Assisting Sensor-Based Application Design and Instantiation Using Activity Recommendation

Zahra Movahedi1,2,*, Walid Gaaloul1, Sami Bhiri1 and Bruno Defude1

1Computer Science Department, TELECOM & Management SudParis, UMR 5157 CNRS Samovar; Evry, France
2Zodianet Company, Palaiseau, France
*Corresponding author: zahra.movahedi@telecom-sudparis.eu

Recently, Sensor/Actuator Networks have become an emergent technology for various application areas such as security and surveillance applications, traffic control, logistics, energy control in public and private buildings, etc. Designing and constructing new applications using these technologies remain, however, a challenging task. Indeed, finding the relevant sensors and actuators, and combining them in a proper way in order to achieve a specific goal is not an easy task and requires several skills from different stakeholders. Moreover, sensor environments are inherently highly dynamic. Furthermore, current applications are in general tightly coupled to the underlying infrastructure which hampers their reuse and flexibility to changes. In this paper, we present a process-oriented and service-based approach for supporting the development of adaptive sensor-based applications. Our approach decouples application logic from its implementation. A design-time model is first specified, as a flow of activities, which is then deployed in a particular environment. Decoupling the application logic from its implementation enables, on one hand, to foster the reuse at the application level and, on the other hand, to adapt the same application to different environments and situations. We propose also an activity recommendation technique to provide assistance to application designers by recommending to them activities that have been used in a similar compositional context. Our approach has been prototyped using service standards such as BPMN within the context of the VITRO European project and validated by several use cases.

Keywords: sensor-based applications; service composition; business process; service adaptation; activity recommendation; neighbourhood context

Received 4 October 2013; revised 1 July 2014
Handling editor: Mohamed Jmaiel

1. INTRODUCTION

Sensor/Actuator Networks (S/A Networks) are becoming increasingly emergent in a variety of applications such as home automation, traffic monitoring, healthcare, environment monitoring, etc. These applications mostly require the composition of a large number of distributed and heterogeneous sensors that report data continuously from physical or environmental phenomena (temperature, light, intensity, location, etc.) of a feature (a lake, a home, etc.).

Owing to the large-scale, heterogeneous and distributed nature of S/A networks, designing and constructing value-added applications is a difficult task. Numerous standards and approaches try to address this issue. The Open Geospatial Consortium is specifying standards for interoperability interfaces and metadata encodings enabling the integrating of data from various S/A networks and the deployment of sensor web applications over the Internet [1].

Although these standards are very important and resolve basic problems, they operate, however, on a low technical level. They fail short when it comes to defining the business logic of sensor-based applications. Without loss of generality, sensor-based applications require the gathering, integrating and processing data from different sources and definition of database decision-taking operators that trigger certain actions as a reaction to the occurrence of certain situations. Developing such applications requires the combining the relevant sensors.
and actuators in a proper way in order to achieve the desired goal. Moreover, sensor environments are inherently highly dynamic which require appropriate mechanisms enabling the application to adapt properly to changes.

In this paper, we are interested in supporting the development (design and implementation) of adaptive sensor-based applications that react properly to sensor environment changes. We present a process-oriented and service-based approach for supporting the development of adaptive sensor-based applications. Our approach decouples between the application logic and its implementation. A design-time business process (BP) is first specified, as a flow of activities, which is then deployed on a particular environment. An activity is a conceptual description of a sensor, actuator, processing data or decision-making service. Decoupling the application logic from its implementation enables, on one hand, fostering the reuse at the application level and, on the other hand, adapting the same application to different environments and situations.

Process design is the initial and key step that impacts on the success of the desired sensor application. Designing process models from scratch is a labour-intensive and time-consuming activity, especially when such models are required to be detailed to support the development of software systems [2]. It would be inefficient if every time a company engages in modelling and re-designing its process; it does it from scratch without consideration of previous design experiences, best practices or how others perform similar tasks.

Several consortia and vendors have defined so-called reference process models that capture proven practices and recurrent business operations in a given domain. They are designed in a generic manner and are intended to be individualized to enable systematic reuse of proven practices across process (re-)design projects. Other industrial and research efforts on facilitating BP design rely on BP discovery mechanisms.

These techniques operate on a coarse-grained level by recommending entire BP models which may be confusing to the designers. Moreover, current techniques based on reference process models lack appropriate automation support hampering their effective application. We present in this paper a complementary technique that operate on fine-grained levels by recommending a list of activities that are related to a specific position in the ongoing designed process. By providing a list of alternatives for a specific position in a process, our approach can help to adjust it in order to improve it or adapt it to the changing environment.

Different from current recommendation techniques, our approach does not require extra information about the position to be filled in, or the activities to be recommended. Instead, our approach takes into account the similarity between compositional contexts of activities which can be computed from existing models.

To illustrate the approach features and show its feasibility, a real use-case scenario is used through this paper. The aim of this scenario is to ensure building automation (alarm and security, energy control), focusing on a fire detection scenario.

The rest of the paper is organized as follows: in Section 2 we describe the main phases of a sensor-based applications lifecycle we are considering in our approach. In Section 3, we introduce our process-oriented and service-based models for specifying sensor-based applications. Section 4 elaborates our proposed methods and algorithm for implementing and executing a sensor-based BP. Section 5 details our recommendation technique for the adaptation and the design of sensor-based applications [3]. In Section 6, we describe the prototype we have done in the context of the VITRO project [4]. Section 7 positions our approach within relevant related work. We draw in Section 8 some conclusions and present future directions.

2. SENSOR-BASED APPLICATION LIFECYCLE

In this section, we present a sensor-based application lifecycle (Fig. 1) which comprises four phases as follows:

1. **Design phase**: captures the application logic as a BP, which is defined as a flow of activities. An activity is defined by a semantic specification that declaratively denotes a (set of) sensor(s)/actuator(s) or an operation (synchronization, manipulation or decision). The designer specifies new business processes either from scratch or using the activity recommendation technique (Section 5) which assists in facilitating the design of new models. This technique provides a list of alternatives for a specific position in the ongoing designed process which can be useful in either expanding a designed process to provide new functionalities, or creating new process variants.

2. **Deployment phase**: implements a BP while taking into account semantic specification of activities. During the implementation of a BP, we maintain the mapping

![FIGURE 1. Sensor-based application lifecycle.](http://comjnl.oxfordjournals.org/)

**SECTION C: COMPUTATIONAL INTELLIGENCE, MACHINE LEARNING AND DATA ANALYTICS**
**THE COMPUTER JOURNAL, 2014**
between activities and the corresponding concrete services that implement them. The implementation of a BP sometimes requires adding extra operation activities. For instance, if a BP specifies the average temperature in Celsius, a converter activity will be needed to convert the measured data to the specified unit if necessary.

3. **Execution phase**: in addition to running an activity, we also ensure its proper execution and the respect of the semantic specifications. In case of a dysfunctional service has been detected, an adaptation mechanism is launched. We go back to the design phase, we use the same recommendation technique to specify a new model by replacing the activity that the dysfunctional service implements. Then we deploy the new model and execute it.

4. **Analysis phase**: this phase analyses the application execution logs in order to analyse and improve them. In this paper, we do not deal with this phase.

3. **COMPOSITION MODEL**

In this section, we introduce a process-oriented and service-based composition model. This model explains the different levels for specifying sensor-based applications. In the following, we present firstly an overview on this model (Section 3.1) and then we detail the notations of activity (Section 3.2) and BP (Section 3.3) in the context of S/A networks. At the end of this section, following our model, we present a use-case scenario that will be developed through the paper to present different phases of a sensor-based application lifecycle (Section 3.4).

3.1. **Overview on composition model**

The proposed composition model is structured on three levels: concrete service level, activity level and BP level (Fig. 2).

1. **Concrete (low-level) service level**: is the lowest level of the composition model, which relies on the different components of an S/A network. In general, an S/A network is composed of heterogeneous (different types of sensors and actuators such as temperature, light, alarm, etc.) and dynamic (sensors may appear and disappear at any time) nodes (sensors or actuators) as well as a gateway (GW). The role of the GW is to offer interconnection between an S/A network and the upper layers. In this sense, each GW contains a uniform API allowing the upper layers to retrieve and invoke sensors/actuators in the same way as web service. It contains also a registry describing capabilities of sensors and actuators. We have chosen SensorML [5] to define these capabilities.

2. **Activity level**: aims to abstract the specificities of the concrete service level vis-à-vis the requester. Applications are defined as business processes referencing activities and not directly as concrete services. This allows the decoupling of the application logic from its implementation. Decoupling the application logic from its actual implementation has many benefits. First, it enables reuse when developing new applications. Secondly, it enables implementing the same application on different environments and according to different user requirements. Activities are defined by semantic specification denoting declaratively a set of concrete services. The created activities are stored in a repository in this level and the BP provider uses them to create business processes.

3. **BP level**: an application is modelled as a BP (a flow of activities) which is then implemented and executed on a particular environment binding to the actual concrete services in level 1. This allows creating an arbitrary complex process and reusing existing activities. These orchestrations are defined as business processes that are stored in a repository. The next section explains in more detail the notion of activities and BP.

3.2. **Activity**

Formally, an activity is defined as a black box with three attributes (Fig. 3): Activity =<IdA, typed input/output, semantic specification>.

![Composition model](http://comjnl.oxfordjournals.org/)
(i) \textit{IdA} is a unique identifier for an activity.
(ii) \textit{Input} specifies the input values which have to be sent to an activity to perform its tasks and \textit{Output} specifies the results produced by an activity. The inputs and outputs are typed, allowing checking of the correctness of an orchestration.
(iii) \textit{Semantic specification} which describes the activity semantically as a set of conditions \{(property, operator, value)\}. \textit{Property} references a concept in a domain ontology which has to be defined for each class of applications, \textit{operator} references a classical comparison operator (equality, inequality, \ldots) and \textit{value} represents either a constant or a concept in a domain ontology.

The domain ontology is used to describe semantic attributes and is a top-level ontology structured in six main concept hierarchies: device (sensors and actuators), measure (contains all the concepts related to measures coming from sensors, i.e. type of measure, unit of measure, it should be compatible with the description language of sensors in our case SensorML [5]), action (contains all concepts related to actions that can be done by actuators, i.e. switch on/off), operation (contains all concepts related to operations for end-users, i.e. monitoring, controlling), time (contains concepts related to time) and location (contains concepts related to location, i.e. building, room). Of course, there are semantic relationships between these hierarchies (i.e., an action can contribute to an operation, a sensor produces values in a given unit of measure). The input and output are in a stream data nature. We define a stream data as following.

**Definition 3.1 (Stream element).** An element \( s \) contains one typed measurement attribute \( m \), and one timestamp attribute \( \text{tmstmp} \). 

\[ s = \langle m, \text{tmstmp} \rangle. \]

Each attribute \( m \) belongs to the measurement concept \( M \) and \( \text{tmstmp} \) to the time concept \( T \). By \( T \) we denote a totally ordered set containing discrete points which represent different moments in time, \( \text{tmstmp} \in T = \{t_0, t_1, t_3, \ldots\}, T \subset \mathbb{N}_0 \).

**Definition 3.2 (Stream data).** A stream \( S = \{s_0, s_1, \ldots, s_n, \ldots\} \) is a sequence of elements \( s_i \) ordered by timestamp value. In addition, elements \( s \) also have linear positional ordering, i.e. the element \( s_n \) is the \( n \)th element of the stream \( S \).

Based on the activity definition, we categorize three kinds of activity as follows (sensor activity, actuator activity and operation activity):

1. Sensor activity: each sensor activity denotes a set of sensor services. The schema of a sensor activity is as follows: \( <\text{IdSA}, \text{typed output}, \text{semantic specification}>. \)

   (a) \textit{IdSA} attribute specifies a unique identifier for a sensor activity.

   (b) \textit{Output} attribute which specifies the measurements produced by the sensor activity. The outputs are a typed data stream. A data stream is defined as a (infinite) set of tuples ordered by timestamp value. Each tuple contains a measurement value and a timestamp attribute.

   (c) \textit{Semantic specification} attribute which uses concepts from device, measure, location and time parts of the domain ontology. For example, one can define the type and unit of measurements of the sensor (concepts from measure) and the frequency (concept of time).

2. Actuator activity: each actuator activity denotes a set of actuator services. The general schema of an actuator activity is as follows: \( <\text{IdAA}, \text{typed input}, \text{semantic specification}>. \)

   (a) \textit{IdAA} attribute specifies a unique identifier for an actuator activity.

   (b) \textit{Input} attribute which specifies the input values of an actuator activity to perform a task. They are considered to have a type which is described by an action concept. For instance, for a siren actuator, the action may be switch on/off.

   (c) \textit{Semantic specification} attribute which uses concepts from device and action parts of the domain ontology. It should be noted that some devices are sensor and actuator at the same time. In our model, we described them as two devices: a sensor and an actuator.

3. Operation activity: operations are needed to orchestrate sensor and actuator activities. An operation activity describes an operation service and specifies the way the input of an operation is transformed into the output.

   In the following, we define several kinds of operation activities that are synchronization, manipulation and decision. New operations can be added by an administrator if needed. More complex operations can also be defined by composing several operators.

   (a) \textit{Synchronization}: In this category, we have operators operating on a single stream (\( \text{Syn}_{\text{freq}} \) and \( \text{Syn}_{\text{temporal}} \)) and operators operating on multiple streams (\( \text{union} \) and \( \text{join} \)). The \( \text{Syn} \) operators operate on an input stream and produce an output stream with a different
frequency or a different time interval. For synchronization on frequency, the operator $\text{Syn}_\text{freq}(S, F, tf)$ operates on a stream $S$ and is parametrized by a function $F$ specifying how to delete values in the input stream if the target frequency $tf$ is lower than the source one or to add values in the other case. For temporal synchronization, the operator $\text{Syn}_\text{temporal}(S, \text{start}, \text{end})$ operates on a stream $S$ and produces as output all the values of $S$ with a timestamp belonging to the interval $[\text{start}, \text{end}]$.

Join and union operate on multiple streams as input and produce a single stream as output. $\text{Union}(S_1, S_2)$ returns a stream composed by all the tuples of stream $S_1$ and all the tuples of stream $S_2$, ordered by their timestamp. $\text{Join}_{JC}(S_1, S_2)$ returns a stream composed by a set of tuples $t$, where $t$ is the concatenation of a tuple from $S_1$ and a tuple from $S_2$ satisfying the join condition $JC$.

(b) **Manipulation**: in this category, we have operators doing arithmetic operations on streams. We present three of particular interest, which are conversion, filtering and aggregation. The Conversion operator operates on a single stream and produces as output a new stream where values of the source stream are transformed by applying a function. A simple example of conversion is the transformation of a value from a unit of measure to another (change Fahrenheit to Celsius). The Filtering operator operates on a single stream and produces as output a stream composed by all values of the source stream satisfying a particular condition. The aggregation operator operates on a single stream or on multiple streams. It applies an aggregate (average, min, max, . . .) on a window of the input stream(s). The windowing can be defined either by a time interval or a number of values.

(c) **Decision operators**: they are typical domain operators and providers have generally to add new ones to fulfil their needs. A decision operator operates on an input stream (or on another decision operator) makes a decision and produces as output an action for an actuator activity. A simple example of such an operator is the triggering of an actuator if one value from the input stream exceeds a given threshold.

### 3.3. Business process

Sensor-based applications share many similarities with business processes. Both of them can be modelled as directed graphs where nodes correspond to the processing tasks and edges express sequencing order between these tasks. However, stream-data processing applications and business processes have also fundamental differences regarding the paradigm of their execution. A BP model is used as a blueprint that is instantiated every time a new request is issued, generating different threads running in parallel, each one serving a specific request. However, the concept of instance is not quite clear and explicit in data-stream processing application as it is in BP.

Given the purpose of our work, we can model sensor-based applications as BP. First, data-stream applications operate on data belonging to a time-window. In this way, an instance of the same BP model can be created for every occurring time-window. Secondly, the focus of our work is more on (design) models rather than on details of implementing sensor-based applications on a specific stream processing engine. Modelling sensor-based applications as BP would enable reusing and adapting a rich set of techniques developed in the context of BPM.

A BP can be seen from two complementary views. Seen as a black box, a BP can be considered as one (composite) activity and thereafter it can be described as any other activity by its Id, input/output and semantic specification. Seen as a white box, a BP can be considered as a flow of activities. Given the focus of our paper, we limit our definition of a sensor-based BP to the white box view.

There are a number of process modelling languages, e.g. BPMN, WS-BPEL, EPC, YAWL and UML activity diagram. Despite their variances in expressiveness and modelling notation, all these languages are graph-based and share common concepts of tasks, events, gateways, artefacts and resources, as well as relations between them, such as transition flows [6].

In our work, we define a sensor-based BP as a directed labelled graph. Our model is generic enough so that it can capture a BP modelled in different notations and languages. Without loss of generality, we use in this paper BPMN to represent a BP. However, BPs are modelled and stored according to our model given in Definition 3.3.

**Definition 3.3 (Business process).** A sensor-based BP is a directed labelled graph $G_P = (A_P, E_P, L_P)$ in which $A_P$ is the set of activities, $E_P$ is the set of edges between activities and $L_P$ is the set of edge labels:

- $E_P \subseteq A_P \times A_P$,
- $L_P = l(E_P)$, where $l : E_P \rightarrow L_P$.

An edge between two activities captures control flow elements between them. How the function $l$ is defined is detailed in Section 5.2. Figure 4 depicts how the Business Process BP1 given in Fig. 5 is specified according to our model.

### 3.4. Use-case scenario: design phase

In this section, we describe the ‘fire detection’ scenario that is used through this article and detail its design phase. In this
scenario, we consider different kinds of sensors and actuators (CO2, monoxide, light, temperature, fire siren, etc.) across four distinct geographical trial sites (Tanagra, Paris, Patras, Rome), connected via the VITRO middleware platform [4] located in Madrid. The overall objective of VITRO is the realization of a scalable, flexible, adaptive, energy-efficient and trust-aware Virtual Sensor Networking platform, focusing on the reduction of deployment complexity and on advanced interoperability. The end-user can access information from S/A networks, through gateways. As a BP provider, we design a set of business processes. These processes are defined using activities described in Table 1. We use the BP1 (Fig. 5) as a reference scenario in order to describe our approach.

BP1 monitors the temperature and CO2 level. A fire siren is activated either directly by a user or when temperature and/or the CO2 level go beyond a certain threshold. This process uses three sensor activities (two temperature sensor activities: SA1 and SA2 and one CO2 sensor activity: SA3), four operation activities (AvG temperature, AvG CO2, User action and Decision) and one actuator activity (siren activation: AA1).

As shown in Fig. 6, we present four other business processes. BP2 regulates a heater depending on the temperature, BP3 activates a fire siren due to a high temperature and monoxide density, BP4 starts a video due to a high monoxide density and motion detection and BP5 activates a fire siren due to a
### TABLE 1. Activities presentation.

<table>
<thead>
<tr>
<th>Activity ID</th>
<th>Activity description</th>
<th>Semantic specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SA_1$</td>
<td>temperature in Celsius</td>
<td>(type, $=$, temperature), (units of measure, $=$, Celsius)</td>
</tr>
<tr>
<td>$SA_2$</td>
<td>temperature in Fahrenheit</td>
<td>(type, $=$, temperature), (units of measure, $=$, Fahrenheit)</td>
</tr>
<tr>
<td>$SA_3$</td>
<td>CO$_2$</td>
<td>(type, $=$, CO$_2$)</td>
</tr>
<tr>
<td>$SA_4$</td>
<td>light</td>
<td>(type, $=$, light)</td>
</tr>
<tr>
<td>$SA_5$</td>
<td>monoxide</td>
<td>(type, $=$, monoxide)</td>
</tr>
<tr>
<td>$SA_6$</td>
<td>detection motion</td>
<td>(type, $=$, detection motion)</td>
</tr>
<tr>
<td>$AA_1$</td>
<td>fire siren</td>
<td>(action, $=$, on/off)</td>
</tr>
<tr>
<td>$AA_2$</td>
<td>heater regulator</td>
<td>(action, $=$, regulation)</td>
</tr>
<tr>
<td>$AA_3$</td>
<td>camera video</td>
<td>(action, $=$, taking video)</td>
</tr>
<tr>
<td>AvG temperature</td>
<td>calculate the average</td>
<td>(type, $=$, temperature), (units of measure, $=$, Celsius), (frequency, $&gt;=5$), (nb of inputs, $&gt;=3$)</td>
</tr>
<tr>
<td>AvG CO$_2$</td>
<td>calculate the average</td>
<td>(type, $=$, CO$_2$), (nb of inputs, $&gt;=2$)</td>
</tr>
<tr>
<td>AvG light</td>
<td>calculate the average</td>
<td>(type, $=$, light), (nb of inputs, $&gt;=2$)</td>
</tr>
<tr>
<td>AvG monoxide</td>
<td>calculate the average</td>
<td>(type, $=$, monoxide)</td>
</tr>
<tr>
<td>Decision</td>
<td>decide an action in terms of inputs</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

#### FIGURE 6. Examples of business processes.
high light and temperature density. These BPs will be used to illustrate the recommendation scenario in Section 5.

4. IMPLEMENTATION OF SENSOR-BASED BP

Once a sensor-based process is designed, it is deployed on an execution engine that takes care of creating an instance for each client request and ensuring the execution of the process instances. In this section, we discuss how a sensor-based process is implemented.

4.1. Implementation of S/A and operation activities

Implementing a sensor-based BP boils down to implementing its sensor, actuator and operation activities (or activity instances depending on the binding mode as explained below). Control and data flows are interpreted by the execution engine. In the context of service-based BP, activities (more precisely activity instances) are carried out by invoking appropriate services. The operation of discovering and invoking the appropriate service that provides the required functionality is called service binding.

A sensor activity (instance) retrieves measures about some features and feeds them (as a data stream) to an operation activity which processes them accordingly. An operation activity may impose constraints on the number of measurement inputs, their reading frequency and measurement unit. Therefore, a sensor activity is implemented by a set of (rather than one) sensor services sensing the same feature and returning potentially different measures in different units and with different reading frequencies. Moreover, additional services may be required to synchronize sensor reading frequencies and convert measurement units. An actuator activity may also be carried out by a set of actuator services performing the same action in different locations. For instance the activity $AA_1$ in our example activates a siren alarm. The implementation of such activity is achieved by firing several sirens, i.e. invoking different actuator services, at different locations. Operation activities are implemented by JAVA services providing the required functionalities. These JAVA services are specified, as annotations to the corresponding activities, before the BP is deployed.

4.2. S/A service binding

We distinguish between two modes for binding services: deployment-time and instantiation-time binding. S/A service binding may occur at deployment time, when the BP is deployed on the execution engine, or at instantiation time, when a process instance is created to answer a given request. Operation services are bound to their activities at deployment time at the latest. However, the validation of their input constraints (See below Algorithm 2 lines 16 and 17) is done at deployment or instantiation time depending on the binding mode.

Algorithm 1 shows how a list of services are bound to a given S/A activity independently of the binding time. Each S/A service description is matched to the S/A activity specification. Services that satisfy the activity requirements are selected and added to the list of bound services (lines 6 and 7). The function $sd()$ returns the semantic description of the given service/activity.

Algorithm 1 S/A service binding algorithm.

```
1: Function binding(n) : l
2: input n: S/A activity
3: output l: list of S/A services implementing n
4: l ← ∅
5: for all S/A service s ∈ S/A repository do
6:   if satisfy(sd(s), sd(n)) then
7:     Append(l, s)
8: end if
9: end for
10: return l
```

Depending on the binding mode, the result of the implementation of a service-based BP at deployment time differs. Binding at deployment time requires binding appropriate services to each process activity. This means that the same set of services will be invoked for all instances of the same activity. Secondly, all activities of the BP are bound to services at deployment time. At instantiation time, the engine has just to invoke the already bound services. The result of a BP implementation when services are bound at deployment time is a composite service (directly executable). The implementation of an activity is the invocation of the bound services.

Binding at instantiation time requires binding a service to each activity instance. This means that instances of the same activity may be carried out by different sets of services. At deployment time the activities are not bound to any services. The implementation result is therefore an abstract composite service in which an activity implementations are performed per instance. At instantiation time, the execution engine has to discover and invoke appropriate services.

Deployment-time binding is more suitable in a rather static environment. Indeed, in a dynamic environment where services may appear and go off, service binding and therefore BP implementation may become invalid. In a static environment where the service landscape changes less frequently, deployment-time binding enables ensuring the implementation of the BP before BP instantiation.

Binding at instantiation time is more suitable for a dynamic environment where the service landscape changes frequently.
Service binding at instantiation time enables to having the most suitable implementation for that particular instance.

### 4.3. Implementation algorithm

The implementation phase consists in binding actual concrete services to activities of the BP. This binding has to satisfy all the semantic specifications of the BP activities and potentially additional conditions defined by the BP requester. For example, the BP requester can add conditions on the location of the sensors and actuators. These additional conditions are defined on activities. In some cases, it can be impossible to implement a BP because some semantic specifications are not satisfied. In this case, the recommendation phase is launched in order to modify the initial BP.

Algorithm 2 describes the main steps for implementing a BP where services are bound at deployment time. The algorithm implements first a skeleton of the process based on the flow of activities (line 5). Then it moves to bind services to process activities (lines 6–23).

For each S/A activity, the algorithm calls the binding function (explained above in Algorithm 1) in order to retrieve a list of services that implement the given activity (line 9). If the returned set is not empty, it is added as an implementation of the given activity (lines 10–12).

An operation activity is bound to its corresponding service (specified as annotation to the activity before the BP deployment) only when its constraint in terms of number of input streams is satisfied (lines 16 and 17). Services that synchronize reading frequencies and convert measurement units as requested by the operation activity are added if necessary (line 18).

The variable `notBound` detains the list of activities that cannot be implemented (lines 4, 13, 20). If there is at least one activity that cannot be implemented, then the recommendation procedure is invoked to redesign the BP (lines 24–26).

#### 4.4. Use-case scenario: deployment phase

Figure 7 shows one implementation of our application example illustrated in Fig. 5. The sensor activities $SA_1$ and $SA_2$ which denote temperature sensors are implemented by three concrete services ($S_1$, $S_2$ and $S_3$). These services satisfy the semantic specifications of $SA_1$ and $SA_2$. They return temperature streams and feed them to the operation activity $Avg$ Temperature. $Avg$ Temperature requires having at least two temperature values to compute their average. In this example, $S_1$, $S_2$ and $S_3$ do not have all the required frequency ($\geq 5$) and the units of measurement (Celsius), as specified in $Avg$ Temperature semantic specification (see Table 1). Therefore, to solve this inconsistency, new services have been added to synchronize the temperature stream frequency and convert $S_3$'s measures. One can note that, even if only two sensor services are required by the $Avg$ Temperature, all the possible bindings are constructed in order to offer more possibilities for further operations of the BP or at execution time. The sensor activity $SA_3$, which denotes CO2 sensors, is implemented by two concrete services $S_4$ and $S_5$, as depicted in Fig. 7. The measures returned by these services are then processed by the operation activity $Avg$ CO2.

### 5. ACTIVITY RECOMMENDATION

#### 5.1. Motivations

Designing sensor-based applications is a challenging task. Indeed, finding the relevant sensor and actuator services, and combining them in a proper way in order to achieve a specific goal is not an easy task and requires several skills detained by different stakeholders. Moreover, sensor environments are inherently highly dynamic. Sensor-based applications need to adapt to changes that may impact its proper function.

---

1Supporting additional conditions requires that binding occurs at instantiation time.
Inspired from previous work proposed in [3], we propose a complementary technique that operates on fine-grained levels by recommending activities for given positions in the ongoing modelled process. Concretely, our approach can recommend activities, which have similar neighbourhood context with a failed activity, in order to react to a dysfunctional sensor or actuator. Our recommendation approach can also be used at design time in order to help the designer to find activities that share similar neighbourhood contexts for given positions in the ongoing modelled process. This neighbourhood context, which represents the activity’s compositional context, is defined as a BP fragment around the activity. For a selected activity, we match its neighbourhood context with the neighbourhood contexts of other activities. The matching between two neighbourhood contexts is scored by a similarity value. Then, based on the similarity values, we recommend to the process analyst activities that have the highest similarity values.

There are rationale and benefits behind using the neighbourhood context as it informs us about the activity’s behaviour and thereafter can unveil its business context. By using it, our objective is two-fold: (i) taking into account the BP fragment surrounding an activity as an input which would help to focus on specific parts of the BP and can avoid the computation complexity problem of BP structure matching and (ii) benefiting from the existing business processes by extracting the implicit knowledge contained in their fragments.

In the following, we detail our approach to generating recommendations for a particular activity/position in a BP. First, we present the concept of activity neighbourhood context (Section 5.2). Then, we detail how we compute the similarities between activities based on the matching of neighbourhood contexts (Section 5.3). In Section 5.4, following our scenario, we show how, for a given activity, we recommend a list of activities based on the computed similarity values. Finally, we discuss the complexity of the activity matching algorithm (Section 5.5).

5.2. Activity neighbourhood context

An activity in a process is a node and the connection between two activities in that process is an edge in the corresponding process graph. On one hand, a connection between two activities is a sequence of connection elements connecting them. On the other hand, activities can be connected in either series (e.g. connected by sequence connection element) or parallel (e.g. split by a parallel connection element) fashion. So, to capture the connections between any two activities in a process and present them in the corresponding process graph, we present those connections in strings of characters and label the corresponding edges in the process graph with these strings. The following presents our solution to label the connection between two activities.

**Definition 5.1 (Connection flow).** Let \( A_P \) is a set of activities and \( C_P \) is a set of connection elements in a business process \( P \); \( e_i \) is connected to \( e_j \) in \( P \), denoted by \( e_i \leftrightarrow P e_j \), if \( e_j \) is performed right after or right before \( e_i \) in \( P \). The label of this relation denoted by \( l(e_i \leftrightarrow P e_j) = e_i e_j \) (if \( e_j \) is performed right after \( e_i \)), or \( e_j e_i \), (if \( e_j \) is performed right before \( e_i \)).

A connection flow from \( a_i \) to \( a_j \), \( a_i, a_j \in A_P \), denoted by \( a_i \rightarrow P f_P \), is a sequence of connection elements \( c_1, c_2, \ldots, c_n \in C_P \) that satisfies: \( a_i \leftrightarrow P c_1 \leftrightarrow P c_2, \ldots, c_n \leftrightarrow P a_j \).\(^2\) \( a_i \rightarrow P f_P \in C_P^P, C_P^P \) is the set of sequences of connection elements in \( P \). The label of a connection flow is the concatenation of labels of connected relations in \( a_i \rightarrow P f_P : l(a_i \rightarrow P f_P) = l(a_i \leftrightarrow P c_1)l(c_1 \leftrightarrow P c_2) \cdots l(c_n \leftrightarrow P a_j) \).

For example, the label of the connection flow from \( SA_3 \) to \( AvG \ CO_2 \) (as shown in Figs 4 and 5) is: \( SA_3 \) ‘sequence’.’sequence’AvG CO2; from AvG temperature

\(^2\)The connection flow from \( a_j \) to \( a_i \) is the inverse of the connection flow from \( a_i \) to \( a_j \).

Intuitively, the closeness between activities is specified by the connection paths connecting them. The shortest path between activities presents their closest relation. To specify the best relations of an activity to others, we define the **neighbourhood context graph** (see Definition 5.2) which presents all shortest paths from an activity to others. Each vertex in the neighbourhood context graph has a number which indicates the shortest path length from it to the associated activity. Vertices that have the same shortest path length are considered to be located on the same layer around the associated activity. Thus, we name the number associated to each activity (vertex) in a neighbourhood context graph **layer number**. The area limited between two adjacent layers is called **zone**. The edge connecting two vertices in a neighbourhood context graph has a number which is called **zone number**.

**Definition 5.2 (Activity neighbourhood context graph).**
A **neighbourhood context graph** of an activity \(a_t \) in a business process \(P\), defined by \(G_P^{a_t} = (V_P^{a_t}, E_P^{a_t}, L_P^{a_t})\), is a labelled directed graph built from the BP graph \(G_P\): \(V_P^{a_t}\) is the set of nodes, \(E_P^{a_t}\) is the set of edges, \(L_P^{a_t}\) is set of edge labels, where:

1. \(V_P^{a_t} = V_P \times N\), \(V_P^{a_t} = \{(a_j, a_j, \nu_P^{a_t}) \mid a_j \in V_P, a_j, \nu_P^{a_t} = SPLG_P(a_i, a_j)\}\), \(SPLG_P(a_i, a_j)\) is the shortest path length of \(a_i \) to \(a_j\).
2. \(E_P^{a_t} = E_P \times N\), \(E_P^{a_t} = \{((a_j, a_j, \nu_P^{a_t})) \mid (a_j, a_k) \in E_P, \nu_P^{a_t} = min(a_j, \nu_P^{a_j}, \nu_P^{a_k}) + 1\}\).
3. \(L_P^{a_t} = L_P\).

For example, the neighbourhood context graphs of the ‘BP1’ process (Fig. 5) and the ‘BP5’ process (BP5, Fig. 6) are presented in Fig. 8.

### 5.3. Neighbourhood context matching

The similarity between two neighbourhood context graphs is computed based on the matching of their connection flows. In this section, we present step by step our computation, including the connection flow matching (Section 5.3.1), the neighbourhood context graph matching (Section 5.3.2), the matching with zone weight consideration (Section 5.3.3).

#### 5.3.1. Connection flow matching

The label of each connection flow between two activities contains the relation information, i.e. connection elements and directions, between them. Each label is presented in pairs of predefined strings, i.e. activity names and connection elements. It can be encoded as a string of characters in which each character represents a pair of predefined strings. We propose to apply the Levenshtein distance (LD) [7] to compute the matching between two edge labels. Concretely, given two connection flows that have respectively labels \(l_l^{P_1}\) and \(l_l^{P_2}\). By \(Encode(l_l^{P_1})\) we denote the function that encodes each pair of strings in the label \(l_l^{P_1}\) by a character. Let \(s_{P_1}\) = \(Encode(l_l^{P_1})\), \(s_{P_2}\) = \(Encode(l_l^{P_2})\) and \(length(s_{P_1}), length(s_{P_2})\) be the lengths of the strings \(s_{P_1}\) and \(s_{P_2}\), respectively. The pattern matching between them, denoted by \(M^{(v_s)}_{l_l^{P_1}, u_f^{P_1}}\), is computed by the following equation:

\[
M^{(v_s)}_{l_l^{P_1}, u_f^{P_1}} = 1 - \frac{LD(s_{P_1}, s_{P_2})}{Max(length(s_{P_1}), length(s_{P_2}))} \tag{1}
\]

#### 5.3.2. Neighbourhood context graph matching

To compute the matching between two activity neighbourhood context graphs, we propose to match all connection flows that connect the same activities and have the same zone number. Particularly, in the first zone, we match connection flows that connect the two associated activities to the same activities in the first layer. For example, in Fig. 8, we match \(a_{i1} f_{BP1}\) and \(a_{i2} f_{BP1}\) and \(a_{i1} f_{BP2}\) and \(a_{i2} f_{BP2}\) (the first zone), \(a_{i1} f_{BP1}\) and \(a_{i2} f_{BP2}\) and \(a_{i1} f_{BP1}\) and \(a_{i2} f_{BP2}\) (the second zone), and so on.

Assume that \(a_t\) is the selected activity in the ongoing design process \(P_t\). Let \(Z_{P_1}^{a_t}\) be the set of connection flows and \(|Z_{P_1}^{a_t}|\) be the number of connection flows in zone \(t\) of the context neighbourhood graph \(G_{P_1}^{a_t}\). The similarity between \(a_t\) and another service \(a_s\) in a process \(P_s\) based on neighbourhood context graph matching within \(k\) zones is equal to the sum of the matchings of all connection flows within \(k\) zones divided by the number of connection flows within \(k\) zones of the neighbourhood context graph of \(a_t\) (Equation 2).

\[
M^k(G_{P_1}^{a_t}, G_{P_2}^{a_t}) = \frac{\sum_{t=1}^{k} \sum_{a_{i1} f_{P_1}, a_{i2} f_{P_2}} M^{(v_s)}_{l_l^{P_1}, u_f^{P_1}}} {\sum_{t=1}^{k} |Z_{P_1}^{a_t}|} \tag{2}
\]

where

\[
M^{(v_s)}_{l_l^{P_1}, u_f^{P_1}} = \begin{cases} 
M_{l_l^{P_1}, u_f^{P_1}}(a_{i1} f_{P_1}, a_{i2} f_{P_2}), & t = 1 \\
M_{l_l^{P_1}, u_f^{P_1}}(a_{i1} f_{P_1}, a_{i2} f_{P_2}), & 2 \leq t \leq k
\end{cases}
\]

As the LD of two inverse strings is equal to the LD of the original strings and the connection flow from \(a_j\) to \(a_i\) is the...
inverse of the connection flow from $a_i$ to $a_j$, the matching between two connection flows $a_i^f$ and $a_j^f$ is computed by $M(a_i^f, a_j^f)$ for $i, j = 1, 2, \ldots, 10$. For example, the neighbourhood context graph matching between $a_3$ and $a_9$ given in Fig. 8 with $k = 2$ is as follows:

$$
M^k(G_{BP1}^{a_3}, G_{BP1}^{a_9}) = \frac{M(a_3, a_9)}{4} + \frac{M(a_4, a_9)}{4} + \frac{M(a_5, a_9)}{4} + \frac{M(a_7, a_9)}{4} + \frac{M(a_8, a_9)}{4} + \frac{M(a_9, a_9)}{4} + \frac{M(a_10, a_9)}{4}
$$

5.3.3. Zone weight consideration
Intuitively, the behaviour of an activity is strongly reflected by the connection flows to its closest neighbours, while the interactions among other neighbours in the further layers do not heavily expose its behaviour. Therefore, we propose to
assign a weight \( w_k \) for each \( k \)th zone, so-called zone-weight, and inject this weight to the similarity computation. Since the zone-weight has to have greater values in a smaller \( k \)th zone, we propose to assign the zone-weight a value computed by a polynomial function, which is \( w_z = (k + 1 - z)/k \), where \( z \) is the zone number \( (1 \leq z \leq k) \), and \( k \) is the number of considered zones around the activity. A connection flow belonging to the \( k \)th zone has a weight \( w_k = 1/k \).

The final matching formula improved with the zone weight consideration is given below.

\[
M^k(G^a_{P_1}, G^a_{P_2}) = \frac{2}{k+1} \times \sum_{t=1}^{k} \frac{k+1-t}{k} \times \sum_{a_i \in f_{P_1} \forall Z_{P_1}(a_i)} M^t(a_i, f_{P_1}, f_{P_2}) \times \frac{|Z_{P_2}(a_j)|}{|Z_{P_1}(a_i)|}
\]

5.4. Use-case scenario: recommendation purpose

The neighbourhood context graph presents the interactions between the associated activity and its neighbours in layers. It induces the behaviour of the associated activity in the process through neighbours. Therefore, the matching between neighbourhood context graphs exposes the similarity in behaviours of the associated activities. In our approach, the higher the matching value, the more similar in behaviour is the activity. To generate recommendations, for each activity in a process, we compute the neighbourhood context graph matching between it and others. Then, we sort the computed matching values in descending order and pick up top-N\(^6\) activities that have the highest matching values for the recommendation.

Back to our use case, we present two scenarios to illustrate our activity recommendation technique. The first one concerns the design phase in order to assisting the design of new business processes and the second one concerns the execution phase in order to adapt the executed business processes to changes.

Design phase: Assuming that the designer wants to model a variant of the business process BP1 where we do not rely on CO\(_2\) measurement to detect fire. So he selects the activity AVG CO\(_2\) and launches our activity recommendation technique. Based on currently defined models, our technique can recommend to him the two activities AVG monoxide and AVG light. Indeed, given the business processes BP3 and BP5, we can see that AVG monoxide and AVG light have been used in compositional contexts similar to the one of AVG CO\(_2\).

It is worthy to note that our approach does not only aim at finding activities that have similar functions with a selected one. Instead, we also aim at retrieving activities that have similar neighbourhood contexts. By recommending activities, our approach not only helps to find suitable activities for a missing position but also helps to find other alternatives for a selected activity. These alternatives can be useful in either expanding a designed process to provide new functionality, replacing existing activities or creating BP variants. In the following, we leverage our recommendation technique to implement fault-tolerant techniques enabling sensor-based applications to adapt to component service failures.

Execution phase: Assuming that during the execution of BP1 (in fact, the deployed model depicted in Fig. 7), the service \( A_1 \) has some execution problems and needs to be replaced. Based on the correspondence table (which maintain the matching between the activities and their implementation service), we trace back the failure to the activity \( AA_1 \) at the BP level. Using the same recommendation technique, the system will recommend \( AA_2 \) and \( AA_3 \) as potential replacements to \( AA_1 \) (for the same reason explained above). The new defined process will then be deployed and executed.

5.5. Activity matching algorithm complexity

The process fragment surrounding an activity is presented by a sub-BP graph. Therefore, the matching problem would become a graph matching problem which was proved to be an NP-completeness problem [8]. However, in our case, we know the root points of the graph comparison, which are \( a_i \) and \( a_j \), and we match only the same pairs of activities in both BP graphs; thus we avoid the NP-completeness problem of the original graph matching.

On the other hand, only the flows connecting common activities in two adjacent layers are taken into account for the matching computation. So, by using queues (data structure) to store the common neighbours and track them from the nearest zones to the furthest zones, we avoid the redundant checking of unrelated neighbours. On the other hand, since the number of common neighbours of two arbitrary activities is not great, our algorithm can run fast in computing the complete matching of two activities. Without optimization, the worst case of this algorithm’s computation time is \( O(n_S \times n_C \times n \times k) \), where \( n_S \) is the number of activities, \( n_C \) is the number of business processes, \( n \) is the maximum number of common activities of two arbitrary activities and \( k \) is the number of considered zones. The worst case only happens when all the business processes in the database are exactly the same.

Also noteworthy the number of connection flows connecting two activities. This number can be \( >1 \), which means that there can be more than 1 connection flows connecting two activities. So, we have to take into account all matching cases of those connection flows (bipartite graph matching problem). However, this rarely occurs in a BP and this number is not so great (in our dataset, the maximum value is 7). Therefore, this problem is negligible as it does not impact much on the computation time.
6. IMPLEMENTATION

In this section, we describe the prototype we have developed in the context of the VITRO project. This implementation is based on Bonita[^7], which is a BP software environment, including a BP designer, BP repository, BP browser and BP engine. The global environment is structured in three tiers (Fig. 9). The first one is the S/A Network, which provides the S/A data and supports interactions with the second tier using gateways. The second tier is the middleware tier responsible for describing and invoking actual S/A services implementing BP activities. This tier is implemented by the VITRO middleware. The third tier corresponds to our software. It implements the sensor BP approach described in the previous sections. It is based on the Bonita environment. Moreover, a set of operation activities has been implemented using Java language. Connections to the middleware layer are ensured by connectors. The Database connector interacts with the S/A description database to select and bind an activity to appropriate S/A services considering semantic specifications.


The Web service connector is in charge of invoking S/A services to get data from sensors or activate an actuator.

This environment can be used by two kinds of users. The first one is the **Business Process Provider (BP provider)** who is the actor responsible for designing Business processes for use by end-users. In our implementation, the BP provider uses BPMN 2.0 standard in the BP design environment of Bonita. The second type of user is the **End-user** who selects and invokes the business processes provided by BP providers.

As described in the design phase of the sensor-based application lifecycle (Section 2), the BP provider can design a BP either from scratch or along with recommendation assisting (step 1, 1.a, 1.b). For each S/A activity, the BP provider defines semantic specifications. The BP repository will be then updated, registering the new designed BPs (step 2).

An end-user selects and deploys a BP using the browser (step 4). Then the system has to interact with the middleware layer to associate an implementation for all activities. This task is done either by a deployment-time method (step 5a) or by a instantiation-time one (step 5b). The deployment-time method analyses the whole BP and transforms it into an implemented...
BP using the S/A description database (step 6, 7). Then this implemented BP is executed by the BP engine, invoking web services from the middleware layer (step 8). The instantiation-time method uses the same principle but processing the BP activity per activity. In both methods, if an activity becomes dysfunctional, the recommendation component is invoked (step 6, 10) to replace the dysfunctional activity (more details in Section 4).

7. RELATED WORK

Compton et al. [9] provide a detailed survey on the different proposed semantic specifications of sensors. Many of these approaches focused on sensor meta-information. Recently, the W3C SSN-XG group has made an effort to provide a domain-independent ontology compatible with existing sensor standards such as SensorML. This contribution will be considered in the evolution of our work.

Service composition has been the subject of many research works in the web services domain (see [10, 11] as good surveys). However, due to the different characteristics of S/A network compared with traditional web service (high dynamicity, resource constraints, heterogeneity), new approaches for service composition in the S/A network are then required.

A dynamic composition of sensor services is proposed in [12]. This approach proposes to model the composition as a data flow graph enhanced with metadata information. They propose then two heuristic algorithms (bottom-up and top-down) to map the composition graph to the appropriate sensor services. Their aim is to optimize the cost of service composition (communication cost, processing cost) considering both user preferences and service metadata information. However, this paper does not provide a clear presentation on the service composition development process.

In [13], the authors present OASiS, a service-oriented middleware for wireless sensor networks. Its main goal is to facilitate the design, development and implementation of applications. Like our proposal, OASiS uses a multi-layer development process. Applications are defined as dataflows (modelled as a graph of services). Services are then dynamically discovered and have to verify constraints. Compared with our work, OASiS does not use semantic description to describe and select services. It does not also explicitly support operations in the graph definition.

Reuse in BP modelling is a hot topic. Several consortia and vendors have defined so-called reference process models that capture proven practices and recurrent business operations in a given domain. They are designed in a generic manner and are intended to be individualized to enable systematic reuse of proven practices and recurrent business operations in a given domain. They are designed in a generic manner and are intended to be individualized to enable systematic reuse of proven practices and recurrent business operations in a given domain.

They recommend to the designer a list of BPs similar to the ongoing designed one. These techniques operate on a coarse-grained level by recommending entire BP models which may be confusing to the designers, especially when the process models are relatively large. Moreover, it is quite often that process designers have a partial idea of the process model and require assistance to complete missing parts.

Some existing approaches [16–18] target hastening the design phase by retrieving a process to the current designed process from repositories. They proposed either ranking existing BP models for similarity search [16, 19], or measuring the similarity between them [17, 18, 20] for creating new process models. Several consortia and vendors have defined so-called reference process models that capture proven practices and recurrent business operations in a given domain. They are designed in a generic manner and are intended to be individualized to enable systematic reuse of proven practices across process (re-)design projects. Other industrial and research efforts on facilitating BP design rely on BP discovery mechanisms. These techniques operate on a coarse-grained level by recommending entire BP models which may be confusing to the designers. Moreover, current techniques based on reference process models lack appropriate automation support, hampering their effective application.

Business processes in reality consist of a large number of activities and flow connections; therefore, recommending to the designer a list of business processes can make him confused and make it hard for him to detect how the business processes are similar and which parts should be inherited from the recommended processes to use for his current design. In addition, computation on the whole process leads the existing approaches to the graph matching problem which is NP-complexity [8] and they have to deal with the trade-off among the complexity, accuracy (efficiency) and system performance [16, 19]. In our approach, we focus partially on the BP and take into account only the activity neighbourhood context for recommendations. Consequently, we retrieve related activities without facing the NP-complexity problem (Section 5.3).

Thomas Gschwind et al. [21] applied workflow patterns for BP design. They aimed at helping business users to understand the context and apply patterns during the editing process. In our work, we help designers better design a BP by automatically recommending activities that have similar contexts instead of patterns (Section 5.4).

8. CONCLUSIONS AND PERSPECTIVES

In this paper, we presented a process-oriented and service-based approach for developing sensor-based applications. Our approach decouples the application logic from its implementation. An application is first modelled as a BP, which is then deployed on a particular environment. Decoupling
the application logic from its actual implementation has many benefits. First, it enables reuse when developing new applications. Secondly, it enables implementing the same application on different environments and according to different user requirements.

To assist designers in specifying new sensor-based applications, we proposed an approach based on recommending activity to given selected positions in the ongoing modelled process. This recommendation technique is based on similarity between compositional contexts the different activities have been used in.

Moreover, the same recommendation technique enables providing a fault-tolerant mechanism enabling a sensor-based application to adapt to changes due to sensor failures or constraint violation. In current approaches, adaptation is ensured by replacing dysfunctional sensor services. The limitation of these techniques is that in case no sensor service is available, the adaptation fails. Our approach enables overcoming this problem by operating on the BP level, rather than on the deployed model. Ensuring the adaptation at the abstract model enables considering other activities or replacing the implementation of a whole fragment and not just one service as current approaches do.

Our approach has been validated by use cases that have been defined in the context of the European project VITRO. We have implemented a prototype for modelling sensor-based applications using our recommendation technique. Business processes are defined using BPMN. We use a Bonita engine to execute them.

As future work, we intend to investigate a service-oriented approach to integrate data-stream and complex-event processing into process-oriented applications. We intend also to investigate the co-existence of connection flows in business processes, as well as the number of times that an S/A service is used in order to refine our matching algorithm. We also aim at extending our recommendation approach to mine the process models from event logs. The mined process models can be then used as inputs for building the configurable process fragments to assist the BP designers.

FUNDING
This work has been partly performed under Project FP7 ICT-257245 VITRO, funded by the European Commission.

REFERENCES


