering their specific real-world counterparts.

## 1.2 How to read this document

The next two sections, chapters 2 and 3, are of general interest. Readers who want to get an overall picture only need to read those. Chapter 2 covers device drivers, the mechanism by which the software accesses the real world. Implementing these device drivers is not an innovation; computer operating systems use this approach for several decades. The innovation is to ensure that device drivers are application-independent.

Chapter 3 covers overall system configurations for installations and systems based on the D3.7 in-depth interoperability platform. Because the digital twins provide (executable) models of their real-world counterpart, these configurations include simulation. This can be a straightforward simulation, which will be software-in-the-loop (i.e. the simulation comprises and uses the control software as it would be deployed). It can execute faster-than-real-time by manipulating time (as seen by the software).

This can be an embedded simulation in which the overall control system uses a faster-than-real-time simulation to predict what will happen. The control may simulate more than one option (i.e. control strategy/setting) by utilizing multiple embedded simulations in parallel. Importantly, these configurations reuse the same digital twin software wherever their real-world counterpart is concerned. There never are two versions, which need to be kept consistent. Moreover, the digital twins are a single of truth within each simulation. When multiple controls share an embedded simulation, they receive a prediction of what their joint control actions are to achieve.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

The remaining chapters cover the software modules, intended for reader familiar with software development. The software modules in D3.7 have been written in Erlang (see further) and have been designed to communicate the concepts (i.e. leading by example). They are not written according to Erlang recommended practice as this would reduce the accessibility for non-Erlang experts.

## 1.3 Ongoing activities beyond STORY – exploitation activities

As a follow-up (exploitation after and beyond the STORY project), the UCL team[1] is translating these software modules into Elixir[2] (cf. https://elixir-lang.org ). In contrast, this effort aims at a robust implementation of the platform software modules. A key difference will be the usage of supervisor processes that handle software crashes. To this end, the elixir version will adopt so-called OTP behaviours, which increase the distance between the code and its corresponding reality; e.g. the reader needs to understand the concept of a "call-back function" used to extend a "gen_server." These follow-up activities are targeting digital twins for manufacturing and inland shipping. This includes digital twins for worker and skippers (i.e. humans in work environment).

## 1.4 How to experiment with the software in Erlang[3]

1) Install Erlang from https://www.erlang.org/downloads
   or https://www.erlang-solutions.com/resources.html.
2) Start up the shell (called "werl.exe" on Windows, "erl.exe" otherwise).
3) Ensure the shell's working directory is where the source code is located.
   The shell understands "cd" to change this directory.
   Simply enter `cd("my_working_directory").`
   On Windows, create a shortcut on your desktop to werl.exe and
   adapt the "start in: " within the properties of the shortcut.

[1] When embarking on the development of your own platform, please contact paul.valckenaers@ucll.be

[2] Elixir relates to Erlang in the manner Python relates to C: Elixir induces "programmer happiness" as Python does. It has a state-of-the-art syntax, tool set, etc. and a newbie-friendly community. Erlang syntax resembles old-fashioned C and has a community of highly-paid but less approachable experts.

[3] Cf. https://www.youtube.com/watch?v=SOqQVoVai6s, an introductory talk on Erlang.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Project STORY - H2020-LCE-2014-3

4) Feel free to experiment. E.g. enter `self().` to discover the process ID of the shell itself. Next, `enter 1 / 0.` to make this process crash. Enter `self().` one more time.
5) Note that each shell command needs to be terminated with a `.` to execute.
   Likewise, all commands need `( )`, possibly with parameters inside.
   In other words, `self` will not do anything, will wait until you enter `.`
   Entering `self.` returns `self,` where `self().` will do the job.
6) Tip: the `up arrow` brings the previous command up, which can be edited before hitting enter.

The source code is provided in text files (compilers have issues with the MS Word format).

The software comprises some utility modules. The main one is msg.erl which supports messaging among the software processes that correspond to parts of the relevant reality.

The remainder of this document covers how test programs ensure the proper operation of the platform modules. In this way, they gradually disclose what they are/do.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Project STORY - H2020-LCE-2014-3

# 2 Connecting to the real world – Device drivers

Device drivers – background

Over the years, industrial automation has developed and adopted its own information technologies. Among the oldest, you have numerical control (CNC), which treated programmable equipment as 'printers of products.' This choice was justified at the time (1940s and 1950s) when an extremely expensive (> 1M€) and bulky computer (i.e. occupying a very large room) generates a paper tape in which holes were punched and several electromechanical devices – attached to milling machines – would process this tape (repeatedly), similar to mechanisms in a pianola. This is a rigid system design; everything is fixed when the holes are punched.

Similarly, programmable logic controllers (PLC) adopted the 'architecture' of installations comprising hard-wired relays, timers and sequencers. The result is excellent at real-time decision-making but representing and manipulating (even moderately complex) state representations does not come natural. Likewise, industrial robot controllers adopted their own programming language for application development. Here, the lack of critical (user) mass brings serious drawbacks.

The above differs from the manner in which mainstream (and many niches such as mechatronics) connect to devices and equipment. In modern computer systems, devices are connected through device drivers. A device driver is a computer program that operates or controls a particular type of device that is attached to a computer. It provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details about the hardware being used.

A key advantage of devices connected to an operating system through device drivers is that the programming technologies, languages and tools are not fixed by the device. Applications may select and combine what suits them best. This is particularly advantageous when applications include multiple devices types. It steadily is becoming a necessity when legacy industrial automation technologies struggle with the demands of smart networked applications.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

As needed, the device controllers also handle start-up, shutdown and power loss issues. E.g. a hard disk controller will ensure any cached data is transferred to persistent storage when power is lost. In industrial applications, a UPS (uninterruptable power supply) may address such issue for multiple devices at once. Parts of a device driver may assist the device controller respectively in start-up, shutdown and even crash recovery. When an application has hard real-time requirements vis-à-vis its devices, a suitable operating system will be needed. Note that extreme (time- and safety-critical) requirements need to be handled in the device control; handling needs to be simple and local to be reliable and verifiable anyway.

### Device drivers and in-depth interoperability

Important for in-depth interoperability is that the hardware functions of the devices – accessible through device drivers – are to be application-independent. These functions bring the abilities of the device to the computer platform without limitations based on assumptions about the application that will use the device.

The straightforward manner to achieve such application independence is to provide access to all the primitive hardware functions with adequate bandwidth. E.g. for a pump, the device driver allows to switch the pump on or off as fast as the pump's hardware will allow. When the pump control allows for multiple settings (e.g. low-medium-high or continuous between min and max), the device driver provides access.

Only when the above is technically unfeasible, non-primitive functions are to be provided but, even in this situation, these functions shall remain application-independent. E.g. when an archaic technology offers very little bandwidth, composite functions may be provided that serve a wide range of applications and penetrate applications as little as possible. E.g. in home automation, the device driver may allow an application to map buttons toward on/off switching (of lights, door locks, …). With adequate bandwidth, the device driver would be notifying the application when a button is pressed and have the application decide/execute a suitable action in response. This provision of non-

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

primitive functions is analogous to modern microprocessor offering specialized operations for encryption/decryption, multimedia encoding/decoding, AI, ...

Conversely, device invariants may be handled by the device driver and/or controller. Indeed, invariant implies application independence. In this respect, a device driver may offer normal and super-user modes. In normal mode, it may impose constraints to ensure the device is used properly (e.g. prevent abnormal wear). In super-user mode, anything that is physically possible is accessible.

### Device drivers for the digital twins

The digital twins of devices reflect the offering of hardware functions by their real-world counterpart. When the digital twin of a resource instance is created, it is to be provided with a function (actually a closure, which is a function augmented by a context) for to access each of the hardware functions of the real-world instance. In the example of a pump that can (only) be switched on or off, the digital twin receives a function that allows to access the hardware to effectively switch the real pump on or off.

For instance, to get a measurement outcome of a temperature sensor, the twin executes a device driver function (closure) that retrieves the data in the format that happens to be supported. This typically would be a low-level C program or PLC code that interacts directly with the metering device. Alternatively, the device is connected through the cloud and the device driver function accesses (reads or writes respectively when it is sensing or actuating) the server in the cloud.

Note that, when working with pre-existing installations or pre-selected equipment, device drivers may face limitations such as very low bandwidth solutions (e.g. LoRa, KNX), devices that do not give acknowledgements whether a command has been received (or not), etc. Unless there is good reason for such constraints, this kind of limitations are “poor in-depth interoperability” from the perspective of their design. It may necessitate to treat their limitations (e.g. available bandwidth) as resources that need to be managed – conceptually, they are moved out of the control system and into the world-of-interest. In view of the cost of ICT versus the cost of pumps, batteries, etc., this is better avoided.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 646426

E.g. to retrieve data from the BaseN platform from a command line, curl can be used:

```
curl -u "site1m2m:password" https://story.pilot.basen.com/_ua/story/site1/rest/...
```

In Erlang, the programming language used for the D3.7 software, it suffices to execute:

```
os:cmd(" < appropriate curl query > ")
```

See http://erlang.org/doc/man/os.html for more details on "os:cmd".

All that remains to be done is formatting and/or extracting (respectively for actuating or sensing) the relevant data.

To illustrate how such a data conversion and/or filtering may look:

```
-module(converteerKNXtemperatuur).
-export([convert/1]).

% first translate into a bitstring
convert([MSB, LSB]) -> convert2(<<MSB:8, LSB:8>>).

% when sign bit equals zero:
convert2(<<0:1, E:4, M:11>>) -> math:pow(2, E) * M / 100;

%when sign bit equals one:
convert2(<<1:1, E:4, M:11>>) -> math:pow(2, E) * (M - 2048 ) / 100.
```

These 3 lines of actual code convert a temperature in KNX-format into an Erlang data format. Note that Erlang originates from the telecom industry (i.e. Ericsson) and handles this kind of data processing extremely well.

For comparision purposes, this is the .NET version of the same transformation written in C#:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace KNXDemo.KNX
{
  class KNXConverter
  {
    public KNXConverter()
    {
    }
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Project STORY - H2020-LCE-2014-3

```
            public static double TempFromEIS5(byte[] data)
            {
                // EIS 5 - DPT 9 Format: 2-Octet Float Value (DPT 9.001 Temperature [°C])
                //  MSB: MEEE EMMM
                //  LSB: MMMM MMMM
                //  Encoding: (0.01*M)*2^(E), M is in 2's complement

                bool complement = (((data[0] & 0x00000080) >> 7) == 0x00000001);
                uint exp = (uint)((data[0] & 0x00000078) >> 3);
                uint m = (uint)(((data[0] & 0x00000007) << 8) | data[1]);
                double mantisse = 0;

                // 2's complement check
                if (!complement)
                {
                    mantisse = m;
                }
                else
                {
                    m = m - 1;
                    m = ~m;
                    m = m & 0x000007FF;
                    mantisse = m * (-1);
                }

                double result = (0.01d * mantisse) * Math.Pow(2, exp);

                return Math.Round(result, 2);
            }
        }
    }
```

### Device drivers and innovation

Device driver implementations do not represent innovation by themselves. They exclusively apply known technologies. Moreover, device driver implementation rarely can select what technology to use. It are the devices that limit and even determine what can be used. Often, this will be legacy and significantly outdated technology. Indeed, when an installation comprises high-quality equipment from an energy perspective, a strong reputation in the energy domain determines how competitive its vendors will be. There is little market pressure to replace obsolete legacy IT interfaces, which therefore will be encountered regularly today and in the future. As a fact of life, device driver

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

implementation has to face and handle a world in which it has to adapt to whatever is available, which is largely decided by the device selection by the energy experts in the team.

The innovation is to limit the penetration of legacy, outdated and obsolete IT and to manage the heterogeneity of the device interfaces. The innovation – as far as device drivers are concerned – is to keep the hardware functions application-independent. In other words, when an application changes, or when the same devices are used in another application, the device driver implementation does not need (software) maintenance (only some minimal configuration). Application developers do not need to master any of the legacy information technologies, only the IT used for the application.

With modern communication, this rarely present any difficulties. Indeed, bandwidth and response times allow the device driver to offer only the primitive functions (e.g. switch on, detect button pressed). When forced to adopt archaic legacy technologies, it may be necessary to decide about composite functions to overcome bandwidth and response time limitations. Here, domain expertise and experience is likely to be required in order to avoid applications needing device drivers enhancements or applications that need multiple versions of their software to cover identical devices.

Overall, building IT systems with device drivers that are application-independent shields from the heterogeneity (a lot of different, sometimes proprietary, communication and programming interfaces) and obsoleteness. Moreover, it limits the inertia/penetration of legacy and this heterogeneity. Thus, it facilitates the phasing-out of outdated legacy technologies and adopting of state-of-the-art communication and programming technologies.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

# 3 Configurations

The in-depth interoperability design can be deployed in the following manners (non-exhaustively as innovative developers may add new entries to this list):

- Ordinary control (cf. figure 3.1).
- Simulation (cf. figure 3.2).
- Embedded simulation (cf. figure 3.3).

In ordinary control mode, the device drivers – provided to every resource instance twin when created – directly execute the corresponding command. E.g. when a twin receives a command to switch on a pump, it mirrors this within its state representation and executes the proper device driver function (actually closure) that will (attempt to) switch the real-world pump on. Any feedback from the device driver function is captured and also mirrored in the state representation.

Even in ordinary control mode, the twins mirror their world-of-interest and compute/estimate what ought to be happening in this corresponding reality. E.g. the flow rate of the fluid in the pump's circuit will account for the pump being switched on. Such reality-mirroring then allows to coordinate interoperation in a conflict-free manner (as far as these twins are concerned). E.g. it allows to monitor system health (by looking at deviations between mirror data and measured data). It allows to compensate malfunctioning (by replacing sensor data of broken instruments by mirror data). It allows to estimate system states without energy losses when actual measurement requires switching on pumps (while the application does not need to do this).

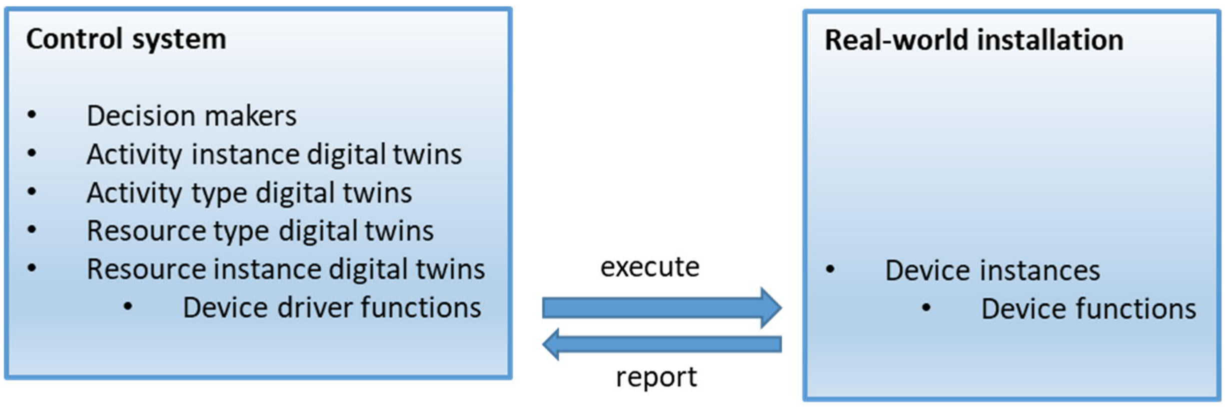
Control system
• Decision makers
• Activity instance digital twins
• Activity type digital twins
• Resource type digital twins
• Resource instance digital twins
• Device driver functions
execute
report
Real-world installation
• Device instances
• Device functions

Figure 3.1 The ordinary control configuration.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

In simulation mode, allowing to experiment with a number of installation and/or control options, the resource twins are created twice. Now, the device driver functions in the control system call the corresponding commands in the simulation. When the control systems decides to switch on a pump, it executes the corresponding function of its internal digital twin (of the pump instance). This adapts the internal state representation of the twin and has the twin execute the appropriate device driver function. This calls the same function of the digital twin in the simulation, which allows the simulation to generate a proper state representation (i.e. simulate the corresponding reality).

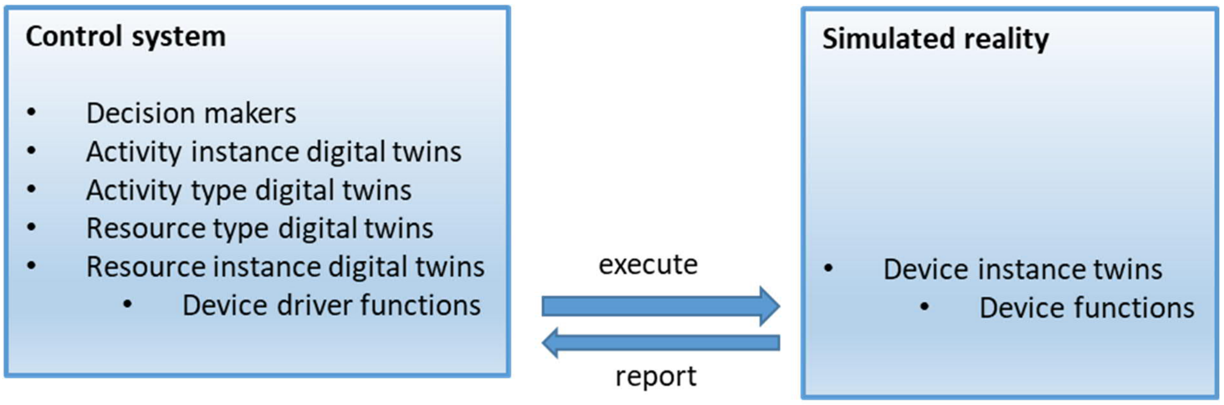

Figure 3.2 The simulation configuration.

The device driver functions of the twin in the simulation do not issue any commands but have to opportunity to report back to the control what happens. E.g. to simulate measurements, the device driver will utilise the internal state representations of the twins in the simulation. Moreover, the device driver functions can be used to introduce randomness in the simulation. They may add some measurement error (glitches, drift, noise, …) and they may introduce malfunctioning (a pump fails to switch on). They may report differently to their own simulation (allowing to mirror deviating behaviours) and the connected control system (restricting the observation of deviating behaviour to the real-world conditions of the ordinary control mode). This allows for having replications in a simulation campaign as desired.

Digital twins, both in the control system and the simulation, offer publish-subscribe services. Thus, the simulation mode allows for selecting which information is distributed to which computing processes. Moreover, the software allows to manipulate the perception of time by the software

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Project STORY - H2020-LCE-2014-3

processes. Time may go (much) faster than real-time. E.g. in thermal systems, the simulation may execute 100 times faster than it would in reality. Speed up can be higher but it may become necessary to verify when this starts to distort the results (e.g. more than a 1000-fold speed increase may cause the control reaction times to be slower in simulation than in reality). Only extreme speedup may require code adaptation (e.g. to jump across periods of non-activity from the control in almost no real-world time).

Importantly, note that the actual control system software differs minimally from the version in simulation mode. It are only the device driver functions that need to change. Basically, control system ramp up and tuning may commence in this simulation mode as the deployed control system is already present in the simulation.

The embedded simulation mode, finally, integrates the (time-accelerated) simulation mode into a two-level control scheme. The top-level is a meta-control, which decides about how the actual control will operate. This level mainly accounts for the impact in the near future by the manner in which control options are selected (think of model-predictive control).

The meta-level triggers one or more embedded simulations, selecting both control system settings (e.g. eager, conservative, optimistic) and the simulation options (e.g. level of disturbances). As the simulation executes much faster than reality, the meta-level is able to assess and decide which options to select. These options are fed into the ordinary control subsystem.

Moreover, the meta-level may indicate, at any given time, what its current options are (i.e. the control system intentions). This is fed into a shared/networked embedded simulation where this control system is not alone. There are a number of control systems, each controlling part of an overall system or infrastructure. The faster-than-real-time joint simulation propagates the joint impact of these controls, allowing to recognize issues before they occur and to grasp opportunities when they appear. This avoid controls having to remedy when it is too late to be cost-effective.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

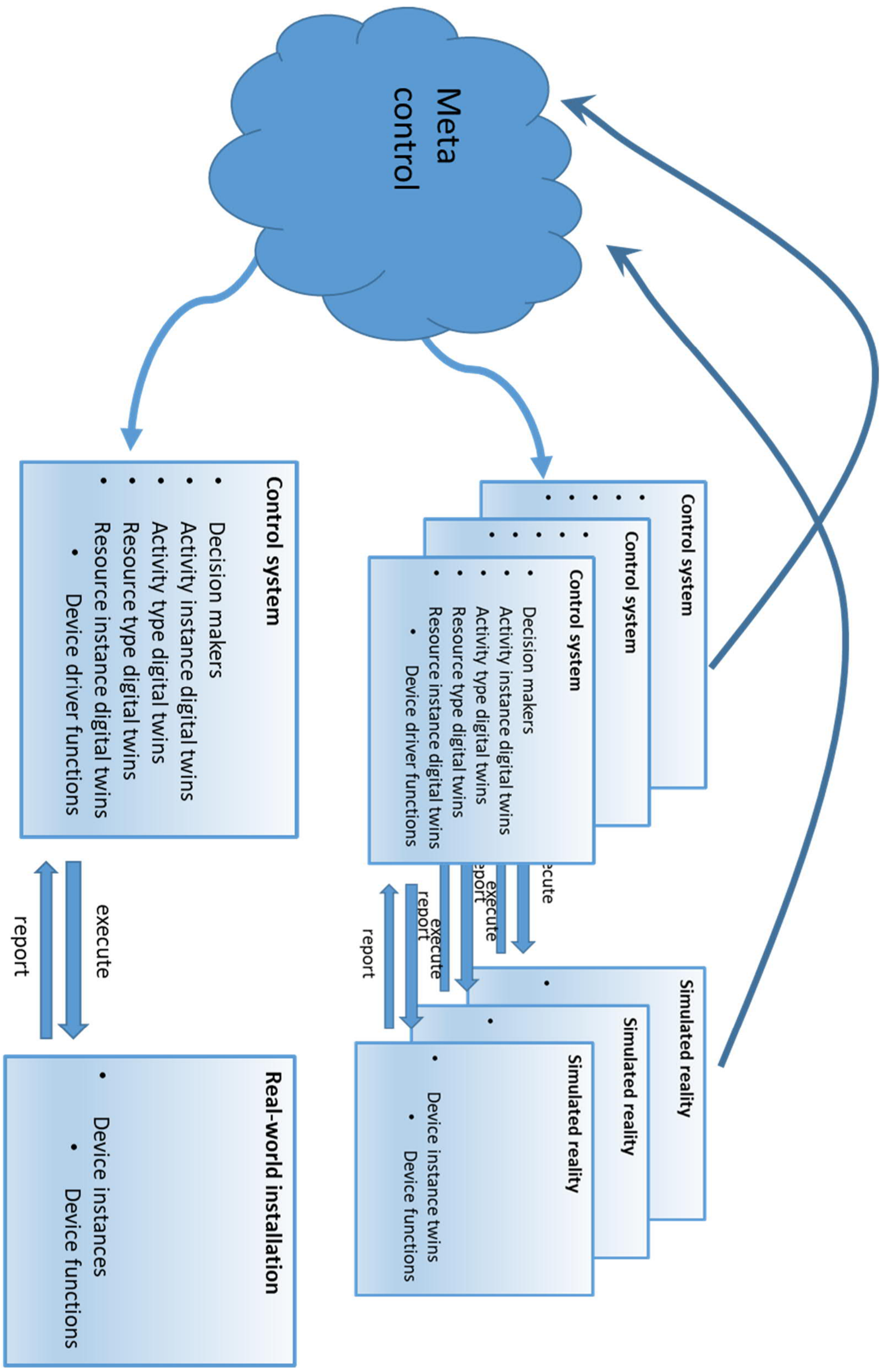

Figure 3.3 Embedded simulation mode.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Note how in-depth interoperability brings its key benefit in such scenarios. Because it mirrors reality, it interoperates. There is no need to co-design or coordinate the design of the respective controls. In a way, this embedded simulation considers the controls (i.e. the decision making elements) to reside in its world-of-interest.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

## 4 Resources

This part covers the code corresponding to resources in a smart energy installation.

First, compile and load the modules. Enter into the shell:

```
c(msg).

c(location).

c(connector).

c(resource_instance).

c(baseResInst).

c(resource_type).

c(pipeTyp).

c(pumpTyp).

c(flowMeterTyp).

c(fluidumTyp).

c(testRes).
```

The software modules related to resources in this part are:

| Module File Name | Description |
|---|---|
| resource_type.erl | Abstract module – common functions for resource types |
| resource_instance.erl | Abstract module – common functions for resource instances |
| baseResInst.erl | Concrete module for basic resource instances |
| location.erl | Concrete module for internal spaces inside a basic resource |
| connector.erl | Concrete module for connectors of basic resources |
| pipeTyp.erl | Concrete module for pipe in a heating system |

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

| | |
|---|---|
| pumpTyp.erl | Concrete module for pump in a heating system |
| flowMeterTyp.erl | Concrete module for flow meter in a heating system |
| fluidumTyp.erl | Concrete module for the fluidum in a heating system |
| testRes.erl | Module to test the above – further discussed below. |

The module testRes.erl exports the function exec. This function can be called in the shell by entering `testRes:exec(n).` where n ranges from 1...9 or a...p. This parameter selects the test that will be performed.

1) Start with `testRes:exec(1).`

This simply answers `ok` which indicates everything (erlang installation, working directory setting, compilation and loading of the code).

2) Continue with `testRes:exec(2).`

```
>> {ok,<0.3216.0>}
```

The numbers will vary. This is PID or process identifier. It is unique across the whole system (Erlang system can be huge and executing in a computer network spanning across the globe).

This is the PID for a pipeTyp (cf. D3.8 and ARTI to learn about the manner in which the world of interest is subdivided in instance and types, resources and activities).

3) Continue with `testRes:exec(3).`

```
>> {ok,#{cList => [<0.3220.0>,<0.3221.0>],
    chambers => [<0.3219.0>],
    resInst => <0.2134.0>,typeOptions => "Test no. 3"}}
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

This is the representation of the initial state of a pipe instance when it is created.

It comprises

- cList, the PID of two connectors (still disconnected),
- chambers, the PID of the location through which the fluidum will flow
- resInst, the PID of the resource instance for which this initial state representation was constructed by the type process.
- typeOptions, a free field that is used for test data but could include a serial number and a manufacturer name.

4) Enter in the shell `{ok, F} = testRes:exec(4).`

```
>> {ok,#Fun<pipeTyp.0.117079786>}
```

Next, enter `F(9).`

```
>> -0.09
```

The first result is a function to compute the (small) resistance exercised on the fluidum when it circulates through an instance of this pipe type. Its 1% of the flow rate. Of course, this function can be adapted in collaboration with the proper domain/manufacturer experts.

5) Enter in the shell `testRes:exec(5).`

```
>> {ok,<0.3230.0>}
```

`The PID of a pipe instance.`

6) Enter in the shell `testRes:exec(6).`

```
>> {ok,#{cList => [<0.3238.0>,<0.3239.0>],
     chambers => [<0.3237.0>],
     resInst => <0.3236.0>,
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Project STORY - H2020-LCE-2014-3

```
typeOptions => {<0.2134.0>,"Test no 6"}}}
```

Again an initial state representation but this time delivered by the instance.

7) Enter in the shell `testRes:exec(7).`

```
>> {ops,[],typ,<0.3241.0>,root,<0.2134.0>,chamber,<0.3243.0>}
```

Querying the pipe instance to discover that its list of supported operation is empty, the PID of its type, PID of its ‘root’ and of its ‘chamber’.

8) Enter in the shell `testRes:exec(8).`

```
>> {in,<0.3250.0>,out,<0.3251.0>,visitor,vacant}
```

The PID of the two connectors of the pipe instance and the absence of fluidum in its chamber.

9) Enter in the shell `testRes:exec(9).`

```
>> {in,<0.3256.0>,out,<0.3257.0>,dummy,<0.3258.0>,c2in,
   <0.3257.0>,c2out,<0.3258.0>,d2x,<0.3256.0>}
```

Checking out connecting connectors.

10) Enter in the shell `testRes:exec(a).`

```
>> {{<0.3261.0>,<0.3265.0>,<0.3269.0>},

 {{<0.3263.0>,<0.3264.0>},

  {<0.3267.0>,<0.3268.0>},

  {<0.3271.0>,<0.3272.0>}}}
```

Checking the construction of a loop with 3 pipe instances (of the same type).

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 646426

11) Enter in the shell `testRes:exec(b).`

```
>> {{<0.3275.0>,<0.3279.0>,<0.3283.0>},
 {{<0.3277.0>,<0.3278.0>},
  {<0.3281.0>,<0.3282.0>},
  {<0.3285.0>,<0.3286.0>}}}
```

Checking whether the connectors know their resource instance.

12) Enter in the shell `testRes:exec(c).`

```
>> {{ok,{<0.3297.0>,<0.3289.0>,<0.3293.0>}},

 {ok,{<0.3289.0>,<0.3293.0>,<0.3297.0>}},

 {ok,{<0.3293.0>,<0.3297.0>,<0.3289.0>}}}
```

Checking whether the loop is a loop: {before PID, pipe PID, next PID}

13) Enter in the shell `testRes:exec(d).`

```
>> {{ok,{<0.3312.0>,<0.3304.0>,<0.3308.0>}},

 {ok,{<0.3304.0>,<0.3308.0>,<0.3312.0>}},

 {ok,{<0.3308.0>,<0.3312.0>,<0.3304.0>}}}
```

The 3rd pipe is replaced by a pump (PID is <0.3312.0>). Note that only the resource type changes whereas the instance remains unchanged (Note: resource instances with a different topology use another resource instance module such as e.g. a valve with 3 of 4 connectors).

14) Enter in the shell `testRes:exec(e).`

```
>> {{ok,{<0.3328.0>,<0.3320.0>,<0.3324.0>}},

 {ok,{<0.3320.0>,<0.3324.0>,<0.3328.0>}},

 {ok,{<0.3324.0>,<0.3328.0>,<0.3320.0>}}}
```

The 2nd pipe is replaced by a flow meter.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

15)Enter in the shell `testRes:exec(f).`

```
>> {ok,#{cList => [<0.3340.0>,<0.3341.0>],
      chambers => [<0.3339.0>],
      resInst => <0.3336.0>,
      typeOptions => {<0.2134.0>,"pipeTyp is no 1"}}},
 {ok,#{cList => [<0.3343.0>,<0.3344.0>],
      chambers => [<0.3342.0>],
      resInst => <0.3337.0>,rw_cmd => #Fun<testRes.0.50409774>,
      typeOptions => {<0.2134.0>,"Flow meter is no 2"}}},
 {ok,#{cList => [<0.3346.0>,<0.3347.0>],
      chambers => [<0.3345.0>],
      on_or_off => off,resInst => <0.3338.0>,
      rw_cmd => #Fun<testRes.1.50409774>,
      typeOptions => {<0.2134.0>,"Pump is no 3"}}}}
```

The initial states for respectively a pipe, a flow meter and a pump. Note the `rw_cmd`.

This is a function (often a closure) to execute command in reality. For the pump instance, this will open and close the relay that operates the real-world pump. When this parameter is a closure, it knows any addresses and/or specific requirements for its specific instance.

For the flow meter, rw_cmd is used to read out the measurement values. These connections to real-world actuation and sensing implements a device driver, which –importantly – brings the device functionality without involvement with the application.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

16) Enter in the shell `testRes:exec(g).`

```
>> {ok,22}
```

Testing rw_cmd with a dummy function.

17)Enter in the shell `testRes:exec(h).`

```
>> {{ok,[]},
 {ok,[measure_flow]},
 {ok,[switch_on,switch_off,is_on]}}
```

Get the operations supported by the resource instances. Pipes can't do anything, flow meters measure and pump can indicate their state, switch on or switch off.

18)Enter in the shell `testRes:exec(i).`

```
>> {ok,off}
```

Checking the pump is switched off initially.

19)Enter in the shell `testRes:exec(j).`

```
>> {{ok,switchOnIssued},{ok,on}}
```

Issuing a swtich_on to the pump, receiving confirmation that the command has been issued to the physical pump, observing the adapted state representation which reflects the execution of this command.

20)Enter in the shell `testRes:exec(k).`

```
>> {{ok,switchOnIssued},{ok,on},{ok,switchOffIssued},{ok,off}}
```

Swiching pump on and back off.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

**Project STORY - H2020-LCE-2014-3**

21)Enter in the shell `testRes:exec(l).`

```
>> {{ok,#Fun<pumpTyp.0.65846145>},
 {ok,#Fun<pumpTyp.0.65846145>}}
```

Verifying that the pump influence the flow of the fluidum depending on its state (on or off).

Enter in the shell `{{ _ , Fon }, { _ , Foff}} = testRes:exec(l).`

Next enter `Fon(9).` >> 43

Also enter `Foff(9).` >> 0

22)Enter in the shell `testRes:exec(m).`

```
>> {{ok,#{circuit =>
        {<0.3478.0>,
         #{<0.3469.0> => processed,<0.3470.0> => processed,
           <0.3473.0> => processed,<0.3474.0> => processed,
           <0.3477.0> => processed,<0.3478.0> => processed}},
      resInst => <0.3480.0>,root => <0.2134.0>,
      rootConnector => <0.3478.0>,
      typeOptions => "Fluidum is no 4"}},
 {{ok,[<0.3469.0>,<0.3470.0>]},
  {ok,[<0.3473.0>,<0.3474.0>]},
  {ok,[<0.3477.0>,<0.3478.0>]}}}
```

A fluidum instance has been created. Its initial state includes the circuit, which was discovered starting from a single connector from the circuit.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Project STORY - H2020-LCE-2014-3

23)Enter in the shell `testRes:exec(n).`

```
>> {{ok,[]},
 {ok,[]},
 {ok,<0.2134.0>},
 {ok,<0.3497.0>},
 {ok,[get_resource_circuit,estimate_flow]}}
```

The fluidum instance has no connectors, no locations/chambers, has a root and type, and supports two operations. It will provide the PID of the resource instance in its circuit and it is able to compute a flow rate estimate (based on the influence functions provided by the types of those instances).

24)Enter in the shell `testRes:exec(o).`

```
>> {{ok,#{<0.3503.0> => processed,<0.3507.0> => processed,
    <0.3511.0> => processed}},
 {<0.3503.0>,<0.3507.0>,<0.3511.0>}}
```

Fluidum instance delivers the PID of the instances in its circuit.

25)Enter in the shell `testRes:exec(p).`

```
>> {ok,9.9609375}
```

Fluidum instance computes a flow estimation.

This completes the first set of code modules for resources.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

## 5 Activities – NEU Protocol

This part covers the code corresponding to activities in a smart energy installation.

First, compile and load the additional modules. Enter into the shell:

```
c(activity_instance).

c(activity_type).

c(baseActInst).

c(circulateActTyp).

c(testAct).

c(circulateDM).

c(dummy_root).

c(root).
```

The software modules related to activities are:

| Module File Name | Description |
|---|---|
| activity_type .erl | Abstract module – common functions for activity types |
| activity_instance.erl | Abstract module – common functions for activity instances |
| baseActInst.erl | Concrete module for basic activity instances |
| circulateActTyp.erl | Concrete module for an simple activity type for circulating the fluid in a circuit for a given duration (where the duration is provided by the decision maker of the activity instance) |
| circulateDM.erl | Ad hoc decision making modules for testing. As noted, these modules are not part of an in-depth interoperability platform. |
| dummy_root.erl | |
| root.erl | |
| testAct.erl | Module to test the above – further discussed below. |

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Project STORY - H2020-LCE-2014-3

The module testAct.erl exports the function exec. This function can be called in the shell by entering `testRes:exec(n).` where n ranges from 2...8.

Tests 2-6 first address the activity type (circulateActTyp).

Tests 7-8 address the activity instance (baseActInst).

1) Start with `testAct:exec(1).`

This answers {ok,switchOnIssued} checks that the pump instance is working.

2) Enter in the shell `testRes:exec(2).`

```
>> {<0.222.0>,
 #{actInst => <0.86.0>,
   decisionMaker => #Fun<testAct.2.35355095>,
   options => "circulate activity type", progress => notStarted,
   root => root}}
```

This is the initial state that the activity type will generate for an activity instance.

Notice that it includes a function (closure) for any decisions that need to be made. As an activity instance belong to the in-depth interoperability platform, any decision making is externalised. It is delegated to the 'decisionMaker' provided at the creation of the activity instance that requested this initial state to be generated by its type.

Notice that the activity is not yet started (progress => notStarted).

3) Enter in the shell `testRes:exec(3).`

```
>> {ok, switchOffIssued}
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

This checks that the activity is able to issue a command to a resource instance when requested by an activity instance. Note that issuing commands to resource instances requires technical knowhow. Therefore, an activity instance will delegate this to its type (and provide this activity type with the necessary information concerning the resource instance). Instance handle 'logistic matter'and have type execute accounting for the remaining 'technical matter.'

4) Enter in the shell `testRes:exec(4).`

```
>> {{ok,[]},
 {ok,[{switch_on, dummy_res_alloc}]},
 {ok,[{switch_off, dummy_res_alloc}]},
 {ok,[stop]},
 {ok,[{wait, dummy_res_alloc}]}}
```

This reveals that the activity type generates the correct list of "next operations" when presented with an activity instance state. In this simple example, the list contains only one operation. In general, there can be multiple possible next operations; technically, all operations in the list are possible. It is the activity instance, aware of resource availability and capacity, to select from the list. The types are only concerned with (technical) capabilities.

5) Enter in the shell `testRes:exec(5).`

```
>> {{ok,#{actInst => <0.86.0>,
    decisionMaker => #Fun<testAct.2.35355095>,
    options => "circulate activity type",progress => notStarted,
    root => root}},
 {ok,#{actInst => <0.86.0>,
    decisionMaker => #Fun<testAct.2.35355095>,
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Project STORY - H2020-LCE-2014-3

```
    options => "circulate activity type",
    progress => just_started,root => root}},
 {ok,#{actInst => <0.86.0>,
    decisionMaker => #Fun<testAct.2.35355095>,
    options => "circulate activity type",
    progress => switchOnIssued, root => root}},
 {ok,#{actInst => <0.86.0>,
    decisionMaker => #Fun<testAct.2.35355095>,
    options => "circulate activity type",progress => wait_done,
    root => root}},
 {ok,#{actInst => <0.86.0>,
    decisionMaker => #Fun<testAct.2.35355095>,
    options => "circulate activity type",
    progress => switchOffIssued,root => root}}}
```

This shows how the activity type updates the state representation of an activity instance, given the report/result of the operation that was executed and the state representation before the start of this operation. The activity instance will use this new state representation to ask its type for an updated next-operations-list. This cycle corresponds to the NEU interaction protocol. The type computes what can be done next, technically correct. The instance arranges for one of the options offered by its type to be executed. The type updates the state based on the reporting of what has been done. And the cycle repeats (with 'next'): next-execute-update-next-execute-update-...

Note that this allows the activity instance to remain ignorant of technical matter while the type remains ignorant of logistical matter. Note also that the type is unaware whether it is being used for real-world control or for simulation purposes.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

6) Enter in the shell `testRes:exec(6).`

```
>> {{ok,[]},
 {ok,[{switch_on,dummy_res_alloc}]},
 {ok,[{wait,dummy_res_alloc}]},
 {ok,[{switch_off,dummy_res_alloc}]},
 {ok,[stop]}}
```

This test has the activity type execute as the activity instance will (should) do. In other words, this test ensures that when test(7) or test(8) fails, the error is within the code for the activity instance.

7) Enter in the shell `testRes:exec(7).`

```
>> <0.250.0>{#Fun<testAct.2.35355095>,
 {{ok,<0.86.0>},
  {ok,<0.249.0>},
  {ok,[]},
  {ok,#{actInst => <0.250.0>,
      decisionMaker => #Fun<testAct.2.35355095>,
      options => "circulate activity type",progress => notStarted,
      root => <0.86.0>}},
  {ok,none},
  {ok,no_ops_supported}}}...
```

This checks that the activity instance is instantiated correctly and that it provides the services required for the generic activity instance.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

8) Enter in the shell `testRes:exec(8).`

```
>>
 Start received and next op is : {switch_on,<0.254.0>}
  Cmd received : {<0.254.0>,switch_on}
  Cmd issued : {<0.254.0>,switch_on}
  Report received : switchOnIssued
  next op : {wait,2000}
  Waiting mode entered / /
  wait_done received and next op is : {switch_off,<0.254.0>}
  Cmd received : {<0.254.0>,switch_off}
  Cmd issued : {<0.254.0>,switch_off}
  Report received : switchOffIssued
  next op : stop
  circulate activity ends normally
 ok
```

This test show that the activity instance interacts with its type and the pump resource instance in the correct manner.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

# 6 Aggregates – performers – resource allocation

Performers aggregate digital twins (of resource types and instances, activity types and instances), which are in-depth interoperable, with their decision makers. Performers therefore build on the platform (by aggregating digital twins) but add potentially harmful constraints through their decision making constituents.

Explicit resource management is a generic mechanism to contain the inertia (= effort needed to remove, disable or adapt these decision making parts in a performer) of these potentially undesirable limitations imposed by a performer. Worst case, it disables the services of the performer fully. Often, it only disables services that have their allocation rights revoked.

Before performers can be discussed, some digital twins need an upgrade.

## 6.1 Managed resources

The above digital twin of a basic resource instance (baseResInst.erl) receives an upgrade. The upgraded module (baseResInstMgt.erl) is equipped with resource allocation handling services, publish-subscribe services and checked for access to the time basis. The latter proved to be non-existent and thus did not require any changes. Publish-subscribe services are needed because resource allocation management restricts access and allow external software to decide how 'verbose' the twin is.

Publish-subscribe services allow for external parties to receive notifications when the resource instance changes state or executes a command. This upgrade is implemented as follows:

- The parameter list of the main loop is extended with a subscriber list (i.e. process identifiers of the subscribers) and PubSubMsg, which is the information that is to be published.
- Whenever the main loop is executed, PubSubMsg is sent to all subscribers unless the message is 'nothing2tell' and nothing is sent.
- Each command that is executed decides what to publish. In this module, only the actual operations report. Monitoring and observation commands report nothing. In more sophisticated versions, the resource type may influence what is published.
- The loop is extended to allow subscribing and unsubscribing.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Explicit resource allocation support has been added. This is needed to contain the inertia of the decision making element in enclosing performers. When there is a conflict, the resource instance can be reconfigured to deny access for the cause of the problem and allow access to whatever provides what is needed. This upgrade goes as follows:

- At creation, the resource instance receives two tokens: SuperToken and UserToken.
- During initialization, rights are assigned. The UserToken has rights for all normal services. The SuperToken is allowed to change these rights.
- In the loop, the Token – provided in the message requesting a service to be executed – is checked for having suitable permission.
- Normal services, initially, require the UserToken. This is generated by the enclosing performer (see below).
- The SuperToken allows to change the Rights. An update will provide this SuperToken and a closure (function with context) to transform the existing Rights. It may deny the original user some rights and re-assign them to another Token. The SuperToken normally will be owned by the developer/provider of the installation.

This scheme can be further enhanced as needed (e.g. store encrypted versions of the tokens only, check through the Internet whether the SuperToken is valid, etc.). How much is needed depends on the specifics of the deployment context: who can(not) be trusted…

Now, compile and load the additional modules. Enter into the shell:

```
c(resource_instance_mgt).

c(baseResInstMgt).

c(testResMgt).
```

1) Enter in the shell `testResMgt:exec(0).`

```
>> {true,false,false,true}
```

This checks to (in)validity of Tokens. UserToken has rights for normal operations. SuperToken has rights to adapt rights but not to do the normal work.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Project STORY - H2020-LCE-2014-3

2) Enter in the shell `testResMgt:exec(1).`

```
>> {ok,switchOnIssued}
```

This indicated UserToken was allowed to switch the pump on.

3) Continue by entering in the shell `flush().`

```
>>

Shell got {exec_op,switch_on,

             #{cList => [<0.1360.0>,<0.1361.0>],

              chambers => [<0.1359.0>],

              on_or_off => off,resInst => <0.1358.0>,

              rw_cmd => #Fun<testResMgt.1.124040693>,

              typeOptions => {<0.1050.0>,"Pump is no 3"}}}

Shell got {op_excuted,{#{cList => [<0.1360.0>,<0.1361.0>],

                chambers => [<0.1359.0>],

                on_or_off => off,resInst => <0.1358.0>,

                rw_cmd => #Fun<testResMgt.1.124040693>,

                typeOptions => {<0.1050.0>,"Pump is no 3"}},

               on,on},

              #{cList => [<0.1360.0>,<0.1361.0>],

               chambers => [<0.1359.0>],

               on_or_off => on,resInst => <0.1358.0>,

               rw_cmd => #Fun<testResMgt.1.124040693>,

               typeOptions => {<0.1050.0>,"Pump is no 3"}}}

ok
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

The flush() command empties the mailbox of the shell. The shell was subscribed to the publisher-subscribe service of the pump PI3. It is informed of the issuing of a switch_on command and of the subsequent updating of the resource state when confirmation of the execution of this command is received. The subscribers receive all the information concerned, respectively the command issued with the state information and the 'report of the executing of the command with the state before' with the updated state representation.

Such a subscriber could be a maintenance application, monitoring how often the pump is used. It may initiate planned maintenance and/or issue warnings when the pump remains idle for too long. In the embedded simulation mode, the meta-control subscribes (and filters out) key information for assessing the future system performance and safety – assuming current control setting remain unchanged.

4) Enter in the shell `testResMgt:exec(2) and flush() once more`

```
>> <<178,137,137,75>>
```

This is the ObserverToken, which will change with every execution.

```
>> Shell got {exec_op,switch_on,
          #{cList => [<0.1377.0>,<0.1378.0>],
           chambers => [<0.1376.0>],
           on_or_off => off,resInst => <0.1375.0>,
           rw_cmd => #Fun<testResMgt.1.124040693>,
           typeOptions => {<0.1050.0>,"Pump is no 3"}}}
Shell got {op_excuted,{#{cList => [<0.1377.0>,<0.1378.0>],
             chambers => [<0.1376.0>],
             on_or_off => off,resInst => <0.1375.0>,
             rw_cmd => #Fun<testResMgt.1.124040693>,
             typeOptions => {<0.1050.0>,"Pump is no 3"}},
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Project STORY - H2020-LCE-2014-3

```
                 on,on},
                #{cList => [<0.1377.0>,<0.1378.0>],
                  chambers => [<0.1376.0>],
                  on_or_off => on,resInst => <0.1375.0>,
                  rw_cmd => #Fun<testResMgt.1.124040693>,
                  typeOptions => {<0.1050.0>,"Pump is no 3"}}}
ok
```

This is the same output as before. However, this time SuperToken was used to give permission to an newly generated ObserverToken to observe the pump PI3. SuperToken provides a function that receives the current right assignments and extends them. Note that SuperToken may equally revoke rights. Note also that ObserverToken has an expire time.

## 6.2 Managed activities

The above digital twin receives an upgrade. It needs to present a suitable Token when accessing resource services. It gets a publish-subscribe service. And, all access to time is channelled through a module 'my_time' that allows for speed-up or slow-down by a given amount (for simulation). More sophisticated activity probably need a rights management service (to stop or influence long-lasting activities).

Compile and load the additional modules. Enter into the shell:

```
c(activity_instance_mgt).

c(baseActInstMgt).

c(my_time).

c(testActMgt).
```

1) Enter in the shell `testActMgt:exec(1) and flush()`

```
>> Shell got {exec_op,switch_on,
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

```
          #{cList => [<0.4008.0>,<0.4009.0>],
           chambers => [<0.4007.0>],
           on_or_off => off,resInst => <0.4003.0>,
           rw_cmd => #Fun<testActMgt.1.4660300>,
           typeOptions => {<0.3320.0>,"Pump is no 3"}}}
Shell got {op_excuted,{#{cList => [<0.4008.0>,<0.4009.0>],
             chambers => [<0.4007.0>],
             on_or_off => off,resInst => <0.4003.0>,
             rw_cmd => #Fun<testActMgt.1.4660300>,
             typeOptions => {<0.3320.0>,"Pump is no 3"}},
            on,switch_on},
           #{cList => [<0.4008.0>,<0.4009.0>],
            chambers => [<0.4007.0>],
            on_or_off => on,resInst => <0.4003.0>,
            rw_cmd => #Fun<testActMgt.1.4660300>,
            typeOptions => {<0.3320.0>,"Pump is no 3"}}}
Shell got {exec_op,switch_off,
          #{cList => [<0.4008.0>,<0.4009.0>],
           chambers => [<0.4007.0>],
           on_or_off => on,resInst => <0.4003.0>,
           rw_cmd => #Fun<testActMgt.1.4660300>,
           typeOptions => {<0.3320.0>,"Pump is no 3"}}}
Shell got {op_excuted,{#{cList => [<0.4008.0>,<0.4009.0>],
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

```
                chambers => [<0.4007.0>],
                on_or_off => on,resInst => <0.4003.0>,
                rw_cmd => #Fun<testActMgt.1.4660300>,
                typeOptions => {<0.3320.0>,"Pump is no 3"}},
              off,switch_off},
             #{cList => [<0.4008.0>,<0.4009.0>],
               chambers => [<0.4007.0>],
               on_or_off => off,resInst => <0.4003.0>,
               rw_cmd => #Fun<testActMgt.1.4660300>,
               typeOptions => {<0.3320.0>,"Pump is no 3"}}}
ok
```

This shows how a subscriber to the pump instance see how the activity is executed.

2) Enter in the shell `testActMgt:exec(2) and flush()`

```
>> Shell got {"Started and next op is : ",{switch_on,<0.4014.0>}}
Shell got {{switch_on,issued_for,<0.4014.0>},
           {report_received,switchOnIssued},
           {next_op,{wait,2000}}}
Shell got {"Waiting mode entered : ",2000}
Shell got {"Wait_done and next op is : ",{switch_off,<0.4014.0>}}
Shell got {{switch_off,issued_for,<0.4014.0>},
           {report_received,switchOffIssued},
           {next_op,stop}}
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Shell got activity_done

ok

This reveal how a subscriber to the activity instance sees the NEU protocol being executed.

## 6.3 Performers

Functional smart energy installations need to combine digital twins of resource and activities with decision making mechanisms. Basically, they encapsulate what has been provided in the above test modules for resources and activities already (Note how the testPerf module has not much to do). The sample module ensures that the resources, which it aggregates, are accessed only with suitable permission (i.e. with UserToken).

Moreover, it allows access to its embedded resources when presented with the SuperToken. SuperToken access to these embedded resources allows to re-allocate these embedded resources, which is useful when the decision making mechanisms inside the performer cause interoperability conflicts. Note that the SuperToken for the performer and for the embedded resources can be different. Typically, the provider (i.e. warranty provider) for the performer and/or resources owns the SuperToken, not their user.

Compile and load the additional modules. Enter into the shell:

```
c(simplePerformer).

c(simpleDM).

c(testPerf).
```

1) Enter in the shell `testPerf:exec(1) and flush()`

```
>> Shell got {exec_op, switch_on,

          #{cList => [<0.4081.0>,<0.4082.0>],

           chambers => [<0.4080.0>],

           on_or_off => off,resInst => <0.4079.0>,
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Project STORY - H2020-LCE-2014-3

```
                rw_cmd => #Fun<simplePerformer.1.40026789>,
                typeOptions => {<0.4067.0>,"Pump is no 3"}}}
Shell got {op_excuted,{#{cList => [<0.4081.0>,<0.4082.0>],
                  chambers => [<0.4080.0>],
                  on_or_off => off,resInst => <0.4079.0>,
                  rw_cmd => #Fun<simplePerformer.1.40026789>,
                  typeOptions => {<0.4067.0>,"Pump is no 3"}},
                 on,switch_on},
                #{cList => [<0.4081.0>,<0.4082.0>],
                 chambers => [<0.4080.0>],
                 on_or_off => on,resInst => <0.4079.0>,
                 rw_cmd => #Fun<simplePerformer.1.40026789>,
                 typeOptions => {<0.4067.0>,"Pump is no 3"}}}
ok
```

2) Wait for 5 seconds and enter again in the shell `flush()`

```
Shell got {exec_op,switch_off,
               #{cList => [<0.4081.0>,<0.4082.0>],
                chambers => [<0.4080.0>],
                on_or_off => on,resInst => <0.4079.0>,
                rw_cmd => #Fun<simplePerformer.1.40026789>,
                typeOptions => {<0.4067.0>,"Pump is no 3"}}}
Shell got {op_excuted,{#{cList => [<0.4081.0>,<0.4082.0>],
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

```
                  chambers => [<0.4080.0>],
                  on_or_off => on,resInst => <0.4079.0>,
                  rw_cmd => #Fun<simplePerformer.1.40026789>,
                  typeOptions => {<0.4067.0>,"Pump is no 3"}},
                 off,switch_off},
                #{cList => [<0.4081.0>,<0.4082.0>],
                 chambers => [<0.4080.0>],
                 on_or_off => off,resInst => <0.4079.0>,
                 rw_cmd => #Fun<simplePerformer.1.40026789>,
                 typeOptions => {<0.4067.0>,"Pump is no 3"}}}
ok
```

The test program creates a simple performer, used the SuperToken to get the Pid of the embedded resources. Next, the shell subscribes to the publish-subscribe service of the pump in the circuit. When the circulate command is issued to the performers, the shell receives the corresponding information from the pump.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

# 7 Simulation

The digital twins in ordinary control mode already mirror their real-world counterpart. This brings as a bonus, almost for free, the ability to simulate. Moreover, it is a 'software-in-the-loop' simulation as the control system software remains unchanged when transitioning from simulation to real-world deployment. The only changes are the device driver functions, which are parameters for the creation of resource instance twins. E.g. the device driver for the flow meter – in the control – has the digital twin of the fluid in the simulation compute/estimate the flow rate. Note: FI is the digital twin of the fluid in the circuit in the simulation; CtrlPI2 is the digital twin of the flow meter in the control.

```
{ok, FT} = fluidumTyp:create(),

{ok, FI} = baseResInst:create(self(), FT, {Out3, "Fluidum is no 4" }),

...

FMfun = fun() ->

          Rx = resource_instance:exec_op(FI, estimate_flow),

               ...

          end,

{ok, CtrlPI2} = baseResInst:create( ... , { FMfun , ... } )
```

For the pump twin, it even more straightforward: the control twin forwards the command to the simulation twin.

```
PuFun = fun(OnOrOff) ->

          resource_instance:exec_op(PI3, OnOrOff)

          end,

{ok, CtrlPI3} = baseResInst:create( ... , {PuFun, ... }),
```

Compile and load the simulation demo. Enter into the shell:

c(simDemo).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

Note all the existing software modules remain unchanged.

Next, enter in the shell `simDemo:start(1).`

```
>> {{ok,9.9609375},
 {ok,9.9609375},
 {ok,9.9609375},
 {ok,0.0390625},
 {ok,0.0390625},
 {ok,0.0390625}
```

The output shows that when the pump control twin receives a switch_on command, the flow estimate from the digital twins of respectively the fluid in the control, the fluid in the simulation and the flow meter in the control agree. Likewise, when the control pump twin receives switch-off, all 3 twins agree on the flow being practically zero (note: the flow estimation algorithm stops when the balance of the flow influence along the circuit is within flow meter accuracy (of 2 decimal digits)).

Note that the device drivers, provided to the control twins, can be modified to make the simulation more realistic, more challenging, mirroring extreme situations, ... E.g. the commands to a pump may fail, flow estimates may be modified to simulate measurement errors and/or flow meter malfunctioning.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

# 8 Embedded Simulation

When translating the scientific results, which originally were targeting manufacturing execution systems, logistic execution systems, robotic fleet management and intelligent traffic, a number of key insights still had to be acquired to get to the present results/achievements.

First, in energy everything is connected. When one part of the system goes berserk, others suffer. And, there have been major blackouts of large electricity grids to witness this. Here, the key insight is that novelty in smart energy will encounter fierce resistance unless it convincingly contributes to system safety and integrity. The answer is to focus the developments on health monitoring and diagnostics (by designing digital twins that are able to this in a kind of shadow mode while a conventional control remains in charge).

In addition, this 'everything being connected' prevented a straightforward translation of the embedded forecasting provided in the scientific results. E.g. in intelligent traffic, traveller twins would virtually execute their intended journeys, informing road infrastructure twins of future loads. Congestion would become known well before it occurred and travellers could adjust their plans. Nervousness control mechanisms dampen this changing of traveller intentions to keep forecasts usable. Indeed, those forecasts are used by traveller twins when exploring possible journeys (to improve the current plans and/or to add another trip).

This would not work for energy applications. Twins of individual activities could not inform twins of individual resources to enable the overall system to predict future system states. Every change would require to compute and estimation for the overall system. At first, the idea of generating embedded short-term forecast was put on hold. Nonetheless, it remained a high-value target, among others because it may allow to handle critical situations on beforehand (and avoid tough real-time issues).

Second, the above-discussed simulation mode was developed. Linear speed-up of this simulation mode revealed to be simple to implement. More ambitious speed-ups are possible but require suitable models in the digital twins (e.g. determine when the next event will be and update state models for variable time intervals); it remains doubtful that this makes economic sense except for niches. This revealed that it is possible to include one or more simulations in a control system.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

In other words, the short-term forecasting became available as a bonus. However, this did not translate the scientific results toward smart energy in full, which used the short-term forecasts to improve control decisions and actions (e.g. adapt when predicted to get stuck in a traffic jam).

This required a third and fourth insight. Thirdly, an innovative publication review procedure, applied to keynote paper on the scientific results, allowed to draw the correct boundary between the 'in-depth interoperability platform' and the applications executing on the platform. The 'untranslatable parts', which were considered services of the platform, imposed a specific structure on the decision making elements. In the innovative review procedure, reviewers pointed out that this was an unjustified requirement toward research on the decision making. This specific structure was no more than an option. The proper requirement was that the decision making had to be 'grounded'(i.e. down to the specific commands issued to the equipment, workers, ...).

This insight translates into significantly more liberty and responsibility for the decision making in the control system. Consequently, it is the responsibility of the control design to manage and use the embedded simulations. The 'in-depth interoperability' is not obliged to offer any services in this regard (short of bonuses from what is already available). Nevertheless, this still raised the question whether this forecast-only represented significant value.

This brings us to the fourth insight. Energy is – to a significant degree – a commodity. Therefore, the already-available embedded simulation suffices for a (meta-)control (level) to use it and adapt control settings/decision-making. In the scientific results, individual activity twins have specific needs, which need accounting for when adapting plans. Smart energy enjoys a different situation, which is more comfortable in some ways (and, as discussed, more challenging in others aspects).

This embedded simulation can be used in three manners. The simplest manner is to have a look-ahead to see problems (e.g. overheating) and missed opportunities early. The optimising manner is to have multiple embedded simulations – each with another control and/or simulation setting – to detect which options to select and execute.

Finally, the embedded simulation can be shared among multiple control systems. This illustrates a key insight into in-depth interoperability: reality (and its mirror image in simulation) is coherent and consistent. This informs each of the control systems of their joint effect and impact on the energy

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

installation on beforehand. Here, the individual controls may adapt (with some nervousness control) based on the forecast. Alternatively, they may initiate peer-to-peer collaboration or employ some coordination services.

Overall, the development in STORY succeeded in keeping the software base compact (one module for a real-world counterpart or less) and single-source-of-truth (no need for different versions for control or simulation). But the road to this result has been tortuous, involving modules and structures that revealed to be replaceable by less and simpler ones in the end.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

## 9 Remarks

Task 3.4.2 resulted in this deliverable D3.7. It translates scientific results out of the domain of manufacturing, which had been translated already into the domains of intelligent traffic, logistics and robotics. Initial steps into healthcare have been made as well. Indeed, the scope of the scientific results is ‘activities executing on resources’ where ‘execution-time smartness’ is desirable.

The translation into the smart energy domain presented some unique challenges. The system-wide impact of actions prevented a straightforward transfer of the embedded forecasting, available in the other domains. The near-commodity nature of energy came to the rescue in combination with a correction of the boundary between an in-depth platform and decision-making applications on top of it. The challenge forced the generation of short-term forecasts to be system-wide. The near-commodity nature allows starting from such a full-system forecast to decide about adaptations that handle issues before they become problematic and grasp opportunities with minimal effort.

After making the unavoidable detours (e.g. designing, coding and testing software that revealed to be replaceable by simpler and less software), the final result is compact and does not require to manually maintain consistency among software modules. The result (D3.7) will be maintained by translating it into Elixir (not in STORY; in a UCL project on digital twins).

The motivation for the switch is the ‘war for talent’. Technically, Elixir offers everything Erlang has to offer. However, Elixir brings so-called “programmer happiness” (think of Python) whereas Erlang has an outdated appearance (think of C) around its cutting-edge core. Equally important, the Elixir community is newbie-friendly and provides state-of-the-art software tools. This translation will be deployment-oriented and available to the STORY participants (i.e. open source).

Finally, translation from one domain to another has revealed that each domain has its own culture, and this culture determines how novelty can(not) be introduced. Because subsystems going berserk create havoc well and far beyond the own subsystem, the energy community has developed a kind of community immune system, like human bodies. When novelty does not prove to be safe from the start, an aggressive response follows (e.g. the same reviewer calls something really innovative but dangerous and rejects even for a forum calling for embryonic research results). Fortunately, D3.7 is easily applied in a shadow mode, increasing safety by health monitoring and diagnostics. In this mode of operation, it can mature and prove itself before attempting to participate in actual control.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 646426

# 10 Acronyms and terms

| | | |
|---|---|---|
| ARTI | Activity-Resource-Type-Instance | Reality-centric reference architecture |
| NEU | Next-Execute-Update protocol | Type-Instance interaction model |

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426

## 11 References

[1] Valckenaers, Paul, Van Brussel, Hendrik. Design for the Unexpected: From Holonic Manufacturing Systems towards a Humane Mechatronic Society. Butterworth-Heinemann[4], November 2015.

[2] Valckenaers P., De Mazière P.A. (2015) Interacting Holons in Evolvable Execution Systems: The NEU Protocol. In: Mařík V., Schirrmann A., Trentesaux D., Vrba P. (eds) Industrial Applications of Holonic and Multi-Agent Systems. HoloMAS 2015. Lecture Notes in Computer Science, vol 9266. Springer.

[4] store.elsevier.com/product.jsp?isbn=9780128036624

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 646426