SitRS – A Situation Recognition Service based on Modeling and Executing Situation Templates

Pascal Hirmer[1], Matthias Wieland[1], Holger Schwarz[1], Bernhard Mitschang[1], Uwe Breitenbücher[2], and Frank Leymann[2]

[1] Universität Stuttgart, Institute of Parallel and Distributed Systems, 70569 Stuttgart, Germany
pascal.hirmer@ipvs.uni-stuttgart.de
http://www.ipvs.uni-stuttgart.de/
[2] Universität Stuttgart, Institute of Architecture of Application Systems, 70569 Stuttgart, Germany
http://www.iaas.uni-stuttgart.de/

Abstract. Today, the Internet of Things has evolved due to an advanced connectivity of physical objects. Furthermore, Cloud Computing gains more and more interest for the provisioning of services. In this paper, we want to further improve the integration of these two areas by providing a cloud-based situation recognition service – SitRS. This service can be used to integrate real world objects – the *things* – into the internet by deriving their situational state based on sensors. This enables context-aware applications to detect events in a smart environment. SitRS is a basic service enabling a generic and easy implementation of Smart* applications such as SmartFactorys, SmartCities, SmartHomes. This paper introduces an approach containing a method and a system architecture for the realization of such a service. The core steps of the method are: (i) registration of the sensors, (ii) modeling of the situation, and (iii) execution of the situation recognition. Furthermore, a prototypical implementation of SitRS is presented and evaluated via runtime measurements.

Keywords: Situation Recognition, IoT, Context, Integration, Cloud Computing, OSLC

1 Introduction

A major challenge for the Internet of Things (IoT) is sensor data integration and sensor data processing [10]. The sensor access should be pervasive and the integration of the sensors has to be automated. Furthermore, the sensor data have to be interpreted in order to derive situations that can be understood and processed more easily than the huge amount of low-level data, which is difficult to handle. To enable situation-awareness for the IoT, different levels of processing are needed. These levels are described in Fig. 1. Here, the first level – the data level – contains the sensor devices. On this level only the raw sensor data is available, which is very complex and difficult to process. Because

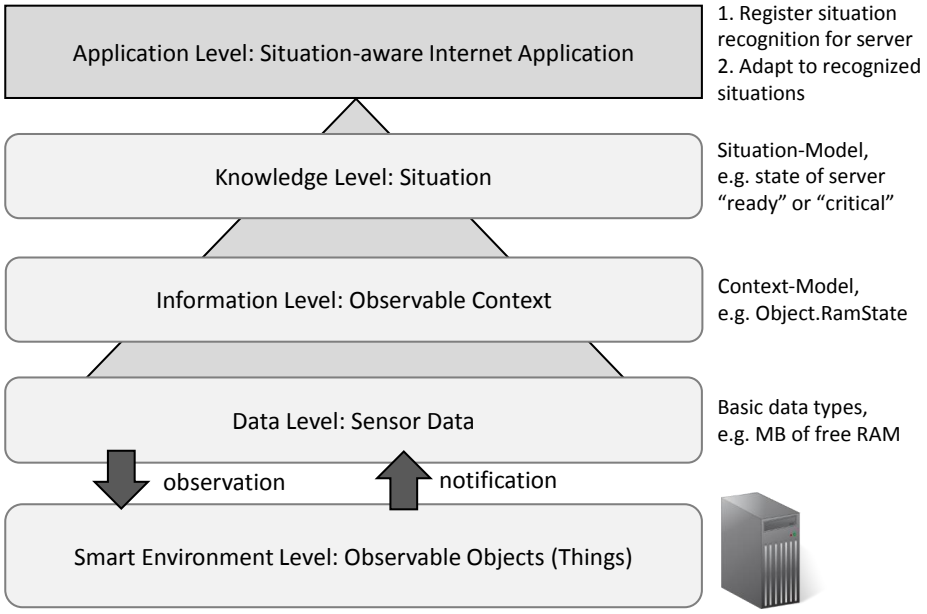

Fig. 1. Transition Levels from Data via Information to Knowledge

of that, the sensors are pushing their data to the next level – the information level. At this point, the sensor data, such as temperature or load percentage, is enhanced with information about their relations to objects, such as smart phones or computers in a smart environment. On the information level, this data i.e., the observable context, is linked to real world objects of the smart environment and becomes information about the environment. Based on this context information sensor data is aggregated and interpreted in order to derive well-understandable situations that lead to knowledge about the smart environment. This knowledge, i.e. high-level context, can be processed on a higher-level of abstraction, which simplifies building situation-aware applications.

A method and a system architecture to provide this sensor data processing for situation recognition as a service in an automatic, cloud-based manner are the main contributions of this paper. Our system architecture supports automated service deployment, a web-based front-end and loose coupling. This has many advantages like concurrent remote access, high availability and scalability in order to support multiple instances of our service as well as the integration of many distributed sensors. In addition, we provide a means to define the situations that could occur, that is, a model containing all necessary information for their recognition. This model, called *situation template* in this paper, contains the sensors being monitored as well as the conditions that have to match for a certain situation. Once the model is created, it can be used to execute a data flow that integrates the sensor information and executes comparison operations to

recognize occurring situations. The result of the processing is the recognition of situations that allow applications to adapt to the smart environments observed by the sensors. The advantage of such a service is, that the applications do not have to care about the sensor access, the sensor data processing and not even about the situation recognition. Instead, the applications only query the needed knowledge or register for push notifications on occurring situations. The service cares for finding appropriate sensor devices, storing the context data for queries, providing a registration service for push notifications and finally automatically setting up the situation recognition for the needed situations. This enables smart applications to integrate real world objects into the internet by deriving their situational state based on sensors.

The remainder of this paper is structured as follows: First, *Section 2* introduces related work. After that, *Section 3* presents an architecture and method for situation recognition that copes with the mentioned issues and enables situation recognition based on sensor data. Afterwards, in *Section 4* we present our prototypical implementation of SitRS. *Section 5* evaluates the approach using runtime measurements and finally *Section 6* gives a summary of the paper and an outlook on future work.

Motivating Scenario: This section introduces a motivating scenario that is used throughout the paper to explain our approach. The goal of this scenario is the monitoring of sensors of several machines simultaneously and the reaction on occurring situations. For example, these machines could be web servers or cloud-based virtual machines in a data center. Using a dashboard, the currently occurring situation of all machines and, as a consequence, the state of a web server or a data center can be seen immediately. It's even possible to receive notifications in case of emerging problems. For that, we define three types of situations: (i) "Failed" indicates that the system is not available due to an occurred error, (ii) "Critical" indicates an occurring problem that could lead to a system failure (cf. example in Fig. 4) and (iii) "Running" indicates that no problem is occurring or emerging. The sensor data that is used to recognize these situations is provided by heterogeneous APIs, depending on the respective machine. A main challenge in this scenario is (i) coping with different representations of the sensor data, (ii) integrating the sensor data, (iii) computing the situation, and (iv) integrating highly heterogeneous APIs. Our solution is able to realize this scenario by representing sensor data as uniform REST resources and by integrating and analyzing them using a data flow-based integration. We will explain the following concepts based on this motivating scenario, which has also been implemented in our prototype.

2 Related Work

Acquiring, modeling and managing context information is a tedious and expensive task [6, 8]. As a consequence, it is beneficial to share this information between different kinds of context-aware applications. We use the definition of context

given by A.K. Dey and G.D. Abowd as "any information that can be used to characterize the situation of an entity, where an entity can be a person, place, or object" [5]. Thus, as Dey and Abowd already defined, context information can be used to identify and derive situations. Context models were introduced in previous work [6] to represent or mirror certain aspects of the real world as closely as possible thereby serving as a shared, common basis for different context-aware applications and systems. In this paper, however, we concentrate on how context and context models can be used to recognize situations. So the basic idea is to enhance an existing context model infrastructure with a situation recognition service based on so called situation templates. A situation template is an abstract, machine-readable description of a certain basic situation, which describes context information considered for being relevant for the situation and a description of how to derive the existence of a situation from these values. Situation templates were introduced before and this paper builds on the definition presented in [7]. Due to the historical development of rule-based expert systems, most context reasoning systems [11] use ontology-based and predefined rule-based approaches. Compared to our approach, most of the existing context-aware systems are supposed to cover only a limited geographical area or support only a specific use case scenario [2]. In our approach, any geographical area can be supported using a global context model and in addition any kind of situation recognition can be modeled as situation template based on the available context model. Unlike situation recognition approaches that are based on pattern recognition using e.g., machine learning [1] or on ontological reasoning [4] our approach executes the situation recognition as a data flow. Furthermore, complex event processing (CEP) [3] engines can be used for data flow execution in our approach. Hence, the only errors and uncertainties in the process result from the sensors and their sensor data readings. The data flow processing is accurate.

For the execution of the situation recognition, we use the *Pipes and Filters* pattern [9] – which is implemented in our prototype using Node-RED[3] – and build on a transformation approach presented previously in [12]. There, the concept of mapping the *Pipes and Filters* pattern to an executable representation was presented. In this paper, this concept is enhanced by a more detailed approach introducing a method for situation recognition as well as a prototypical implementation.

3 SitRS – Architecture and Method

The SitRS architecture, displayed in Fig. 2, consists of the situation model, the situation recognition service, and the sensors. The components of the situation recognition service can be deployed as cloud services, on a local machine or in a hybrid manner. The service is subdivided into two core components, the *situation recognition system* and the *resource management platform* and furthermore contains two repositories, one for storing the situation templates and the other for storing sensor information. In addition, it contains the following software components: the situation registration service and the sensor adapters.

[3] http://nodered.org/

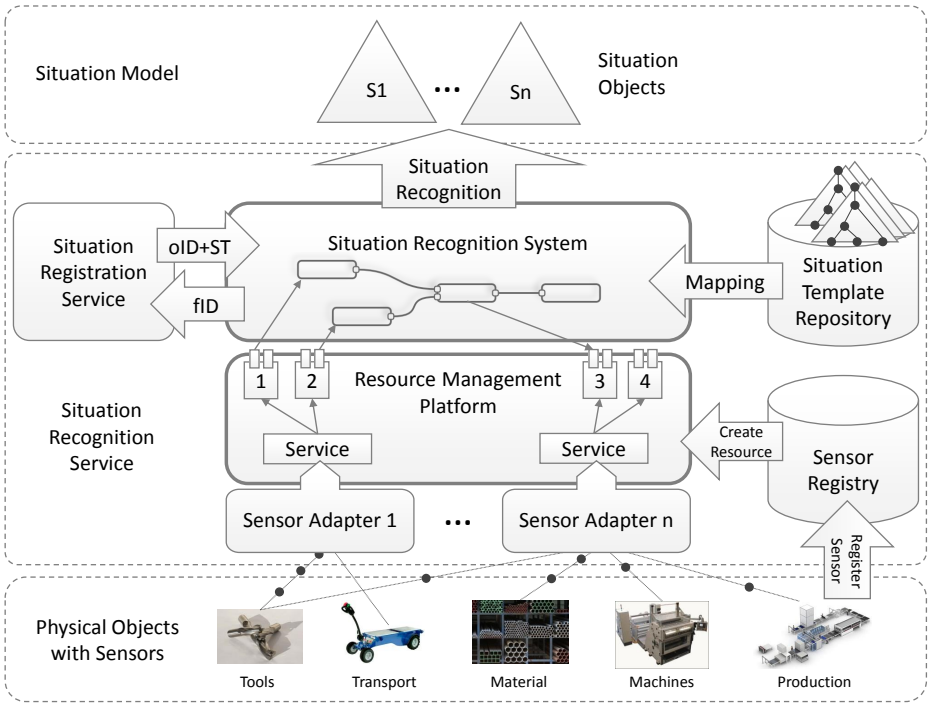

Fig. 2. SitRS – Architecture

The sensors at the bottom level can be registered in the Sensor Registry, which invokes the resource management platform that extracts the sensor data via the adapters and provisions them as uniform REST resources. Based on the registered sensors, a description defining the conditions for an occurring situation is modeled using so called situation templates. These situation templates are stored in the Situation Template Repository. The Situation Registration Service is used for the registration on occurring situations based on the situation templates. The situation templates are mapped onto an executable representation – we call *executable situation template* in the context of this paper – and executed in the Situation Recognition System, i.e., an execution engine. The output of this engine determines if modeled situations occurred. Note that the mapping from a situation template to an executable representation is necessary to support different execution engines, i.e., to prevent being dependent on a specific engine.

The introduced architecture is used as shown in Fig. 3. There are two kinds of actors participating in this method, the *situation recognition user* and the *situation recognition admin*. The admin has to register the sensors to be used. The situation recognition user models the situation to be recognized as situation template and processes the notifications of the situation recognition. This separation enables the usage by non-expert users regarding sensor integration and technical details. The method contains all steps needed for defining the

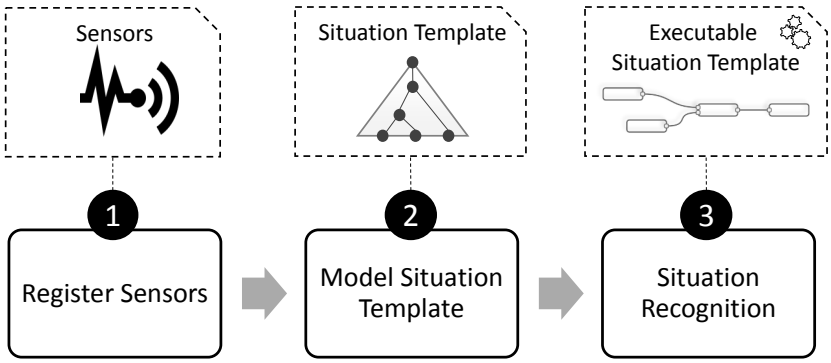

Fig. 3. Method for Situation Recognition

continuous recognition of a situation. Due to a design decision, only a situation for a *single* object, e.g. a web server, can be monitored by our approach. As a consequence, this method has to be re-applied for different objects. Because of that, it makes sense to create a single (cloud-based) instance of the service for each object to be monitored. Recognizing situations regarding multiple objects is part of our future work. The overall method consists of the following steps:

Step 1 – Sensor Registration: In the first step of the situation recognition method, the available sensors are registered in the sensor registry component (cf. Fig. 2). This registry is connected to the resource management platform, which provides the sensor's data as uniform REST resources. To register a sensor of a specific object, e.g., the heat sensor of a machine, the object's id, the type of the sensor as well as its access path have to be specified. Thereupon, an entry is created in the sensor registry containing the given information and an unique id of the registered sensor. Once a sensor is registered, an event is generated that notifies the resource management platform. Thereupon, this platform creates an adapter to connect to the sensor and provides its data through a REST resource. Note that each sensor is represented by exactly one REST resource. The URI of this resource can be requested from the sensor registry using the sensor's id and is used for the transformation of a situation template to an executable representation, which is described in Step 3.2.

Step 2 – Situation Template Modeling: Before we are able to recognize situations, we need a means to define them. To enable this, we build *situation templates* (ST), using Situation-Aggregation-Trees (SAT) that were defined by Zweigle et al. [13]. These SATs are directed graphs resulting in a tree structure, in which the branches are aggregated bottom-up (as shown in Fig. 4). As a consequence, all paths are joined in a single root node that represents the situation. The leaf nodes of the situation template – called *context nodes* – represent the sensors. These context nodes are connected to condition nodes for

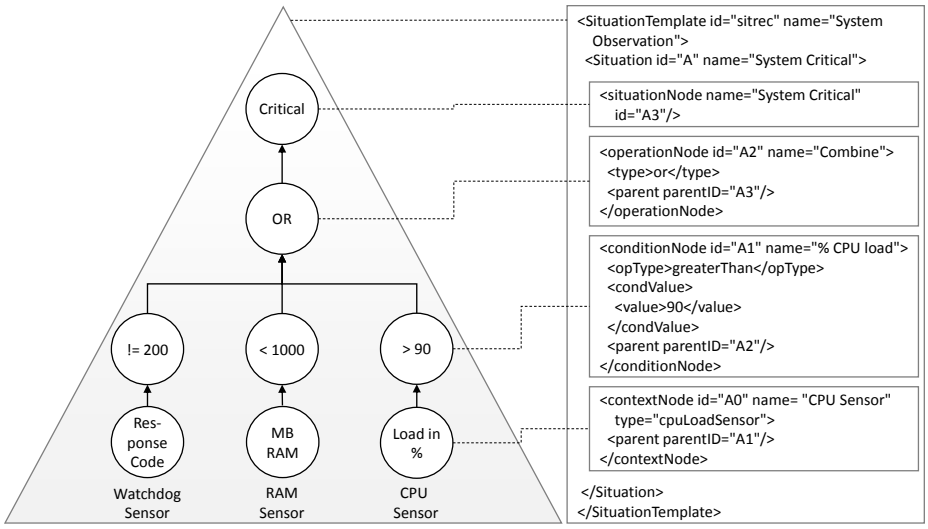

Fig. 4. Example of a Situation Template modeled in XML

filtering the incoming sensor data based on a condition. The output of these condition nodes can be aggregated by operation nodes using logical operations until the root node is reached. In previous work, no machine-readable format has been properly defined for the exchange and definition of these SATs, which is important to enable automated processing. To overcome this issue, we propose a schema based on XML. Of course, other formats such as JSON could be used as well. Note that modeling XML manually is a time-consuming and also error-prone task due to the lack of an automated schema validation. To cope with this issue, we recommend using existing XML modeling tools, both graphically or textually. Due to the fact that a large variety of XML modeling tools already exist, we do not provide an additional modeling tool for situation templates.

Figure 4 shows an example of a situation template that serves the recognition of the situation "Critical" of a web server as described in the motivating scenario in Section 1. To model such a situation, firstly, the available sensors of the machine have to be modeled using context nodes that are containing the type of the sensor. These context nodes are then connected to condition nodes that compare the sensor's data with predefined values. In the shown example, (i) the CPU load percentage should be greater than 90, (ii) the available RAM should be lower than 1000 MB, and (iii) the response code of the machine should not equal 200 in order to produce the output *true*. These condition nodes are aggregated using operation nodes that represent logical operators, in this specific example the OR operation node. The root of the SAT is the situation itself, i.e., the situation occurs if the root node evaluates to *true*.

In the following, we describe the individual parts of a situation template in detail, which is defined using an XML schema definition that can be found

online[4]. Each situation template has a unique identifier, a name and may contain an arbitrary number of situations. This enables the simultaneous monitoring of many different occurring situations within a single situation template. A situation describes, which conditions have to apply for its occurrence, i.e., it is defined by a directed tree. This tree contains a single root node, the situation node, which occurs once inside a situation. The situation node describes the situation to be monitored. A situation is uniquely defined by an identifier and its name. Furthermore, a situation contains an arbitrary number of context nodes, condition nodes and operation nodes. These nodes are connected using the *parent* element, which contains a reference to the parent node.

Context nodes are used to describe the sensors that provide the data and are defined with an identifier, a name and its type. Detailed information about the sensor can be requested from the registry using the type attribute of the context node as well as the identifier of the monitored object. Note that each sensor that is being modeled has to be registered in the sensor registry first (cf. Step 1). The parent nodes of context nodes are always condition nodes, as shown in Fig. 4.

Condition nodes are used to compare sensor data with values that are predefined in the situation template. Possible types of condition nodes are *greater than*, *less than*, *equals*, *not equals* and *between*. The value used for comparison can be determined in the XML element *condValue*. Furthermore, each condition node can have an arbitrary number of operation nodes as parents.

Operation nodes are used to aggregate the output of the condition nodes and are restricted to the logical operations *AND*, *OR*, *XOR* and *NOT*. That is, a situation usually occurs if more than one condition applies. However, if a situation is dependent on only one condition, no operation nodes have to be modeled. The parent of an operation node is either a single situation node or an arbitrary number of operation nodes.

To ensure reusability and concurrent access, the modeled situation templates are stored in the *situation template repository*.

Step 3 – Situation Recognition: The third step of our method is subdivided into several sub-steps that are shown in Fig. 5 and are described in the following.

Step 3.1 – Situation Registration: The situation registry serves the registration on a specific situation to be recognized. The input of the situation registry is the id of the situation template as well as the id of the object (oID) to be monitored. On successful registration, the situation registration service returns an observation flow instance id (fID) that can be used for deregistration or management purposes. Once a situation is registered, an event is generated that invokes the transformation of the situation template. This transformation receives the situation template from the situation template repository using the given id and transforms it into the executable situation template. After that, this executable situation template can be deployed and executed in the respective

[4] http://pastebin.com/TyBNPUEs

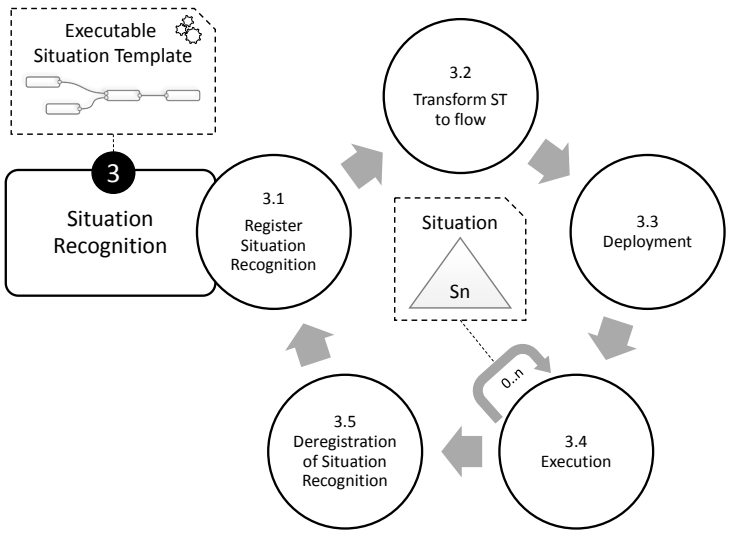

Fig. 5. Detailed View of the Situation Recognition Step

runtime environment. The execution runs until a situation template or all objects relating to a situation template are deregistered in the situation registry.

Step 3.2 – Situation Template Transformation: The transformation of situation templates serves the creation of an executable, event- and flow-based representation that is able to recognize occurring situations based on the modeled situation template. The input of the mapping is the identifier of the object to be monitored (e.g., a web server) that was entered in the situation registry. In our prototypical implementation, e.g., the format of the executable representation is defined in JSON so it can be executed in the Node-RED environment. However, depending on the execution environment, many different formats are possible. We provide a 1-to-1 transformation from the XML-based situation template to an executable representation. That is, each element of the situation template is represented by exactly one element in the executable representation. In the first step of the transformation, we map the context nodes onto calls of REST resources that provide the latest sensor data. To do so, we receive the access information, i.e. the URL of the resource, from the sensor registry (cf. Step 1), using the object id from the mapping's input and the type of the sensor defined in the situation template. The second step of the mapping processes the condition nodes, i.e., the nodes implementing comparison operations such as *greater than*, *less than* or *equals* that compare sensor data with predefined values. These condition nodes are mapped to predefined function nodes, implementing the comparison operations, e.g., using JavaScript in Node-RED. In a similar fashion, the third step maps the condition nodes *AND*, *OR*, *XOR* or *NOT* to

corresponding function nodes that implement these logical operators. In the final step, the nodes are connected using the means of the respective execution model. The result is an executable situation template that recognizes occurring situations through its execution. The time interval, in which the data flow will be executed, that is, in which a situation should be monitored, has to be predefined by the user of the solution and is used as input for the transformation. This is necessary, because the execution time interval strongly depends on the use case. For each situation, modeled in the situation template, a single flow graph is created and can be deployed and executed separately.

Step 3.3 – Situation Template Deployment: After its transformation, the executable situation template is deployed into the execution environment, e.g., Node-RED, CEP-Esper[5] or Odysseus[6]. As a consequence, the deployment serves as the interface to the execution engines being used. It should be flexible enough to support different engines and should also be able to handle occurring errors during the deployment. Operations supported by the deployment are *deploy*, *start situation recognition* and *stop situation recognition*. In our prototype, for example, the deployment sends a HTTP REST call to the Node-RED engine to deploy a mapped situation template. After that, the situation recognition flow is initiated by executing the *start* command. The situation recognition is active as long as the modeled situation should be monitored, i.e., until it is deregistered in the situation registry.

Step 3.4 – Situation Template Execution: After the deployment, the executable situation template is executed using the respective execution environment, e.g., an event-processing engine such as Node-RED. In the predetermined time interval, the sensor data is requested from the REST resources that return the latest sensor data. Thereupon, the further nodes of the situation template are processed. These are always condition nodes that were mapped onto predefined function nodes that compare the sensor data with predefined values and return a Boolean value determining whether the condition applies. After each of these condition nodes is processed, their output is concatenated using the mapped operation nodes that implement logical operations. The concatenation of the paths is processed until a single output emerges. This output is a Boolean value, determining whether a situation occurred or not. This flow is repeated in the given time interval until the situation is deregistered.

Step 3.5 – Situation Deregistration: The final step of the situation recognition is the deregistration of a situation template for a certain object. After the need for the recognition of a situation expires, it is deregistered in the situation registry. In case no more registrations exist for a situation, two steps are processed. First, the execution engine stops the situation recognition flow. Second, the executable situation template is undeployed from its execution environment.

[5] http://esper.codehaus.org/ [6] http://odysseus.informatik.uni-oldenburg.de/

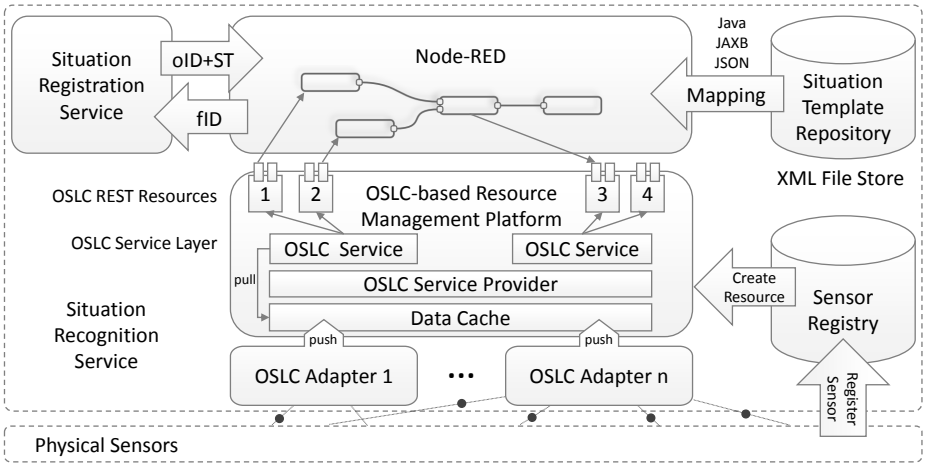

Fig. 6. Architecture of the SitRS Prototype

The deregistration of a situation secures that no unnecessary resources are spent for an (even temporary) unneeded situation recognition.

4 Prototypical Implementation

In this chapter, we describe our prototypical implementation of the introduced concepts. The overall architecture of the prototype is shown in Fig. 6. Furthermore, the prototype is available on GitHub (https://github.com/hirmerpl/SitOPT).

Firstly, we implemented a *mapping of situation templates* defined in XML to an executable representation in JSON. This mapping has been implemented as a Java library using the Java Architecture for XML Binding (JAXB), which is used to parse the situation template. Furthermore, we used the Apache Wink[7] JSON library to create a JSON model for the executable representation.

After the mapping is processed, we *deploy the executable situation template to Node-RED* using the provided HTTP REST interface. There exist many technologies that could have been used for processing the situation template such as RestFlow[8]. However, for our prototype, it is a requirement that the used engine is web-based, RESTful and offers a graphical user interface to enable an easier development as well as advantages in debugging. Because of that, we used Node-RED here, which provides a nice user interface showing a graphical representation of the executed flows, supports automatic deployment and offers a native REST support. The flow is started automatically and processes the situation recognition in predefined time intervals.

The *resource management platform provides sensor data of heterogeneous sources* to be processed by the executable situation template deployed in Node-RED. The resource management platform is currently being implemented using

[7] https://wink.apache.org/ [8] https://github.com/restflow-org/

Eclipse Lyo[9] – the Java-based implementation of the Open Services for Lifecycle Collaboration (OSLC)[10] specification –, however, it is not yet available in our prototype. Same with the sensor registry that could e.g, be realized using a web service with an underlying database to store the sensor's information.

The implementation of the resource management platform *is based on OSLC* because OSLC provides a mature specification that describes how to provide data as uniform REST services. In this paper, we use a push approach from the sensors to the resource management platform and a pull approach from the situation recognition system to the resource management platform. This mixed push/pull-approach is necessary because the situation recognition processes the sensor data independent of the sensors' reaction, i.e., sensor values have to be available at all times not only if pushed by the sensors. The resource management platform consists of (i) OSLC adapters to connect to the sensor data sources, (ii) a data cache to store intermediate data, (iii) an OSLC service provider managing OSLC services, (iv) OSLC services that create, modify or delete REST resources, and (v) the *REST resources themselves that provide data of the connected sensors* to enable uniform accessibility. The architecture of these components is displayed in Fig. 6. Note that the OSLC specification is usually used for the integration of lifecycle tools for software development. For our prototype, we designed an *OSLC-inspired* architecture that transfers the concepts of OSLC to enable the integration of sensor data sources. As a consequence, our design could slightly differ from the OSLC specification. In the following, the components of the resource management platform are described in detail:

OSLC Adapter: An OSLC adapter is used to connect to a sensor's API in order to extract its data. This can be realized using either a push or pull approach. In the pull approach, the adapter requests the data from the sensor, in the push approach, the data is sent directly to the OSLC adapter as soon as the sensor reacts. Note that the pull approach requires a sensor API that caches its data and provides it on request, independent of the sensor's reaction. In the motivating scenario, an adapter for each machine to be monitored has to be created by accessing the machine's sensor APIs and by extracting data, e.g., the CPU load, the currently available RAM or the CPU temperature. This data is stored into a data cache, e.g. a key-value store, to be available on request. Note that details about the binding of the sensors are part of our future work.

OSLC Service Provider: The OSLC service provider represents the entry point of the platform and manages the OSLC services that provide the REST resources. In our approach, we use a single service provider, managing all services.

OSLC Services: OSLC services are responsible for the *on-demand* creation, modification and removal of REST resources. Each service represents an object to be monitored. This object may contain an arbitrary number of sensors. For each sensor of an object, an OSLC REST resource providing the sensor data is created by these services.

REST Resources: The REST resources represent the interface to the user of our OSLC platform, that is, the situation recognition. The data extracted by the

[9] http://eclipse.org/lyo/ [10] http://open-services.net/

Table 1. Runtime Measurements of the Prototype

Measurement	ST Transformation	ST Deployment	ST Execution
1	219 ms	141 ms	6 ms
2	219 ms	126 ms	6 ms
3	234 ms	125 ms	5 ms
4	203 ms	141 ms	5 ms
5	204 ms	140 ms	6 ms
∅	215,8 ms	134,6 ms	5,6 ms

OSLC adapters is accessed through the data cache and is made available through a RESTful interface, providing the uniform methods GET, PUT, POST and DELETE that can be invoked using the Hypertext Transfer Protocol (HTTP). The actual implementation of the resources is defined in the corresponding OSLC service. In our prototype, only the GET and DELETE methods are relevant to either receive the sensor data or delete the resource if a sensor is deregistered.

Currently, the SitRS prototype has the following limitations: Firstly, it is not yet possible to compare sensor values with each other. It is only possible to compare sensor data with fixed values. Secondly, no concept of time exists because it is focus of this paper to recognize only current situations.

5 Evaluation

This section contains the evaluation of our approach by presenting runtime measurements and a load test based on the prototypical implementation.

To conduct the runtime measurements, we used an Ubuntu image hosted on Openstack[11] with 8 GB RAM and 8 Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz CPUs for our measurements. We measured the runtime of the situation template transformation, the deployment and the execution, separately. The situation template we used for these measurements monitors a remote machine, modeled as shown in the example in Fig. 4. All in all, this situation template contains 8 nodes to be mapped, deployed and executed. Table 1 shows the measurement results. These measurements are based on the transformation, deployment and execution of a single situation template. Our measurements are used as proof of concept that the introduced steps are processed in a reasonable time.

We further executed a load test to check how many situation templates can be transformed, deployed and executed in parallel inside a single runtime environment, using the same situation template as above. The results are shown in Table 2. As displayed, the runtime highly increases with increasing situations to be monitored in parallel. Our measurements show that executing two flows in parallel increases the runtime to 38 ms, when executing ten flows in parallel even to 404 ms. This means, Node-RED produces an overhead when processing multiple parallelized flows. This happens due to Node-RED's inability to process the flows in parallel using multiple threads. Furthermore, the internal execution

[11] http://www.openstack.org/

Table 2. Load Test of the Prototype

# ST	Transformation ∅	Deployment ∅	Parallel Runtime ∅	Sequential Runtime ∅
1	215,8 ms	134,6 ms	5,6 ms	5,6 ms
2	424,4 ms	209,2 ms	37,6 ms	13 ms
5	1093 ms	350 ms	176,4 ms	27 ms
10	2475 ms	659,2 ms	404,4 ms	57 ms

scheduling leads to waiting periods between the execution of nodes. However, when executing the flows sequentially, the runtime is growing approximately linearly as expected, e.g. 10 sequentially executed flows lead to a runtime of 57 ms instead of 404 ms when executed in parallel (cf. column "Sequential Runtime"). In conclusion, when using the Node-RED runtime environment, it would be necessary to implement a self-made runtime scheduler to avoid a poor runtime. As a consequence, we use the Node-RED runtime environment only for our proof-of-concept implementation. In the future, we will implement and compare further execution engines such as CEP-Esper that are suitable for highly parallel scenarios.

6 Summary and Outlook

In this paper, we presented an approach for a situation recognition service called SitRS. This service can be used to integrate real world objects (things) into the internet by deriving their situational state based on sensors. For that, we introduced a method for the recognition of situations based on modeling and executing situation templates. These templates represent a model to define situations by the sensor data to be used as well as the conditions for the situations. The SitRS service transforms this description into an executable situation template that can be automatically deployed and executed in a (cloud-based) execution engine. The architecture of our approach is separated into two components, the situation recognition component and the resource management platform. The situation recognition component is used to execute a data flow that reads sensor data, compares them with predefined values and uses this information to determine if a certain situation occurred. The sensor data is provided by REST. The sensor registry can be used to register new sensor data sources or deregister them if they aren't needed anymore.

As future work, we plan to integrate SitRS into a workflow system in order to realize situation-aware workflows. Furthermore, we plan to use the presented method in a different use case to enable situation recognition in advanced manufacturing (Industry 4.0) environments, i.e., we will introduce and implement an IoT scenario based on *"real"* objects such as production machines. On the technical side, we want to provide additional situation template mapping algorithms for other execution engines such as CEP-systems like Esper and data streaming systems like Odysseus. The current SitRS prototype provides the basis for further development and is available as open source implementation

(https://github.com/hirmerpl/SitOPT). In addition, we plan to enable automatic sensor binding and registration based on ontologies.

Acknowledgment: This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - Grant 610872, project SitOPT.

References

1. Attard, J., Scerri, S., Rivera, I., Handschuh, S.: Ontology-based Situation Recognition for Context-aware Systems. In: Proceedings of the 9th International Conference on Semantic Systems (2013)
2. Brumitt, B., Meyers, B., Krumm, J., Kern, A., Shafer, S.: EasyLiving: Technologies for Intelligent Environments. In: Handheld and Ubiquitous Computing. Springer Berlin Heidelberg (2000)
3. Buchmann, A., Koldehofe, B.: Complex event processing. it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik (2009)
4. Dargie, W., Eldora, Mendez, J., Mobius, C., Rybina, K., Thost, V., Turhan, A.Y.: Situation Recognition for Service Management Systems Using OWL 2 Reasoners. In: Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on (2013)
5. Dey, A.K.: Understanding and Using Context. Personal and Ubiquitous Computing (2001)
6. Großmann, M., Bauer, M., Hönle, N., Käppeler, U.P., Nicklas, D., Schwarz, T.: Efficiently Managing Context Information for Large-Scale Scenarios. In: Proc. of the Third IEEE Intl. Conf. on Pervasive Computing and Communications (2005)
7. Hussermann, K., Hubig, C., Levi, P., Leymann, F., Siemoneit, O., Wieland, M., Zweigle, O.: Understanding and Designing Situation-Aware Mobile and Ubiquitous Computing Systems. In: Proceedings of the International Conference on Computer, Electrical, and Systems Science, and Engineering 2010 (ICCESSE 2010) (2010)
8. Lange, R., Cipriani, N., Geiger, L., Großmann, M., Weinschrott, H., Brodt, A., Wieland, M., Rizou, S., Rothermel, K.: Making the World Wide Space Happen: New Challenges for the Nexus Context Platform. In: Proceedings of the 7th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom '09). Galveston, TX, USA. March 2009 (2009)
9. Meunier, R.: The pipes and filters architecture. In: Pattern languages of program design (1995)
10. Vermesan, O., Friess, P.: Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems. River Publishers (2013)
11. Wang, X., Zhang, D.Q., Gu, T., Pung, H.: Ontology based context modeling and reasoning using OWL. In: Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on (2004)
12. Wieland, M., Schwarz, H., Breitenbücher, U., Leymann, F.: Towards Situation-Aware Adaptive Workflows. In: Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom) (2015)
13. Zweigle, O., Häussermann, K., Käppeler, U.P., Levi, P.: Supervised Learning Algorithm for Automatic Adaption of Situation Templates Using Uncertain Data. In: Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human (2009)