Project Number: 257909

SPRINT

Software Platform for Integration of Engineering and Things

Collaborative project
Information and Communication Technologies

D4.10 Contract and Sensor and Actuator integration services definition

Start date of project: October, 1st 2010
Duration: 42 months

Deliverable	Contract and Sensor and Actuator integration services definition		
Confidentiality	PU	**Deliverable type**	R
Project	SPRINT	**Date**	2014-02-07
Status	FINAL	**Version**	1.0
Contact Person	Michael Wagner	**Organisation**	Fraunhofer FOKUS
Phone	+49 30 3463 7391	**E-Mail**	Michael.Wagner @fokus.fraunhofer.de

THE RESEARCH LEADING TO THESE RESULTS HAS RECEIVED FUNDING FROM THE SVENTH FRAMEWORK PROGRAMME UNDER GRANT AGREEMENT N° 257909

AUTHORS TABLE

Name	Company	E-Mail
Michael Wagner	Fraunhofer FOKUS	Michael.Wagner@fokus.fraunhofer.de
Björn Riemer	Fraunhofer FOKUS	Bjoern.Riemer@fokus.fraunhofer.de
Massimiliano Dangelo	ALES s.r.l.	massimiliano.dangelo@ales.eu.com

CHANGE HISTORY

Version	Date	Reason for Change	Pages Affected
1.0	2014-02-07	Final Version	all

CONTENT

Content of Figures

Content of Tables

Content of Appendix

None.

1 Introduction

This document is the final public version of the restricted "D4.9 Contract and Sensor and Actuator integration services definition" to be made public at the end of the project.

1.1 Overview and Purpose of the Document

The purpose of this document is to identify the context for the integration of contracts, sensors and actuators (physical devices). Here the focus of the document is especially on integration during design and simulation (verification) time and the different deployments with regard to physical devices. Finally the document specifies the integration service for sensors, actuators and contracts.

This document is based on the input from

- D.2.1 SPRINT Requirements
- D.4.1 Foundation of Contract for Things
- D.4.3 Contract Based Verification Methods

and conforms to the input from

- D.5.4 Architectural principles for Internet of System Design and the IoT

1.2 Scope of the Document

The scope of the document is on the definition of the integration service for sensors, actuators and contracts and the information necessary for the context.

What is inside the scope of the document:

- Overview on contracts, sensors and actuators
- Integration context during design time
- Integration context during simulation time
- Contracts, sensors and actuators integration service definition

2 SPRINT Requirements

This section contains all relevant requirement input for this document. This includes high level requirements, use cases and technical requirements.

2.1 High Level Requirements

This section lists all in WP2 identified requirements relevant for this document.

ID	Description	
6.1.5	The SPRINT Engineering Environment shall allow project management to manage access rights to project data, including, but not limited to, modelling data, requirements, and test and analysis results.	
6.1.8	The SPRINT platform shall adopt standard or de-facto standard technologies wherever available.	
6.1.9	The communication among the components of the SPRINT platform shall be based on RESTful web services.	
6.2.2	Meta-models description shall support standards technologies such as EMF (Eclipse Modelling Framework), Ecore, Resource Description Framework (RDF) and (Web Ontology Language) OWL.	
6.2.3	The semantic integration of design models shall support a 'light' version of HRC.	
6.2.4	Shared Resources shall have the following services available to them :(1) Queries(2) Administration(3) Storage	
6.2.5	All resources shall be accessible via a URI and have a single owner.	
6.3.2	Simulation tools in SPRINT shall support interleaving semantics for the components execution.	
6.3.3	Design element shall be exchanged between tools by having their resources share the same semantic model.	
6.4.6	The integration of physical devices and System designs shall be done using semantic mediation.	
6.5.2	The SPRINT platform shall allow for the verification of contract satisfaction by using simulation and contract monitoring.	
6.5.3	The SPRINT platform shall allow for the verification of contract satisfaction by using contract monitoring of deployed physical devices.	
6.5.4	The SPRINT platform shall provide remote monitoring capabilities based on contracts.	
6.4.1	SPRINT Contract tools (authoring and analysis) shall support CSL (SPEEDS CSL or other agreed contract specification means)	
6.4.6	The integration of physical devices and System designs shall be done using semantic mediation.	
7.1.3	SPRINT Engineering Environments shall allow for co-simulation of models with HiL (Physical Devices)	
7.1.4	The communication among the components of the SPRINT platform shall be firewall friendly.	
8.1.1	SPRINT Engineering Environment shall allow for contract monitoring\analysis of Physical Devices.	
8.1.2	SPRINT Engineering Environment shall allow for remote monitoring of Physical Devices.	
8.1.3	Physical SPRINT devices shall be REST compliant.	
8.1.4	SPRINT physical devices shall communicate via the internet to SPRINT tools.	

Table 2-1: SPRINT requirements considered in this deliverable

2.2 Goal

The goal is that physical devices like sensors and actuators can be as easily integrated into all the phases of the developments as normal design artifacts (e.g. models). This includes the interplay with contracts and the contracts based verification method. This allows for faster identification of faulty system elements or assumptions.

2.3 Technological Requirements

The main technological requirements are already given through the requirements of D2.1:

- Rest full communication
- Fire wall friendly communication over the internet
- Support for integration via semantic mediation
- URI for accessing elements
- Allow for querying and browsing (basic internet capabilities)

3 Overview

This section gives an overview on sensors, actuators and contracts in order to better understand the challenges and innovation in the integration.

3.1 Sensors and Actuators

A sensor is device that converts physical values (like temperature, light intensity, or rotation) into an electrical measurable signal. To further process, store, or transmit the measured signals they need to be digitalized. This is typically done by an analog to digital converter and some kind or programmable control unit like an embedded microcontroller. Sensor devices are equipped with different amount of computing power, storage memory, and (communication-) interfaces, so they can be categorized in multiple classes like **small**, **medium**, and **large**.

Devices with the smallest amount computing power can simply measure values and output them via an electrical connection. If the output needs to be further processed or monitored for changes an additional computing devices needed. For example an RPM sensors in a car counts the rotations of the motor axis and transmits the values onto the CAN bus. A controller that receives the rotation count values from the CAN bus can calculate the RPM value. Monitoring rules would run on a gateway system which receives the measured rotation value from the can bus and checks if is meets the requirements.

Figure 1 ZigBee Wireless Temperture Sensor

Medium sized devices include some kind of controlling circuit that can be used to do simple calculations based on the measured values. These devices can include wireless radios to allow the transmission of values to remote monitoring devices. To reduce the amount of transmitted data the embedded controller implements some kind of watches that are used to generate an event message if the value changes below a certain threshold. A typical device for this class is a wireless temperature sensor that can be programmed to notify the wirelessly connected base station if the temperature reaches a predefined value.

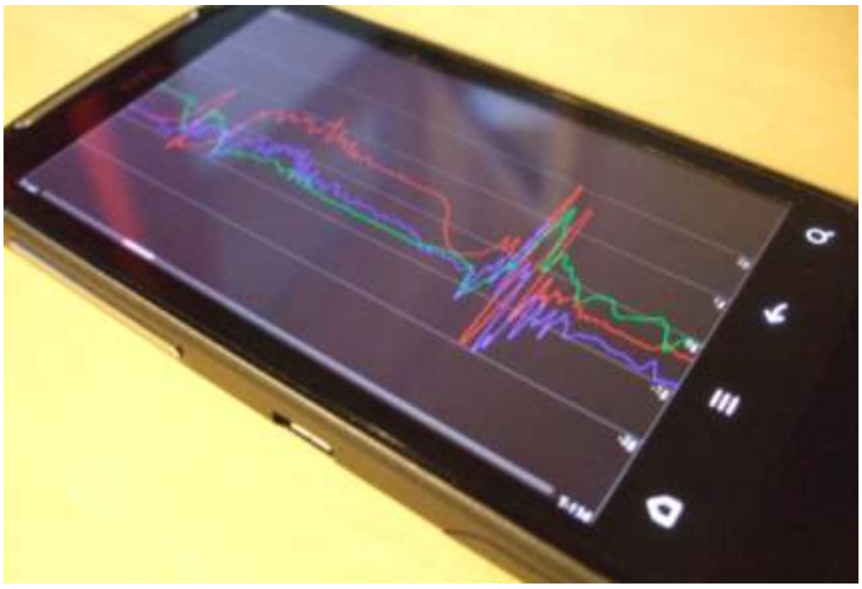

Figure 2 SmartPhone with Acceleration Measurement Application

The most powerful sensor devices include large amount of storage memory combined with powerful processors that can be used to run custom applications to process, store, or forward the measured values. Most likely will they also include a communication interface that provides network access to distributed networks like the Internet. These complex devices combine multiple sensors to measure different types of physical values that can be combined by computing algorithms. An example for this category of devices is a modern smartphone that includes among other things a GPS sensor, three axis acceleration sensors, and a magnetic compass. It can run custom monitoring applications that will combine the measurement all sensors to compute the orientation of the device and check the results against complex rules. The communication interface can be used to transmit the results to a remote monitoring or storage system.

In our context every device that can measure some kind of physical values or detect events and forwards this information to other systems can generally speaking be called a sensor.

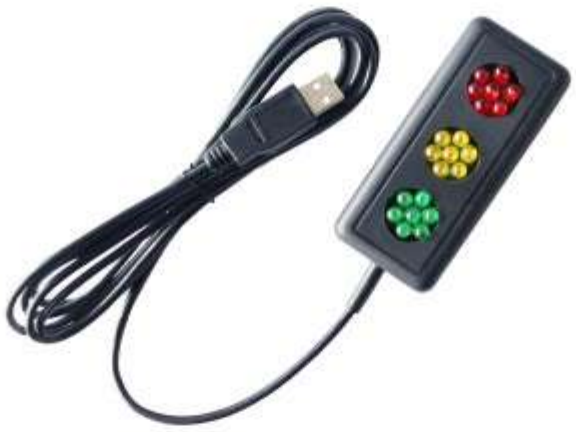

Figure 3 USB controlled Traffic Light

In contrast to sensors, actuators are devices that can generate physical values from an electrical input, like an electric motor generates rotational movement from an electric current. Actuators, like sensors, are also available in different configurations in regard to computing power, interfaces and storage capacity. In our context every device that can be programmed to output an arbitrary signal is called an actuator.

A very basic Actuator is for instance a traffic light that can be controlled by a serial line to display a combination of red, green, and yellow light signals. For this type of sensor no internal processing power is needed, the external controller directly selects the colors to display. Actuators that can be ordered to do a specific amount of work like a speed regulated motor controller need internal computing power to calculate the needed motor current for the requested RPM value.

Often sensors and actuators are combined into a single device like a movement sensor that on a detected movement will switch on a light and send a notification message to a remote monitoring system.

3.2 Contracts

A *contract* [D4.3] (Contract Based Verification Methods) is a specification of a component that defines the properties *G* that a component must be able to guarantee, and the context *A* or the *environments* under which the guarantee must be established. This *assumption* on the environment is what distinguishes a traditional component model from a contract model. When the assumptions are not satisfied, a component is free to behave as desired, potentially violating its own guarantees.

Verification usually consists in ensuring that each component satisfies a number of properties. In a contract-based design methodology, the availability of the pair of specifications (*A*, *G*) for each component opens up the possibility for checking additional conditions which are not limited to property checking. In particular, we are interested in two essential aspects:

- Satisfaction: the verification that a component satisfies its contract, i.e., that when placed in a context that meets the assumptions *A*, the component is able to guarantee the properties in *G*.
- Compatibility: the verification that components work well in combination, i.e., that the guarantees of one are able to meet the assumptions of the other, and vice-versa.

Establishing that components satisfy their contracts, and that contracts are mutually compatible, is sufficient to establish the correctness of the design with respect to the specification. This is because contracts express the partitioning of the design responsibilities between components, and the contract theory ensures that if contracts are compatible, and if components satisfy their contracts, then the interaction between the components will not raise exceptions due to assumption violations. In particular, compatibility of contracts is sufficient to show that the context of a specification was designed according to the specification requirements, and that the specification itself was designed according to its own guarantees, as well as the requirements of the surrounding context. In other words, compatibility shows that the design responsibilities have been distributed correctly, so that collectively they are able to ensure the global system properties. The relation of satisfaction is used to make sure that compatibility is retained at the level of the implementation. Splitting the verification in two steps is essential for an efficient methodology: compatibility, which amounts to a global property of the system, is checked on abstract specifications; satisfaction, on the other hand, is checked only locally for each component, while the global result is obtained by compositionality. This way, one avoids checking global properties on the collection of components, which are presumably much more complex that their specifications, therefore rendering the verification task potentially impractical.

There are different methods for contract verification. In SPRINT, we adopt the monitor-based contract verification. A monitor is a component that verifies at run-time a specific relation between its inputs. In the context of contract-based design, monitors have been successfully adopted for the run-time verification of satisfaction and compatibility relations. In the SPRINT project the monitors are automatically synthesized starting from a specification of both the component and the contracts as depicted in Figure 4 Monitor Synthesis Flow. The assumptions and promises are described using the BCL and a synthesis step is performed to translate them to executable specifications. Each component is then translated to a monitored component that exposes the same interface of the original, enriched by two additional outputs, called a and g of type $B_{\perp}$={TRUE, FALSE, UNDEFINED}: the a signal is true, false or undefined if the contract assumption is true, false or undefined respectively. Similarly the g signal exposes the acceptance of the promise. Using the two additional signals it is possible to evaluate at run-time the satisfaction and compatibility relations between the component implementation and the contract: defined as mi the component execution trace up to i-th simulation instant, the contract is satisfied if and only if (ai $\leftrightarrow$ gi) and compatibility imposes that for every i (ai $\leftrightarrow$ TRUE).

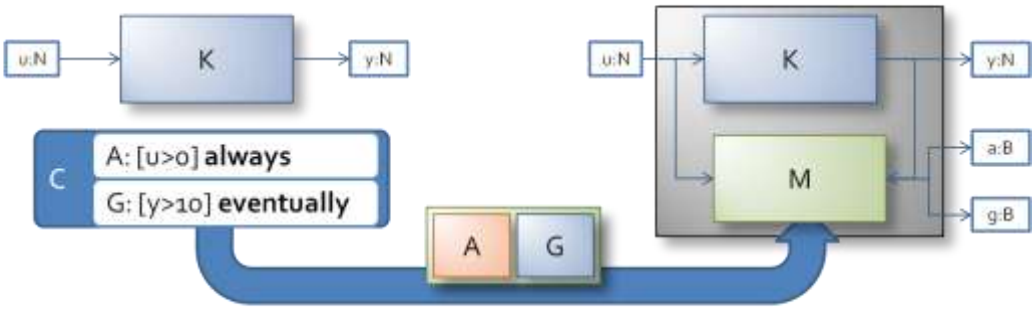

Figure 4 Monitor Synthesis Flow

Different monitoring scenarios are possible in case a physical device is part of the simulation environment and one or more contracts are attached to it. As already described in the deliverable D4.3 monitors must be executed synchronously with the corresponding components and it is responsibility of the simulation framework to ensure such a synchronicity. In the context of classical component monitoring, three possible monitoring scenarios have been defined where the monitors can be executed locally or remotely with the component. In the context of physical device monitoring a similar approach can be followed identifying two possible monitoring scenarios:

a) *local physical device monitoring*: the monitor is embedded in the physical device and executed as a device process
b) *remote monitoring*: the monitor is not executed with the device and it remotely checks the observable behaviour of the component

The two approaches have different strength and weakness and in particular a) can be employed when the computational power of the device is sufficient to execute the monitor without affecting the performance of the device. Note that in this case it could be partial responsibility of the monitor to communicate with the simulator in order to ensure a correct execution of the simulation. The remote monitoring b) can be applied without affecting the computational performance of the monitored component and the monitor relies on the synchronisation capabilities of the simulator in order to be executed synchronously with the device.

3.3 Challenges

The heterogeneity of physical devices may lead to ad-hoc integration so that an abstraction for commonalities is needed. This includes the identification of classes of physical device that can be treated in a similar way.
Another challenge is the synchronisation with the simulation environment since physical devices usually run in real world time and cannot be reset or have try-steps (what would happen if steps).
There is also the challenge on the deployment of the monitor. It might be necessary that monitor code has to be deployed on the device. Here the optimization to reduce the footprint of the monitor code and the automation of the flow in order to make the deployment usable poses a challenge.

4 Integration

Integration of sensors, actuators and contracts in the Sprint Engineering Environment has to happen on two levels. One level is during design time where the design artefacts are created. The other level is during simulation (and possible design validation/integration) time, where the interplay of the components if verified via simulation. Both levels have different requirements on the integration so that different integration approaches can be expected. These approaches are elaborated in detain in the following sections.

4.1 Integration during Design Time

It has to be possible to assign contracts during design time to sensors and actuators like for all the other components during design time. However, this poses no mayor problem since physical device types register as component types and the devices as their instances. In this case, the Sprint engineering Environment does not make a distinction between physical devices and other component oriented design artefacts.

However, physical devices behave different. Registration data is provided automatically to the engineering environment on registration and updated on change. There is no conflict on the data, since the data is only written by the physical device and otherwise read only for the tools.

Semantic mediation is utilized to identify device classes and to integrate the devices seamlessly in the sprint engineering environment and its internet capabilities like search, browse and view. Here the nature of the low level device integration utilizing the same underlying technology (REST, HTTP, and RDF) is of advantage. Since the semantic mediation itself can stay agnostic to the presence of physical devices.

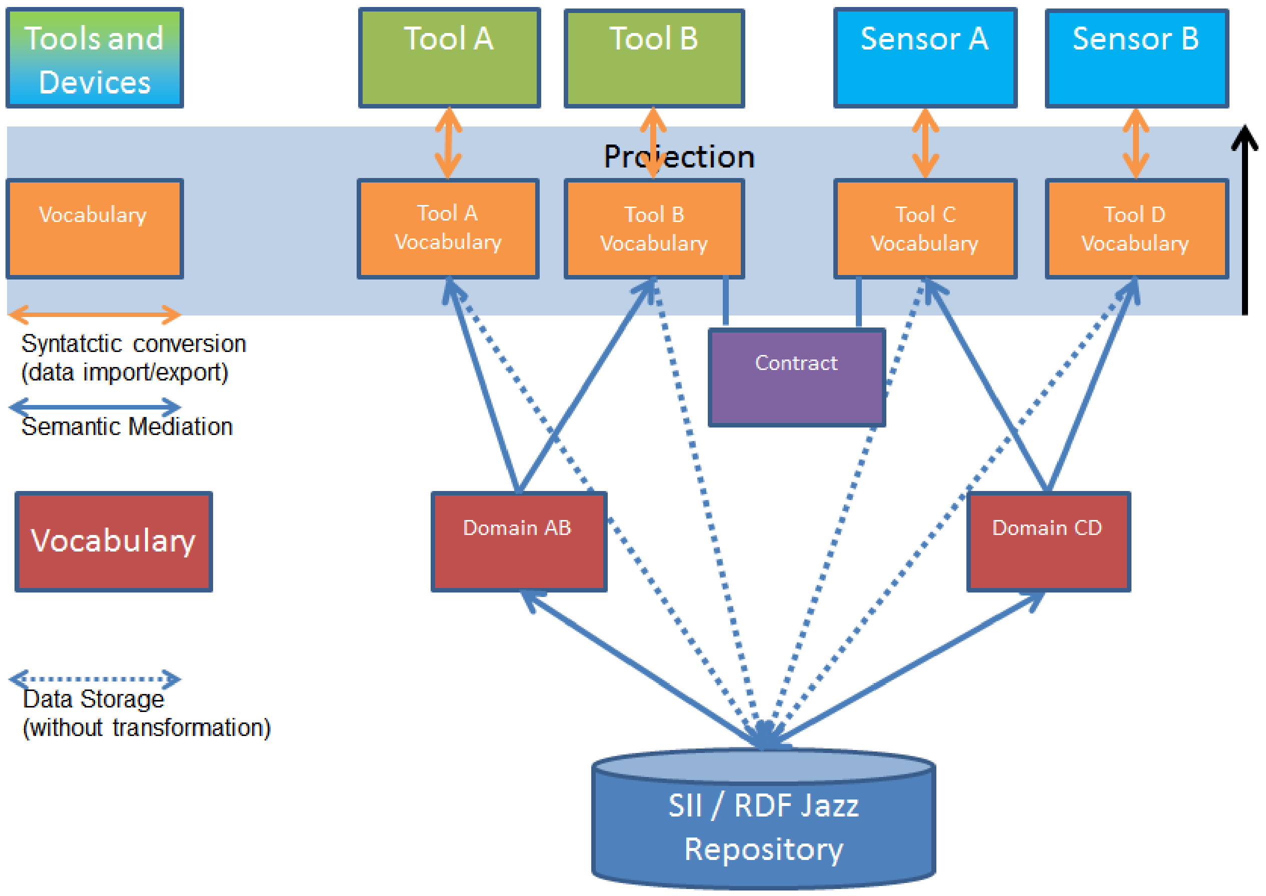

Figure 5 Integration at Design Time

4.2 Integration during Simulation Time

Integration during simulation time is important in order to allow verification of the design through simulation, possible backed through test cases steering the simulation. This would allow identifying faulty components which violate contracts during simulation time.

The challenge of simulation in the Internet of System Engineering is the distribution over the internet. The Sprint Engineering Environment as one implementation of the Internet of System Engineering has to allow simulation of different components[1] across the internet [D4.15] (Real Time Service Definition). A simulator is responsible for coordinating the orchestrated simulation of the different components over the internet[2].

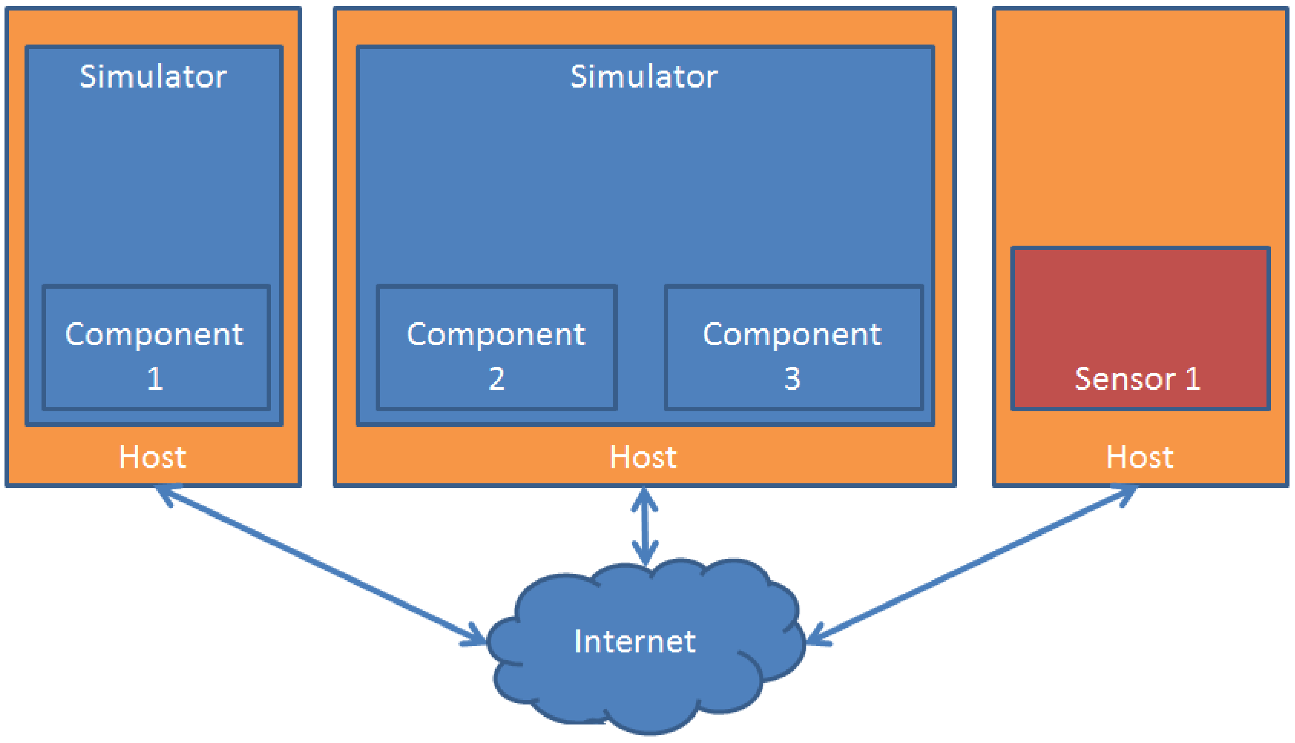

Figure 6 Overview Integration during Simulation Time

The preferred mechanism to check consistency during simulation is through contract monitoring (see [4.3]). Placing a monitor and the corresponding contract on a component during simulation time allows checking for conformant behaviour. In At this level sensors and actuators are treated as components in the first place. The different deployment scenarios for contract monitoring in combination with sensors and actuators are described in detail in the following subsections.

4.2.1 Monitoring on Master Simulator

The most obvious and straight forward approach is to place all the monitors in the coordinating simulator. The simulator has everything he needs to monitor all the contracts since he receives all the communication of the component (see Figure 7 Monitoring on Simulator). In this case the components, sensors and

[1] This can be designed components from different tools or physical devices (e.g. sensors or actuators)
[2] A possibility is also to use hosted simulation in a cloud environment utilising this internet technology to achieve higher performance. However, the remoteness of the physical devices would with high probability remain.

actuators would only communicate with the simulator in order to exchange primary simulation data (input and outputs, parameters, variables, and control commands like event updates)

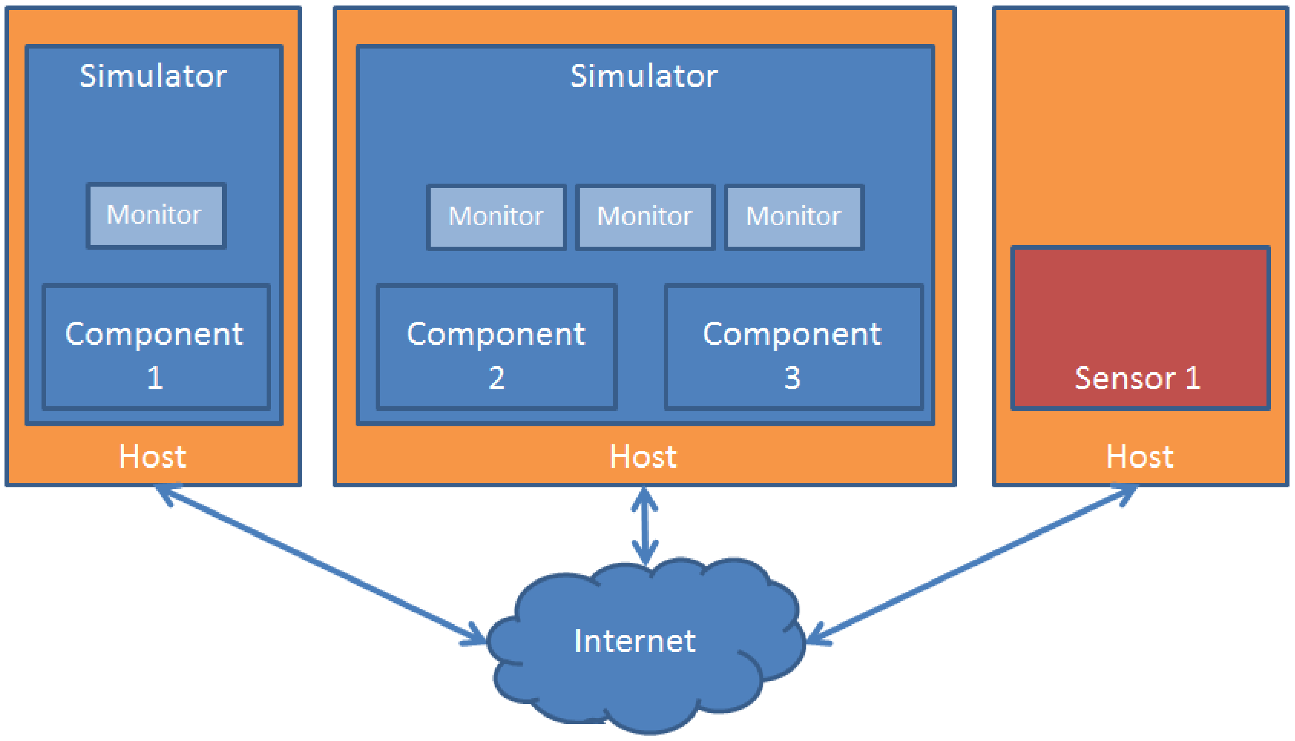

Figure 7 Monitoring on Simulator

The one advantage of this approach is that the contract monitoring mechanism has possibly only to be implemented for one platform[3]. Another advantage is that only the simulator has to be considered with regard to contracts deployment. This means communication between simulator and design server in order to receive the contracts information for the components deployed in the simulation (including things like authentication, access rights management, and discovery). Additional, trust worthiness would have only to be verified for the simulator[4].

A disadvantage may be the amount of needed computational power on a single machine for monitoring[5], especially if done in an interpreted way.

4.2.2 Monitoring on Physical Device

A different approach is to deploy the monitoring capability directly on the actuator or sensor. Due to the different types of sensors and actuators and there connection to the internet of systems engineering one can assume three different deployment scenarios. These scenarios have certain characteristics in common. They reduce the needed for computational power on the simulation host computer. The communication with the simulator has to be extended with the capability of exchanging information relating to contract monitoring (e.g. notification in case of contract violation through the physical device). Monitoring capability has probably

[3] Simulating across different platforms and architectures is also an advantage of the distributed simulation. Imagine precompiled third party Functional Mock-up Units (Units) which are not compiled for the platform of the host running the simulation engine.
[4] This is very important if you want to utilize this approach in a development environment underling certification.
[5] However, with arising multicore and GPU technology and the inherent relatively low complexity of the contracts this can probably be neglected.

to be made available for a different platform than the simulator host platform. The physical device has to be capable of receiving the contracts information necessary for monitoring (including trust worthiness). All three scenarios are more elaborated in the following three subsections.

4.2.2.1 Monitoring on rich Physical Device

In case of a rich[6] physical device; the contract monitor can directly be deployed on the sensor or actuator. The device has all the capabilities necessary for monitoring the conformance to the contract during simulation time.

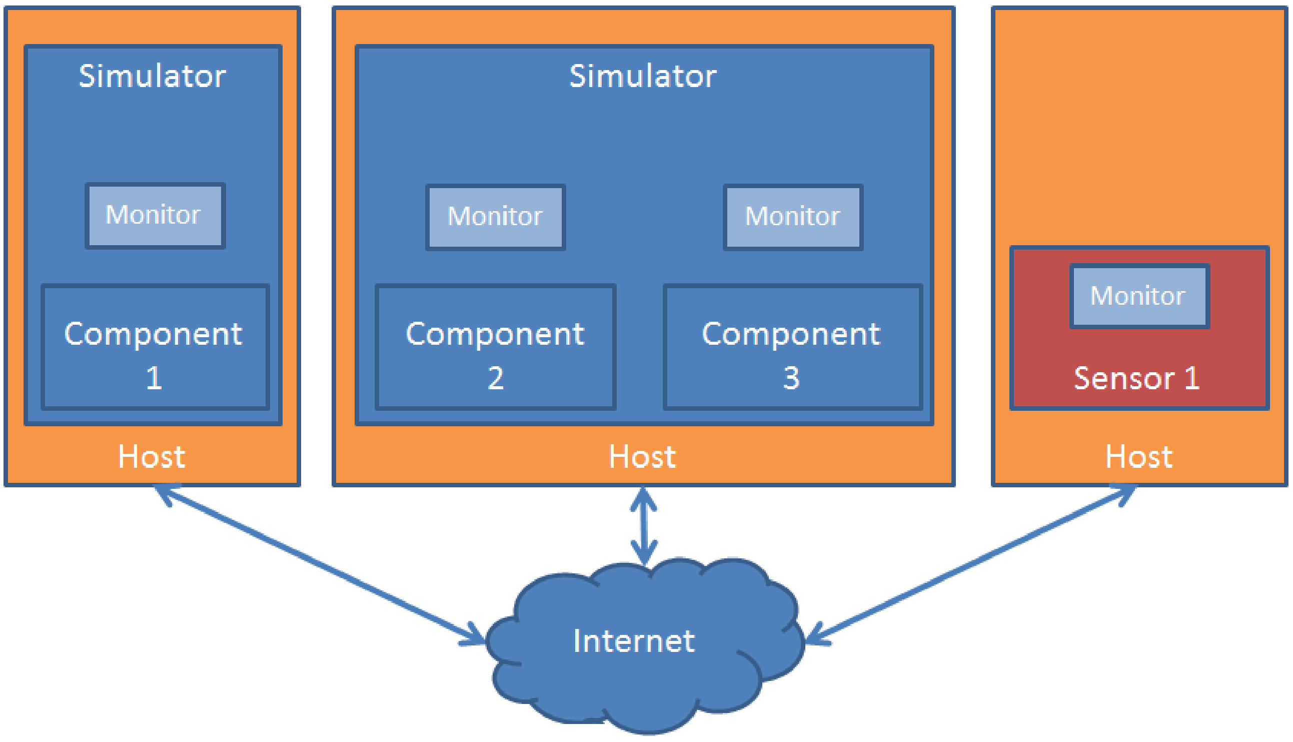

Figure 8 Monitoring on rich Physical Device

This is the simplest case. However, this case is usually unlikely in the area of system engineering.

4.2.2.2 Monitoring on Gateway

In many cases the sensor is not directly connected to the internet but via a gate way. This is for example the case if the sensor is a low power device. Another scenario is the connection of non-internet communication interfaces which are relevant in systems engineering (like in the automotive domain) and in embedded systems like CAN, LIN and FlexRay. In this case it is sensible to deploy the monitor on the gateway. This has also the advantage that the behaviour of the sensor or actuator is not influenced[7].

[6] Rich in the sense, that it has direct access to the internet and enough computational power to host an http server side for REST communication.

[7] Deployment of the monitoring on the physical device might have an impact on the real-time behavior of the device.

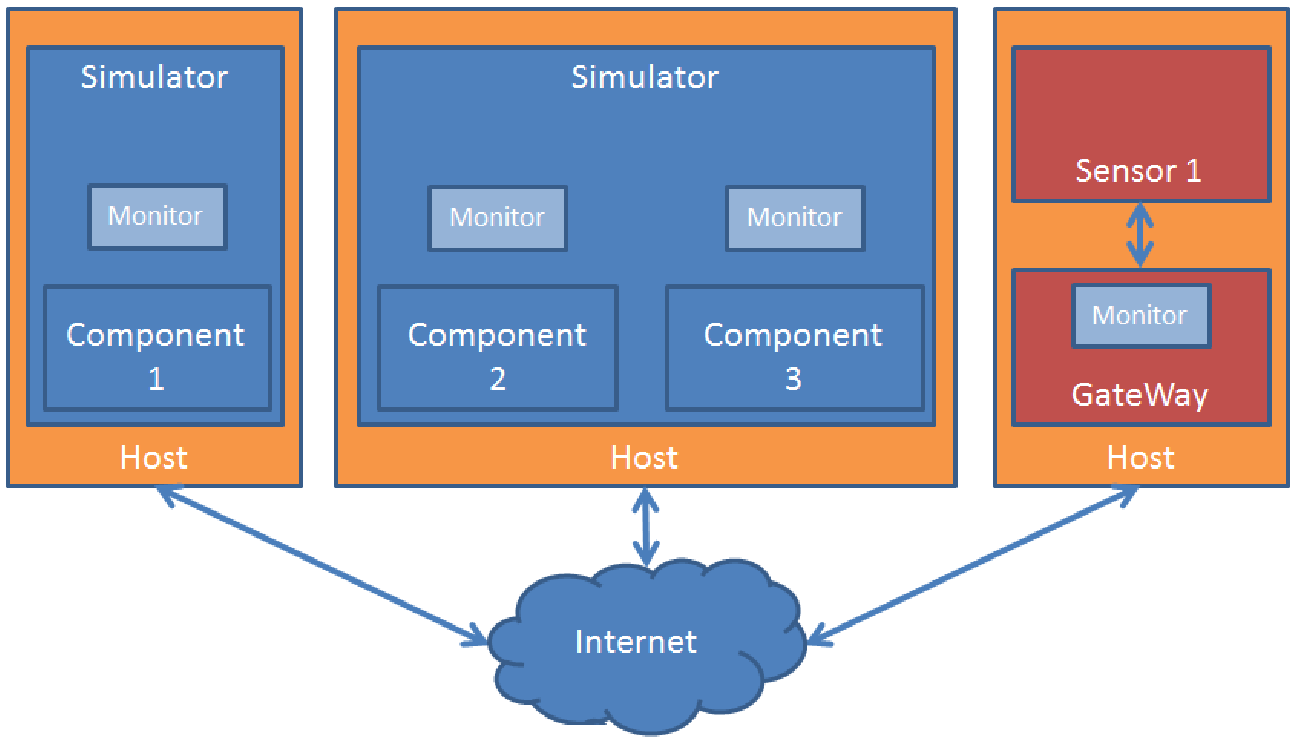

Figure 9 Monitoring on Gateway

The gateway would then also have the responsibility of being able to receive the contracts information necessary for monitoring (including issues like trust worthiness).

4.2.2.3 Monitoring on tiny Physical Device with Trigger

Especially on low powered sensors monitoring might happen in log intervals since communication is reduced to a minimum in order to save power[8]. In certain cases these devices have the capability to send in time updates in case of an event. These events occur through triggers which monitor value rages in a simple way (e.g. temperature from 10 to 20 centigrade). These triggers are not as sophisticated as but allow early updates in case of an event (see Figure 10 Monitoring on tiny Physical Device with Trigger).

In this setup the gateway has to analyse the contract for relevant variables to be monitored. Depending on the structure of the contract the triggers on the devices have to be set in a sensible way as to ensure in timeliness of notification. This makes the deployment and monitoring on the gateway more complex than in the case without triggers.

[8] These devices might rely on energy harvesting (e.g. through temperature difference).

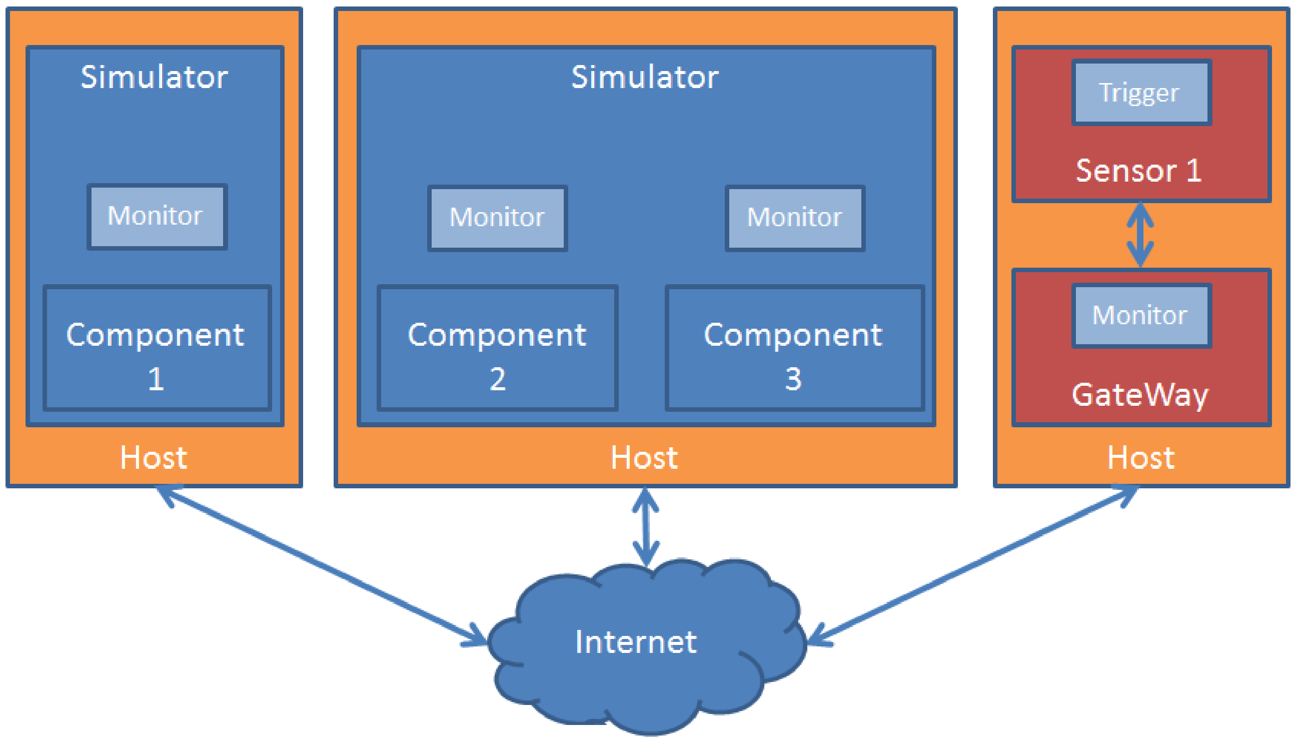

Figure 10 Monitoring on tiny Physical Device with Trigger

5 Service Definition

This section contains the necessary service definitions in order to allow integration of sensors actuators and contracts.

5.1 Device Registration Service

This service is mainly for the handling of the device during design time but also important for simulation set up (discovery of the available devices and their functionality).

A separate definition of the service is not necessary since the devices behave like tools as defined in D5.4 Architectural principles for Internet of System Design and the IoT [D5.4]. Which means:

On start up or introduction into the Sprint Engineering Environment the device registers itself in the SSI uploading all relevant meta-data allowing it through semantic mediation to be presented as component instances in tools. Additionally, devices may implement a REST service allowing the discovery, updating and viewing of device information via the internet (HTTP).

5.1.1 Meta data registration

As meta data vocabulary the Basic Structural Ontology [BSO] form D3.2 [D3.2] has been chosen. BSO contains all the necessary and essential elements to represent physical devices as components in the engineering environment.

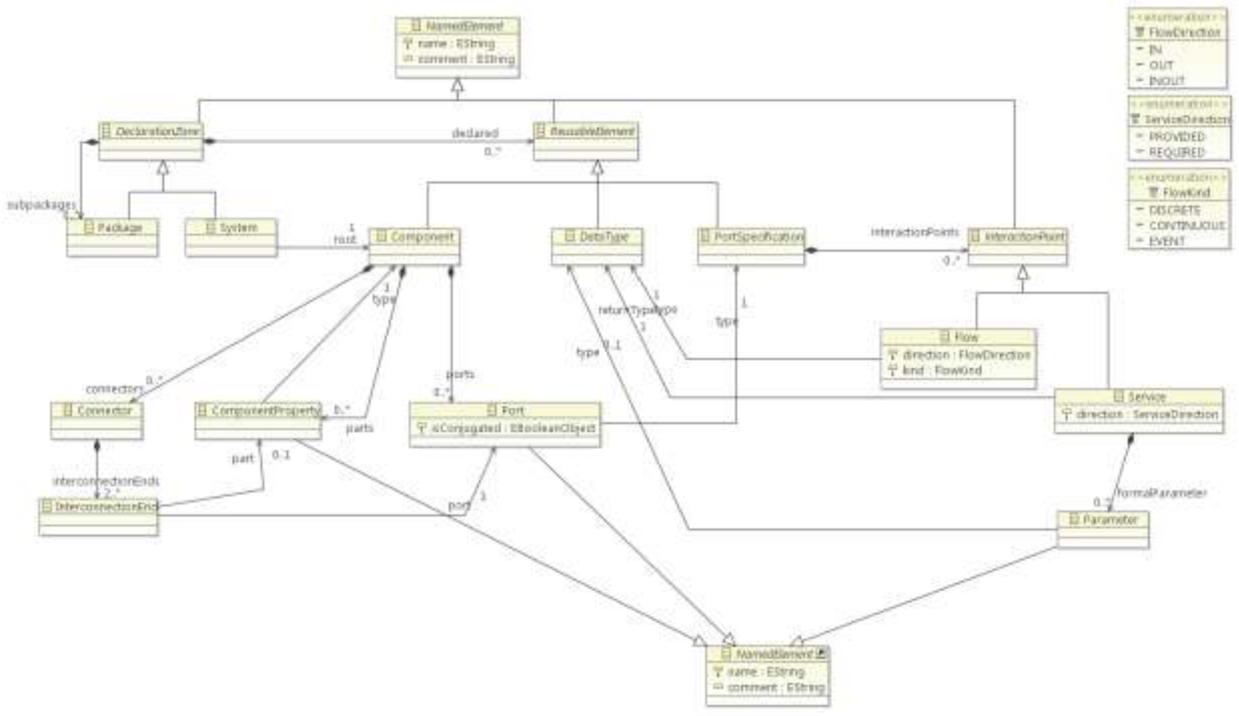

Figure 5-1: Basic Structural Ontology Meta Model

On device start-up certain kind of steps are gone though in order to ensure correct exposing of the meta data. These steps are:

- Check if BSO is registered on the server
  - If not register and proceed, else proceed
- Check if type of device is registered on the server

- If not register and proceed, else proceed

An example device x would register as components x:

```
<rdf:Description rdf:about="x">
  <bso:name>X</bso:name>
  <rdf:type rdf:resource="http://www.sprint-iot.eu/bso/Component"/>
</rdf:Description>
```

An example port y with type z would register as port y:

```
<rdf:Description rdf:about="y">
  <bso:type rdf:resource="z"/>
  <bso:name>Y </bso:name>
  <rdf:type rdf:resource="http://www.sprint-iot.eu/bso/Port"/>
</rdf:Description>
```

5.1.2 Instance registration

If meta data registration or check was successful. Devices can register themselves as instances of the registered device types (components) on start-up. Devices may unregister themselves by removing device information form the server on shut down if through the nature of the device longer unavailability can be assumed.

Steps to register the device are:

- Check if the device is registered on the server
    - If not register and proceed, else proceed

An example of a device instance x of type y registration look like:

```
<rdf:Description rdf:about="X">
  <bso:type rdf:resource="Y"/>
  <bso:name>X</bso:name>
  <rdf:type rdf:resource="http://www.sprint-iot.eu/bso/ComponentProperty"/>
</rdf:Description>
```

5.2 Monitor Deployment Service

In case of monitoring on a simulator monitoring deployment is handled by the simulator. In this deliverable we focus on the deployment on the physical device. The best suggestion proposed there is to reuse the Functional Mock-Up Interface [FMI] (for model exchange) standard. The advantage is that it already contains the necessary in formation in a container format (e.g. Functional Mock-Up Unit) which can then be directly uploaded to the sensor or actuator.

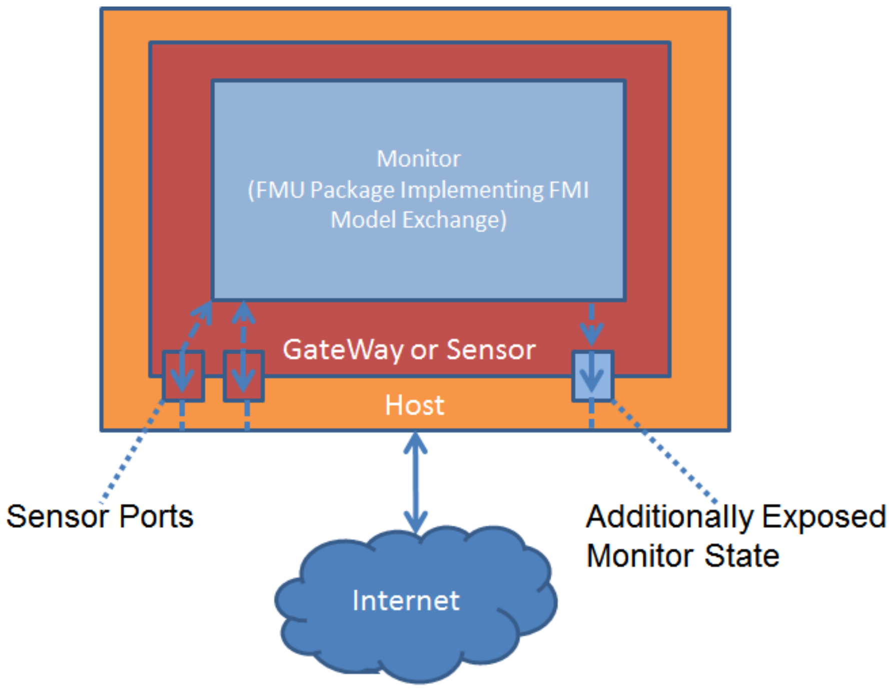
Monitor
(FMU Package Implementing FMI Model Exchange)
GateWay or Sensor
Host
Sensor Ports
Additionally Exposed Monitor State
Internet

Figure 2 Monitoring on physical Device Integration using FMI

The sensor or actuator is then responsible to feed the monitoring FMU with the sensor data (for simplicity reasons we require a one to one name mapping). Additionally, it has to expose the outputs of the monitoring FMU to the simulator interface. The simulator has the required understanding for the interpretation of these outputs.

Using this approach the only thing required as interface definition is the contracts uploading mechanism. This is done using a post with mime type ‘application/fmu’ to the device URL containing the FMU in the payload. If no FMU is in the payload the monitor is unemployed (if one is already deployed).

6 Abbreviations and Definitions

Architecture	The fundamental organization of a system (maybe on different abstraction levels) embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.
Artefact	A physical piece of information that is used or produced by a software development process. Examples of Artefacts include models, source files, scripts, and binary executable files.
Attribute	A piece of information associated with an entity. Used to describe a characteristic of an entity. In the meta-model, an attribute is a placeholder for the actual information that is defined in the models.
BSO	Basic Structural Ontology, Ontology introduced by the Sprint project and described in [D3.2]
CESAR	"CESAR" stands for Cost-efficient methods and processes for safety relevant embedded systems and is a European funded project from ARTEMIS JOINT UNDERTAKING (JU). (See: http://www.cesarproject.eu/)
Domain-specific Language	Domain-specific language (DSL) is a programming language or specification language dedicated to a particular problem domain.
DSL	Domain-specific Language
Closed World Assumption	The closed world assumption is the presumption that what is not currently known to be true, is false. The opposite of the closed world assumption is the open world assumption. (See: http://en.wikipedia.org/wiki/Closed_world_assumption)
Eclipse Modeling Framework	Is a modeling framework and code generation facility for building tools and other applications based on a structured data model. (See: http://www.eclipse.org/emf/)
eCore	Pendant to EMOF in EMF.
EMF	See: Eclipse Modeling Framework
Entity	An object with an identity that is distinguishable from other entities.
Error	Discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition.
Extensible Markup Language	Is a set of rules for encoding documents in machine-readable form defined by W3C. (See: http://www.w3.org/TR/2008/REC-xml-20081126/)
Heterogeneous Rich Component	Component describing meta model to facilitate heterogeneous information from different domains (e.g. software development, hardware development) and aspects (e.g. functional, non-functional), first developed in the IP-SPEEDS project
HRC	See: Heterogeneous Rich Component
Instance	An entity that is obtained from another entity by the process of instantiation.
Interface	Abstraction of a service that only defines the operations supported by that service (publicly accessible variables, procedures, or methods), but not their implementation.

IP-SPEEDS	See: Speeds
Link	Representation of a relation between two objects.
Metamodel	A model for describing (structure of) other models on a meta level.
Model	A semantically closed abstraction of a software or hardware system, i.e. a simplification of reality that gives a complete description a system from a particular perspective.
Object Management Group	Is a standardization organization in the modelling area (e.g. programs, systems and business processes). (See: http://www.omg.org)
OMG	See: Object Management Group
Open Services for Lifecycle Collaboration	Open Services for Lifecycle Collaboration (also known as OSLC or Open Services) is a community and set of specifications for Linked Lifecycle Data. The community's goal is to help product and software delivery teams by making it easier to use lifecycle tools in combination. (See: http://open-services.net/html/Home.html)
OSLC	See: Open Services for Lifecycle Collaboration
OWA	See: Open World Assumption
OWL	See: Web Ontology Language
RDF	See: Resource Description Framework
Resource Description Framework	It's a family of W3C specifications for conceptual description or modelling of information that is implemented in web resources. (See: http://www.w3.org/TR/rdf-primer/)
Speeds	Speeds (speculative and exploratory design in systems engineering) is an FP7 Project. (See: http://www.speeds.eu.com/)
System	A collection of components organized to accomplish a specific function or set of functions.
Traceability	Traceability is a technique used to provide relationships between objects (e.g. in requirements, design and implementation of a system in order to manage the effect of change).
UML	See: Unified Modeling Language
UML Profile	A profile in the Unified Modeling Language provides a generic extension mechanism for building UML models in particular domains. Profiles are based on additional stereotypes and tagged values that are applied to elements, attributes, methods, links, and link ends. A profile is a collection of such extensions and restrictions that together describe some particular modeling problem and facilitate modeling constructs in that domain.
Unified Modeling Language	From Grady Booch, Ivar Jacobson and Jim Rumbaugh (Co. Rational software) developed, and by the Object Management Group (OMG) confirmed abstract description language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for other non-software systems.
View	A representation of a whole system from the perspective of a related set of concerns.
W3C	See: World Wide Web Consortium
Web Ontology Language	Is a family of knowledge representation languages defined by W3C for authoring ontologies. (See: http://www.w3.org/TR/owl-ref/)

World Wide Web Consortium	Is the main international standardization organization for the World Wide Web. (See: http://www.w3.org)
XML	See: Extensible Markup Language
XML Schema	Language for XML schema description which has Recommendation status by the W3C. (See: http://www.w3.org/TR/xmlschema-1/, http://www.w3.org/TR/xmlschema-2/)
XSD	See: XML Schema

7 References

[D3.2]	D3.2 Definition of the Semantic Services Integration Layer
[D4.3]	D4.3 Contract Based Verification Methods, Sprint Deliverable
[D4.15]	D4.15 Real Time Service Definition, Sprint Deliverable
[D5.4]	D5.4 Architectural principles for Internet of System Design and the IoT, Sprint Deliverable
[FMI]	FMI for Model Exchange, version 1.0, http://www.modelisar.com/specifications/FMI_for_ModelExchange_v1.0.pdf