An Approach of Early Disease Detection using CEP and SOA

Juan Boubeta-Puig, Guadalupe Ortiz, and Inmaculada Medina-Bulo
UCASE Software Engineering Group
Department of Computer Languages and Systems, University of Cádiz
Cádiz, Spain
Email: {juan.boubeta, guadalupe.ortiz, inmaculada.medina}@uca.es

***Abstract*—Service-Oriented Architectures (SOAs) have emerged as an efficient solution for modular system implementation, allowing easy communications among third-party applications; however, SOAs are not suitable for those systems which require real-time detection of significant or exceptional situations. In this regard, Complex Event Processing (CEP) techniques continuously process and correlate huge amounts of events allowing to detect and respond to changing business processes. In this paper, we propose the use of CEP in SOA scenarios to facilitate the efficient detection of relevant situations in heterogeneous information systems and we illustrate it through the implementation of a case study for detecting early outbreaks of avian influenza. Results confirm that CEP provides a suitable solution for the case study problem statement, significantly decreasing the amount of time taken to generate a warning alarm from the occurrence of an avian influenza outbreak and thus reducing disease impact.**

***Keywords*-CEP; complex event patterns; SOA 2.0; ESB; public health.**

I. INTRODUCTION

In recent years, Service-Oriented Architectures (SOAs) have emerged as an efficient solution for the implementation of systems in which modularity and communication among third parties are key factors. This fact has led to the increasing development of distributed applications made up of reusable and sharable components (services). These components have well-defined platform-independent interfaces, which allow SOA-based systems to quickly and easily adapt to changing business conditions. However, these architectures are not suitable for environments where it is necessary to continuously analyze all the information flowing through the system, which might be a key factor for an automatic and early detection of critical situations for the business in question.

This limitation may be solved by the joint use of Complex Event Processing (CEP) [1] together with SOA. CEP provides a set of techniques for helping to make an efficient use of Event-Driven Architecture (EDA), enabling it to react to multiple events under multiple logical conditions [2]. In this regard, CEP can process and analyze large amounts of events and correlate them to detect and respond to critical business situations in real time; in this scope event patterns are used to infer new more complex and meaningful events. These events will help to make decisions when necessary.

Currently, the integration of EDA and SOA is known as event-driven SOA (ED-SOA) or SOA 2.0 [3], an extension of SOA to respond to events that occur as a result of business processes. SOA 2.0 will ensure that services do not only exchange messages between them, but also publish events and receive event notifications from others. For this purpose, an Enterprise Service Bus (ESB) will be necessary to process, enrich and route messages between services of different applications. Thus, combining the use of CEP and SOA, we may detect relevant events in complex and heterogeneous systems, i.e., CEP will let us to analyze and correlate events in real time SOA 2.0.

To our knowledge, no architecture providing an appropriate and efficient integration of SOA, EDA, CEP and the detection of complex patterns has been proposed yet: there are proposals that use non-standard approaches to the integration of SOA and EDA [4], [5], while others use rule engines [6]. Implementations using rule engines are slower and less efficient in handling and receiving notifications, compared to those using CEP engines. Also, these approaches do not take into account that the system may have to handle a mass of events at any given time, causing a strong impact on system performance.

In this paper we propose an approach for the integration of SOA 2.0 and CEP in order to ease complex events detection in SOA scenarios. Showing the advantageous of using a CEP engine to facilitate an efficient detection of relevant situations is the main aim of this paper. Moreover, detection will be more efficient if the ESB prioritizes events by type, avoiding the bottleneck effect in the engine since CEP engine will analyze, firstly, higher priority events received at one particular point in time. In addition, this approach differs from others in the use of NoSQL (Not only SQL) databases [7], an emerging database management system based on key-value relationships, which is easily horizontal scalable and efficient for managing huge amounts of data.

In order to illustrate our proposal, a case study for detecting early epidemic outbreaks of diseases is also described in this paper. The case study will be implemented according the proposed technologies and will be evaluated through a simulation scenario.

The rest of the paper is organized as follows. In Section II

 ISBN: 978-1-61208-152-6

we describe the main features of CEP and compare them with SOA's, it is followed by the proposed solution for their integration in Section III. Then, a case study for real-time detection of epidemic and pandemic cases of influenza is explained, implemented with the proposed technologies and tested in Section IV. Afterwards, in Section V our approach is discussed and in Section VI related approaches for the integration of CEP and SOA are summarized and compared to the one proposed in this paper. Finally, conclusions and future work are presented in Section VII.

II. CEP Background

CEP [1] is a technology that provides a set of techniques for helping to discover complex events by analyzing and correlating other basic and complex events. A basic event ocurrs at a point in time and it is indivisible and atomic, while a complex event can happen over a period of time, it is aggregated from basic or other complex events and contains more semantic meaning. Some of these techniques are: detecting causality, membership or timing relationships between events, abstracting event-driven processes and detecting event patterns. Therefore, CEP allows detecting complex and meaningful events, known as *situations*, and inferring valuable knowledge for end users.

The main advantage of using CEP to process complex events is that the latter can be identified and reported in real time, unlike in traditional software for event analysis, therefore reducing the latency in decision making.

Thus, CEP is a fundamental technology for applications that (1) must respond quickly to situations that change rapidly and asynchronously and where interactions do not have to be transactional, (2) must support management by exception, (3) must react rapidly to unusual situation and (4) require loose coupling and adaptability [8].

CEP has some similarities and differences with SOA. The main similarity is that both approaches provide modularity, loose coupling and flexibility. Some of the main differences are shown in the following lines:

- On the one hand, SOA interactions are based on services (a user must know the service producer and interface in advance in order to send requests to it). On the contrary, event-driven CEP is reactive and more decoupled since events are generated by event producers and consumers are responsible for intercepting and processing them.
- On the other hand, while SOA processes use events to drive control flow [9] (these processes can both send and receive events), CEP engines continuously analyze and correlate these events to assess if they meet the conditions defined in any of the event patterns stored in them.

III. Our Proposal in a Nutshell

We propose a solution based on the integration of CEP and SOA 2.0. A CEP engine is the key element of the integration, which will facilitate the efficient detection of relevant situations in heterogeneous information systems.

Event producers can be Web services, applications and sensors. Some of these applications are Web applications that allow users to interact with information management systems and legacy applications. Sensors are devices that monitor the environment to capture information (temperature, light, rain, etc.), which is then transmitted to the system using the controller integrated into the mentioned sensors.

These events are then published in the ESB and stored in a NoSQL database to be used as historical events.

Events are sent in parallel to the database management system for their storage as well as they are sent to event streams of a CEP engine. This engine will contain event patterns specifying the conditions to identify relevant situations and the actions to be carried out. Some of its functions are: filtering events (deleting irrelevant events from event streams), correlating and merging events from different event streams (complex events could be created) and aggregating them (grouping events). The generated complex event will be published immediately into the ESB.

Finally, these events will be notified to the event consumers that have subscribed to them. These consumers can be Web services, applications (such as dashboards for displaying alarms) and/or actuators that perform some action (switch on/off, open/close, etc.) on a specific device.

IV. Case Study

In recent decades the globalization has caused a huge increase of people movements between countries resulting in a dramatic increase of the impact of emerging disease epidemics. This situation is becoming a major threat to life, safety and the world economy [10]. This fact motivated the decision of implementing a case study to detect avian influenza outbreaks in real time.

Thus, the objective of this case study is to demonstrate that CEP is an effective solution for detecting epidemics and pandemics in real time compared to most existent tools that report these situations weekly: FluNet [11] presents influenza information in all countries of the world and Euroflu [12] presents it only in member states of the WHO European region. The use of a CEP engine will allow health officials to mitigate as soon as possible the impact of epidemics and global pandemics, rather than being exposed to have a week delay on receiving the up-to-date information.

In the following subsections we describe the case study, define the complex event patterns necessary to detect critical situations in this scenario, enumerate the steps followed to implement the case study and finally present the results obtained after testing it.

 ISBN: 978-1-61208-152-6

A. Description of the Case Study

As previously mentioned this case study focuses on early detection of avian influenza outbreaks using CEP in SOA environments. In particular, this health system will launch real-time alerts when some of the following avian influenza cases are detected: (1) suspected cases of patients who may be infected by this virus, (2) confirmed cases of suspected patients, (3) epidemic cases (countries suffering outbreaks of avian influenza) and (4) pandemic case (the epidemic affects several countries). See WHO documentation [13] for further information about the definition of these cases.

The event producers are:

- **Hospitals**: health workers, particularly physicians, will diagnose symptoms and will get relevant information about patients. They will issue events inserting patient diagnoses in hospital information systems.
- **Laboratories**: laboratories will be able to detect confirmed cases of avian influenza by blood tests and other techniques, and they will publish this information as events within their information systems.

On the other hand, the event consumers are:

- **WHO and other international organizations**: these organizations will subscribe to relevant events about outbreaks of avian influenza and will be aware of suspected, confirmed, epidemic and pandemic cases that might have been detected worldwide.
- **Hospitals**: health workers will need to know which cases have been identified in order to take measures for relieving the situation, such as patient isolation measures.
- **Laboratories**: they will be continuously informed of the virus evolution and spread, facing the development of new antidotes or drugs to help authorities to fight the disease.

For example, hospital services may trigger a complex event if a suspected case of avian influenza is detected. This event will be received by the pharmacy and WHO services, which will react immediately to this situation: pharmacies automatically notify to suppliers an increased demand for those drugs that help fighting the disease and WHO will launch warning alarms to those laboratories and international health agencies that are interested in this situation.

B. Complex Event Patterns for Detecting Avian Influenza Outbreaks

In the following lines, we describe the definition of complex event patterns for detecting suspected, confirmed, epidemic and pandemic cases of avian influenza. To this end, we have adapted Buschmann's design patterns scheme [14]: pattern *name* and a short summary, real-world *example* demonstrating the existence of the problem and the need for the pattern, *context* (situations) in which the pattern may be applied), *problem* addressed by the pattern, *solution* proposed by the pattern, detailed specification of the pattern *structural aspects*, pattern *implementation* in a specific language and *consequences* (benefits and drawbacks) provided by the pattern. In this work we will describe the pattern name and implementation, which are the main relevant parts of the schema for the case study illustration.

According to real requirements for detecting avian influenza cases we defined the next complex event patterns:

- **Suspected case**: this pattern detects possible occurrences of avian influenza cases, when the following conditions are met:
  1) The patient has fever (above 38 °C) or cough or headache or myalgia or conjunctivitis or pharyngitis or encephalopathy or multiple organ failure or pneumonia.
  2) And, moreover, he/she presents a history of exposure to known infection sources in infectious period (7 days prior):
     - Staying in an area where avian influenza human cases have been reported.
     - Having contact with a person already diagnosed of avian influenza.
     - Having contact with animals that could be infected.
     - Handling gases in a laboratory.
- **Confirmed case**: the laboratory confirms an avian influenza infection, based on the detection of a suspected case and a biological sample of the patient.
- **Epidemic case**: there are 25 or more confirmed cases of avian influenza in a particular country during a week.
- **Pandemic case**: there are 2 or more epidemic cases during a week.

C. Implementation

The presented case study has been implemented using Java and the Esper engine. Moreover, the complex event patterns defined above have been implemented in the complex event processing language of Esper [15], EPL (Event Processing Language). Several reasons have motivated EPL choice: firstly, the learning curve is not high because its syntax is very close to SQL, widely known worldwide. Besides, EPL natively supports multiple event format types: Java/.NET objects, maps and XML documents what facilitates its use in multiple platforms. Even more, it is also possible to customize not only the language but also Esper engine, which is written in Java and is open source.

The steps followed to implement the case study are enumerated and described below:

1) **Configuration and initialization of the Esper engine**. An instance of *com.espertech.esper.client.Configuration* represents all configuration parameters. The *Configuration* is used to build an *EPServiceProvider*, which provides the administrative and runtime interfaces for an Esper

 ISBN: 978-1-61208-152-6

engine instance. A *Configuration* instance is then obtained by instantiating it directly and adding or setting values on it. The *Configuration* instance is then passed to *EPServiceProviderManager* to obtain a configured engine, as the following code shows:

```
Configuration conf = new Configuration();
config.addEventType("PatientState",
  PatientState.class.getName());
config.addImport("es.uca.esper");
EPServiceProvider epService
  = EPServiceProviderManager.getProvider(
  "sample", conf);
```

2) **Creation of an event generator to simulate patients treated worldwide and their health state evolution**; the simulator will be directly connected to the CEP engine. We will make use of this simulator to produce patient state events randomly rather than using real information from hospital and laboratory systems (due to the access restrictions to official information in the above mentioned systems). In this simulator there are two types of objects:
   - *Patient*: each patient in the simulation has a specified person id, date of birth, sex and country.
   - *PatientState*: this object represents patient health state evolution, which has the following attributes: identification, registration time, current location of the person, symptoms and dates in which the patient has been exposed to infection sources.

3) **Introduction of the generated events in Esper event streams**. The *PatientState* instance insertion in Esper is implemented as follows:

```
epService.getEPRuntime().sendEvent(
  PatientState);
```

4) **Implementation and registration of complex event patterns in Esper**. The suspected case of avian influenza implemented using EPL is presented below:

```
String suspectedCase =
"insert into AvianInfluenzaSuspects
select avianInfluenzaSuspect.id,
  avianInfluenzaSuspect.registrationTime,
  avianInfluenzaSuspect.patient.sex,
  avianInfluenzaSuspect.currentLocation
from pattern [every avianInfluenzaSuspect
  = PatientState((cough or fever > 38
  or headache or multipleOrganFailure
  or myalgia or pharyngitis or pneumonia
  or conjunctivitis or encephalopathy)
  and ((PatientState.DayCounter(
  registrationTime, infectionArea)<=7)
  or (PatientState.DayCounter(
  registrationTime, infectionPerson)<=7)
  or (PatientState.DayCounter(
  registrationTime, infectionAnimal)<=7)
  or (PatientState.DayCounter(
  registrationTime, laboratoryGases)<=7)
)]";
EPStatement suspectedCaseStatement =
  epService.getEPAdministrator().
  createEPL(suspectedCase);
suspectedCaseStatement.addListener(
  new AvianInfluenzaSuspectListener());
```

Concerning the code, the complex event pattern illustrating suspected case implementation is defined by the *from pattern* clause. *PatientState* events meeting described conditions for suspect case are selected from the Esper event stream. For this purpose, *every* operator is applied to obtain all these events and the *avianInfluenzaSuspect* alias is assigned to them.
DayCounter is a function to count days passed from the date on which *PatientState* event was registered to the date on which the patient was in contact with a risk source, if there were any contact.
Afterwards, identification, registration time, sex and current location attributes of the met *avianInfluenzaSuspect* complex events are selected and inserted in a new event stream called *AvianInfluenzaSuspects*, by using an *insert into* clause.
A specific listener, known as *AvianInfluenzaSuspectListener*, will receive suspect patient event notifications and will alert those interested to these situations. These warning alarms could be used to infer statistical data, e.g., the amount of suspected case grouped by sex in a specific time for a given country.

5) **Detection of complex events according to the registered patterns and notification of these events to the listeners**. The implementation of *AvianInfluenzaSuspectListener*, which receives events detected by *SuspectedCase* pattern, is shown below:

```
public class AvianInfluenzaSuspectListener
  implements UpdateListener {
    @Override
    public void update(EventBean[]
      newEvents, EventBean[] oldEvents) {
        String currentLocation =
          (String) newEvents[0].
            get("currentLocation");
          System.out.println(
           "\n***SUSPECTED CASE IN: " +
           currentLocation + "***\n"); }}
```

6) **Definition and implementation of test cases using the JUnit framework and validation of the application**. For example, the *testGen* test case is presented below, which checks if our implemented event generator creates the specific amount of events and insert them in an event stream:

```
public void testGen() throws Exception {
    final int EVENT_N = 100000;
    PacientStateGenerator generator =
      new PacientStateGenerator();
    LinkedList stream = generator.
      makeEventStream(EVENT_N);
    assertEquals("The amount of events
      generated randomly should be " +
      EVENT_N, stream.size(), EVENT_N); }
```

 ISBN: 978-1-61208-152-6

D. Testing and Results

In this study, 100.000 - 600.000 patient states, from 119 countries, have been generate randomly, using the implemented event generator.

In our simulations we have observed that up to 300.000 generated patient states no epidemic case has been alerted. The reason is there are not enough patient states to detect suspected cases, which are also confirmed, requiring at least 25 confirmed cases for the same country during a week.

Only 2 epidemic cases have been detected of 350.000 patient states. However, the amount of epidemic cases significantly increased from 400.000 states. We can deduce that the more patient states are generated the more epidemic cases will be detected.

As a conclusion, we can assert that using CEP permits an inmediate and efficient detection of complex patterns in large amounts of flowing information.

V. Discussion

Through the case study implementation and evaluation we have seen that our proposal provides an efficient solution for early detection of avian influenza outbreaks. Besides we can stress the following additional advantageous characteristics:

The use of a CEP engine instead of a rule engine provides substantial benefits, as discussed in this section. According to Chandy and Schulte [8] there are some differences between CEP and rule engines: normally, rule engines are request-driven, i.e., when an application needs to make a decision it will invoke this engine to derive a conclusion from a set of premises. The general model for a rule engine is *If "some condition" then "do action X"*. In most applications, a large number of rules will have to be analyzed before making a decision, thus becoming a problem for real-time decision making. However, CEP engines are event-driven and run continuously, and according EDA principles, they can process notification messages as soon as they arrive. In this case, the general model for a CEP engine is a when-then rule (known as a *complex event pattern*) *When "something happens or some condition is detected" then "do action X"*, instead of an if-then-else rule. The equivalence of the if-else clause for a CEP engine is the one that specifies *When "something has not happened in a specific time frame" then "do action Y"*. Thus, event patterns use time as another dimension. Moreover, CEP engines are faster and more efficient in handling and receiving notifications since they can directly manage inputs and outputs with messaging systems, while rule engines behave as services used by input and output systems.

Another improvement is our approach provides event prioritization according to the order previously set for every event type. Event prioritization will prevent the bottleneck effect in the CEP engine, as it will serve firstly higher priority events, thus avoiding the management of a huge number of events at one particular point in time.

Finally, our proposal uses NoSQL databases. They provide the following adavantages [16]:

- NoSQL have asynchronous BASE (Basically Available, Soft state, Eventual Consistency) updates rather than synchronous ACID (Atomicity, Consistency, Isolation, Durability).
- NoSQL databases are optimized to react to changes, not to manage transactions, and they do not require neither schemes nor data types definitions.
- They are also distributed, easily horizontal scalable and very efficient for managing huge data amounts.

VI. Related Work

Several works about CEP and SOA integration in different domains can be found in the literature; in the following paragraphs we summarize the most representative ones.

To start with, He et al. [4] implement an event-driven system based on radio-frequency identification to monitor gases emitted by vehicles and detect if vehicle's emissions are not standard, keeping those interested in protecting the environment and quality air informed about this situation. The authors claim that all events in the CEP engine are represented by POJO (Plain Old Java Object); the language used to process events is similar to SQL, however it is not specified whether this language has been extended to manipulate time windows, which is a relevant feature for a CEP application. Besides, events are not represented in XML (what would allow obtaining more readable and reusable events) as the Esper engine [15] does. On the other hand, they do not specify whether the motor and the language can be customized and extended as Esper allows to.

On the other hand, Taher et al. [5] propose to adapt interactions of Web service messages between incompatible interfaces. In this regard, they develop an architecture that integrates a CEP engine and input/output adapters for SOAP messages. Input adapters receive messages sent by Web services, transform them to the appropriate representation to be manipulated by the CEP engine and send them to the latter. Similarly, output adapters receive events from the engine, transform them to SOAP messages and then they are sent to Web services. This architecture has some limitations regarding our proposal: firstly, services interact with a framework that integrates both message adapters and the CEP engine, instead of using an ESB directly connected to the services and the engine. The bus would provide a decoupled, flexible and reusable system. Secondly, the proposed adapters do not provide additional functionalities which an ESB usually provides, such as message routing based on content and transformation protocols.

Sottara et al. [6] propose an architecture for the management of waste water treatment plants, in which an ESB is used to connect services distributed in different nodes in a transparent way for end users. They integrate JBossESB solution [17] and Drools rule engine [18], which is embedded

 ISBN: 978-1-61208-152-6

in the bus. Our proposal improves this one by using a CEP engine instead of a rule-based one.

Finally, there are two projects funded under the EU 7th Research Framework Programme that integrate CEP and SOA: MASTER and COMPAS. MASTER [19] provides an infrastructure that facilitates monitoring, enforcement, and auditing of security compliance and COMPAS [20] designs and implements an architectural framework to ensure dynamic and on-going compliance of software services to business regulations and stated user service-requirements. However, while our solution allows to store event logs in NoSQL databases, these projects do not consider it.

VII. Conclusion and Future Work

We have proposed and discussed an approach for the efficient use of CEP in SOA 2.0 scope. Thanks to this approach, when relevant situations arise from the detection of certain predefined event patterns, real-time alerts will be sent to the interested parties. In this paper, we have mainly focused on the use of a CEP engine to detect event patterns.

A case study illustrating this approach has also been described and implemented. Our system can detect epidemics and pandemics in real time, while most current tools report these situations weekly. So, we can conclude that CEP technology is suitable for this purpose. Although we have taken the example of the avian influenza virus; once new complex event patterns are defined, our system could be used for the prevention of other diseases as well as for non-medical fields, should it be necessary.

In our near future work we will approach a complete architecture for the integration of SOA 2.0 and CEP making use of an ESB. In this regard, the CEP engine will be able to process *real* events that will be published into the ESB by different event producers, replacing the random event generator developed in this work to simulate patient states. Moreover, both event producers and consumers will be Web services, providing a loosely coupled more complex, modular and flexible system.

Acknowledgements

We would like to specially thank Novayre [21] managers for their fruitful comments and discussions on the topic dealt with in this paper. The second author acknowledges the support from TIN2008-02985 and FEDER.

References

[1] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. MA, USA: Addison-Wesley, 2002.

[2] H. Taylor, A. Yochem, Les Phillips, and F. Martinez, *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*. Indiana, USA: Addison-Wesley, Mar. 2009.

[3] B. Sosinsky, *Cloud Computing Bible*. Indiana, USA: Wiley, Jan. 2011.

[4] M. He, Z. Zheng, G. Xue, and X. Du, "Event Driven RFID Based Exhaust Gas Detection Services Oriented System Research," in *Proc. 4th International Conference on Wireless Communications, Networking and Mobile Computing*, Dalian, China, Oct. 2008, pp. 1–4.

[5] Y. Taher, M. Fauvet, M. Dumas, and D. Benslimane, "Using CEP Technology to Adapt Messages Exchanged by Web Services," in *Proc. 17th International Conference on World Wide Web*, Beijing, China, Apr. 2008, pp. 1231–1232.

[6] D. Sottara, A. Manservisi, P. Mello, G. Colombini, and L. Luccarini, "A CEP-based SOA for the Management of WasteWater Treatment Plants," in *Proc. IEEE Workshop on Environmental, Energy, and Structural Monitoring Systems*, Crema, Italy, Sep. 2009, pp. 58–65.

[7] "NoSQL databases," Mar. 2011. [Online]. Available: http://nosql-database.org/

[8] K. M. Chandy and W. R. Schulte, *Event Processing: Designing IT Systems for Agile Companies*. USA: McGraw-Hill, 2010.

[9] M. Havey, "CEP and SOA: Six Letters Are Better than Three," Feb. 2011. [Online]. Available: http://www.packtpub.com/article/

[10] "United Nations," Mar. 2011. [Online]. Available: http://www.un.org/en/

[11] "FluNet," Apr. 2011. [Online]. Available: http://www.who.int/csr/disease/influenza/influenzanetwork/flunet/en/

[12] "EuroFlu," Apr. 2011. [Online]. Available: http://www.euroflu.org/index.php

[13] "World Health Organization," Mar. 2011. [Online]. Available: http://www.who.int/en/index.html

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester, UK: Wiley, 1996.

[15] "Esper," Mar. 2011. [Online]. Available: http://esper.codehaus.org

[16] E. Meijer and G. Bierman, "A co-Relational Model of Data for Large Shared Data Banks," *ACM Queue*, vol. 9, pp. 30–48, Mar. 2011.

[17] "JBoss ESB," Jan. 2011. [Online]. Available: http://jboss.org/jbossesb

[18] "Drools," Jan. 2011. [Online]. Available: http://www.jboss.org/drools

[19] "MASTER," May 2011. [Online]. Available: http://www.master-fp7.eu/

[20] "COMPAS," May 2011. [Online]. Available: http://www.compas-ict.eu/

[21] "Novayre," May 2011. [Online]. Available: http://www.novayre.com/

 ISBN: 978-1-61208-152-6