SMART REIFIER: Model-Driven Development of Service-Oriented SCADA Applications from Models of Sensor and Actuator Networks

Margaux Bosshardt, Clémentine Geslin and Jérôme Rocheteau
Institut Catholique d'Arts et Métiers, 35 avenue du Champ de Manœuvres, Carquefou, France

Keywords: Model-Driven Development, Domain-Specific Language, Service-Oriented Application, Supervisory Control and Data Acquisition, Sensor and Actuator Network.

Abstract: This paper aims at presenting SMART REIFIER a tool for designing networks of sensors and actuators and for generating a set of web services for supervisory control and data acquisition. Such a code generation is achieved by the means of model-driven engineering: a specific meta-model for sensor and actuator networks is designed as well as a model-to-model mapping into a web service meta-model that enables source code generation of JEE applications.

1 INTRODUCTION

The FUSE-IT project aims to design and implement smart and secured building management systems for safety-critical sites. *Smart* as electricity can be supplied either from inside the building thanks to different types of generators or from the outside by the grid. *Secured* as management systems should not be misused in safety-critical contexts. During this project, ICAM designs and implements a smart lighting demonstrator on its own site, a university of engineering, with solar panels, batteries, light-emitting diodes, presence and luminosity sensors, remote-controlled switches and dimmers. Hence, such a smart lighting system requires a management system that fulfills the following features:

- operational for data acquisition and supervisory control (SCADA),
- compatible with other SCADA applications e.g. from project partners,
- available for third-party applications for visualization, regulation, optimization, etc.

Moreover, this smart lighting implementation encountered some delays for selecting available components according to their cost and features, for fixing areas where they should be located, for stating the accurate number of each type of component, for designing a safe and independent electric network, for determining communication networks and technologies, etc.

This leads us to adopt a model-driven approach with distinct models from their implementations and automatic code generation. In fact, model-driven engineering of such a sensor and actuator network (SAN) makes it possible to update the smart lighting design as often as it is mandatory with the less modifications possible regardless of its implementation. The specific component implementations of generated SCADA applications only corresponds to communication modules of each type of sensors and actuators that compose such a network. Such a model-driven engineering then allows us to focus more on SAN design than on SCADA development. The most important added-value consists in enabling SCADA application development according to a top-down approach alongside a co-design of its sensor and actuator network instead of a bottom-up approach that is posterior its network and actuator network design which is mostly and currently the case; with a better support for design checking, component reuse, etc.

This paper is organized as follows: Section 2 presents the smart lighting demonstrator design at ICAM. Section 3 defines a domain-specific language for sensor and actuator networks and formalizes the previous study case. Such models will be embedded into web services which meta-model is presented by section 4 thanks to a model-to-model transformation detailed in section 5. This meta-model of web services makes possible to generate SCADA application source code as section 6 explains. Related work is then investigated in section 7 in order to prove the novelty and the relevance of our approach.

Rocheteau J., Geslin C. and Bosshardt M.
SMART REIFIER: Model-Driven Development of Service-Oriented SCADA Applications from Models of Sensor and Actuator Networks.
DOI: 10.5220/0006198201250136
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 125-136
ISBN: 978-989-758-210-3

2 SMART LIGHTING DEMONSTRATOR

This section aims at introducing specifications of the smart lighting demonstrator at ICAM: a network composed of sensors and actuators set up to regulate the light level in offices and to monitor its energy consumption. It requested a long time to implement this design and, moreover, the latter could change according to different parameters such as component costs, component availability, etc. This design variability has motivated the model-driven approach described in section 3. The entire system, including power sources, will not be described. However, a specific focus will be made on the lighting system as a running example. This system should comply to the European norm EN 12464-1 that requires a minimum light level for working zones and their surrounding area. 500 LUX are needed in working areas, but in other areas of the room a minimum of 300 LUX need to be maintain.

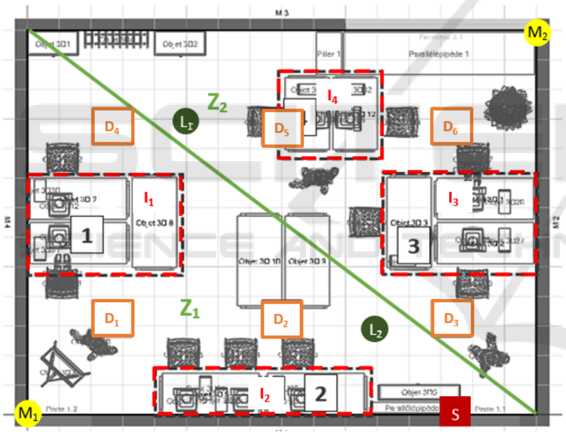

Figure 1: Smart Lighting Supervisory Demonstrator.

As stated above, ICAM designs and implements a smart lighting demonstrator in its own building based in Nantes. Solar panels and batteries are used to light up a 22.84 square meters room, composed of 4 working islets. Thus, the demonstrator is composed of three main parts which are (a) the photovoltaic panels located on the roof, (b) an energy part equipped with electric storage batteries, electric inverters, powermeters, computers hosting SCADA applications (c) and the lighting system itself located in the IT laboratory. This lighting system is illustrated by figure 1. It is composed of these physical or logical items:

- 2 zones Z_1 and Z_2 composing an upper zone.
- 2 motion sensors M_1 and M_2 associated to their corresponding zone which report at a fixed rate whether somebody is detected.
- 2 light sensors L_1 and L_2 that provide the luminosity value when they are requested to.
- 6 light-emitting diodes (LED), each one equipped with a remote-controlled dimmer D_1, D_2, D_3, D_4, D_5 and D_6 that settle the light level of their corresponding LED.
- 4 islets or working zones I_1, I_2, I_3 and I_4 that are dispatched between the 2 zones Z_1 and Z_2.
- 1 switched S that allows users to to switch on or to switch off lights.

Light level of each islet can automatically be computed according to the luminosity values of the light sensors L_1 and L_2.

In addition, the lighting system should perform as follows: It has to adjust to every islet light levels according to the luminosity values retrieved from the light sensors when somebody is detected in one of the two zones or the switch S is on. It has to switch off LED when nobody is detected in any zone and the switch S is off. It has to toggle the switch S if nobody is detected in any zone after a certain time.

3 SENSOR AND ACTUATOR NETWORK METAMODEL

These previous specifications of the smart lighting demonstrator leads us to identify required features of the SAN meta-model. In fact, such a meta-model should make it possible to:

Locate Sensors and Actuators in a Zone. For example, the motion sensor M_1 belonging to the zone Z_1.

Locate a Zone Inside another Zone. The zone Z_1 belonging to the zone that corresponds to the entire office.

Distinguish Models and Devices. In fact, the two motion sensor devices will probably be the same model but located in different zones.

Define Computations. Simple ones like “receive the motion sensor M_1 message” or more complex ones like “adjust light levels”.

Define Parametric Computations. In fact, the two computations “receive the motion sensor M_1 message” and “receive the motion sensor M_2 message” corresponding to a single computation “receive the motion sensor X message” applied to different parameters M_1 and M_2.

Trigger Computations on Events. For instance, it corresponds to adjust the light levels when somebody is detected in a zone.

```
   network   =   (name:name,instruments:instrument*,places:place*,processes:process*)

instrument   =   (name:name,mode:mode,component:component,attribute:variable*)
      mode  ::=  sensor | actuator

     place   =   (name:name,place:place?,instances:instance*)
  instance   =   (name:name,instrument:instrument,parameters:parameter*)

   process   =   (component:component,triggers:trigger+)
   trigger  ::=  event | loop | task | user
     event   =   (instance:instance)
      loop   =   (delay:long)
      task   =   (resource:resource)
      user   =   (path:name)
```

Figure 2: Meta-Model of Sensor and Actuator Networks.

Trigger Computations at Fixed-rate. For instance, it corresponds to toggle the switch *S* if nobody is detected after a while.

These SAN requirements leads us to design the figure 2 meta-model by the means of a formal grammar. The main syntactic category network enables to specify SAN models as a list of instruments (sensor and actuator models), places (zones) and processes (triggered computations). Syntactic categories: name, resource, component, variable and parameter are drawn out from figure 3 meta-model of component-based web services and will fully be explained in section 4. Components are parametric ones: they can be defined either as abstract components that are implemented by a Java class or as compound components that are composed of concrete components. Concrete components correspond to components specialized by some parameter values.

Instruments are defined by the means of the so-called category instrument. They are defined by their mode i.e. sensor or actuator. They are also defined by their component which corresponds to the communicating unit implementation that retrieves or receives data from this instrument. Moreover, instrument data structure is defined by a list of attributes; the latter made of a name and a data type.

Areas or zones are defined by the means of the category place. They can be linked to an upper place by the feature called *place* and they are defined by a list of instances. Instances correspond to sensor or actuator devices. In fact, instances are related to an instrument and can be applied to a list of parameters. The latter should exactly match the instrument component parameters.

Finally, processes are defined by the means of the category process which extends the category of component with a feature *trigger* at least. This feature refers to the so-called category trigger and is defined either (1) as an event on interactions with a specified instance or (2) as a loop at a fixed-rate specified by a feature delay or (3) as a background task specified by a REIFIER's resource that is launched at the SCADA application deployment or (4) as a specific service launched by SCADA application users given a *path*.

The meta-model of SAN is an extension of that of web services. As the model-to-model transformation maps SAN models to web service ones (see section 5), the SAN meta-model then consists of an extension of that of web services. It can be seen as syntactic sugar. However, it allows to focus on business modelling mainly and to hide technical implementations. It then provides a flexible abstraction layer with full support. Full support as a compliant SCADA application is generated from SAN models. Flexibility is ensure because this framework allows designers to focus on describing networks of sensors and actuators only and it allows developers to focus on customized computing units only. For instance, figure 1 shows the smart lighting demonstrator model at ICAM in XML format. This model is composed of three parts. The first part consists of listing the instruments i.e. the sensor or actuator models. Every instruments specify one attribute only which correspond to the data structure required or emitted by sensors or actuators. Every instruments specify one component as required which is related to the Java qualified name of the component that has to retrieve or provide data respectively from or to sensors and actuators. In addition, some instruments specify abstract parameters that will be useful in order to specialize their instances. The second part of the model consists of listing the places with their associated instances. i.e. the physical devices whose models are declared within the instrument part of the model. Some instances define values to parameters declared by their instruments. The third part of the model consists of listing the processes that have to be launched either while interacting with instances or at fixed-rate as specified by their inner trigger tag. Component attributes of these process tags corresponds to the Java qualified names of components formalized by an underlying model of web services as section 6 explains whose meta-model is described in section 4.

Listing 1: Smart Lighting Demonstrator Model.

```
<san:network name="fr.icam.fuseit">
  <san:instrument name="motion" mode="sensor" component="fr.icam.fuseit.drivers.MotionSensor">
    <san:attribute name="presence" type="boolean"/>
  </san:instrument>
  <san:instrument name="luminosity" mode="sensor" component="fr.icam.fuseit.drivers.LuxSensor">
    <san:parameter name="uri" type="uri"/>
    <san:attribute name="lux" type="integer"/>
  </san:instrument>
  <san:instrument name="dimmer" mode="actuator" component="fr.icam.fuseit.drivers.Dimmer">
    <san:parameter name="uri" type="uri"/>
    <san:attribute name="level" type="float"/>
  </san:instrument>
  <san:instrument name="switch" mode="sensor" component="fr.icam.fuseit.drivers.Switch">
    <san:attribute name="state" type="boolean"/>
  </san:instrument>
  <san:place name="c115">
    <san:instance name="dimmer-1" instrument="dimmer">
      <san:parameter name="uri" type="uri" value="http://172.21.220.12"/>
    </san:instance>
    <san:instance name="dimmer-2" instrument="dimmer">
      <san:parameter name="uri" type="uri" value="http://172.21.220.13"/>
    </san:instance>
    <san:instance name="dimmer-3" instrument="dimmer">
      <san:parameter name="uri" type="uri" value="http://172.21.220.14"/>
    </san:instance>
    <san:instance name="dimmer-4" instrument="dimmer">
      <san:parameter name="uri" type="uri" value="http://172.21.220.15"/>
    </san:instance>
    <san:instance name="dimmer-5" instrument="dimmer">
      <san:parameter name="uri" type="uri" value="http://172.21.220.16"/>
    </san:instance>
    <san:instance name="dimmer-6" instrument="dimmer">
      <san:parameter name="uri" type="uri" value="http://172.21.220.17"/>
    </san:instance>
    <san:instance name="switch-1" instrument="switch"></san:instance>
  </san:place>
  <san:place name="c115a" place="c115">
    <san:instance name="motion-a" instrument="motion"></san:instance>
    <san:instance name="luminosity-a" instrument="luminosity">
      <san:parameter name="uri" type="uri" value="http://172.21.220.10"/>
    </san:instance>
  </san:place>
  <san:place name="c115b" place="c115">
    <san:instance name="motion-b" instrument="motion"></san:instance>
    <san:instance name="luminosity-b" instrument="luminosity">
      <san:parameter name="uri" type="uri" value="http://172.21.220.11"/>
    </san:instance>
  </san:place>
  <san:process component="fr.icam.fuseit.components.DimmerAdjust">
    <san:event instance="motion-a"/>
  </san:process>
  <san:process component="fr.icam.fuseit.components.DimmerAdjust">
    <san:event instance="motion-b"/>
  </san:process>
  <san:process component="fr.icam.fuseit.components.SwitchWatch">
    <san:loop delay="30"/>
  </san:process>
</san:network>
```

4 WEB SERVICE METAMODEL

The meta-model of component-based and service-oriented applications has been presented in (Rocheteau and Sferruzza, 2016). It is powered by a tool called REIFIER that generates JEE applications from compliant models of web services. Hence the name of this prototype built on top of REIFIER generates services-oriented SCADA applications: SMART REIFIER.

The complete meta-model of web services is defined by the grammar in figure 3 and by the claass diagram in figure 6. Only its relevant features with respect to the SAN meta-model are presented. A model of web service is defined by three main sub-models: data model, component model and service model. Data model encompasses type, aspect and entity categories. Types correspond to basic data types in Java, aspects are abstract data types that will be mapped to Java interfaces whereas entities correspond to data structures and will be mapped to Java classes. Component model corresponds to parametric components. And service model enumerates entry points of applications where services can be seen as specializations of components to data types. In fact, parametric components can be instantiated and applied to parameter values and those values can be data types like entities. Moreover, web service meta-model allows to specify application resources that can be available to components. These resources can be used to schedule background tasks at fixed rate and will be used for mapping the last kind of processes in the SAN meta-model.

5 MODEL TRANSFORMATION

As JEE applications can be automatically generated from models of web services thanks to the REIFIER tool, the code generation of SCADA applications from SAN models will then consists of a model-to-model transformation by the means of the SMART REIFIER tool. Generated SCADA applications will therefore be service-oriented ones.

The model transformation is twofold: On the one hand, it consists of a fixed model of entities, resources and components that are used to embed elements of the SAN meta-model such as instruments and places as well as resources required for data persistence management. On the other hand, it consists of a variable model of aspects, entities, instances, components and services drawn out from the content of SAN models.

Fixed Transformation. The fixed part of target models is composed of two entities. The first entity Place owns two properties that correspond to its name and its upper place. The second entity Instrument owns three properties: its name, its place and its type i.e. sensor or actuator. These two entities will be stored into a database and instances of such entities will be introduced from SAN models. Another entity TimeSpan is added into the target model with two properties started and stopped which types are timestamps. This entity represents a span of time that will be used as the data structure of request messages that aims at retrieving data of instrument instances. However, this entity TimeSpan will not be stored into a database. In addition, the fixed part of target models is composed of several abstract components which are provided by Java libraries. The first two components concern data acquisition by the SCADA application from instrument instances: the first one called TimeStamper sets the time-stamp of object data, the second one called HibernateCreator inserts object data into a database. The last six components concern data retrieving by the users from the SCADA application: Two components called JsonDeserializer and JsonSerializer in charge of, respectively, decoding or coding messages in JSON format are introduced into target models. A component called FeatureGrabber able to extract data features is introduced into target models in order to extract started and stopped values from TimeSpan objects. Three components provided by a REIFIER component library for Hibernate are also introduced into target models. The first one called HibernateCriteriaProvider prepares SQL statements from a given entity. The second one called HibernateTimeSpanFilter inserts SQL restrictions from the grabbed started and stopped features from TimeSpan objects. The third one called HibernateListRetriever executes SQL statements and provides data lists. Finally, the fixed part of target models is composed of several services that consists of SCADA application main functionalities:

- a GET method service retrieving the list of places,
- a POST method service retrieving the sub-places of a given place,
- a POST method service retrieving the instrument instances of a given place.

These services allows to navigate through out sensor and actuator networks in order to access their data obtained by the means of services

Variable Transformation. As for the variable part of target models, transformation is driven either by instrument, or by place, or by instance, or by process.

```
       model  =    (name:string, entities:entity*,instances:instance*, ...
                   ..., resources:resource*,components:component*,services:service*)

     service  ≻    component[name:string,path:name,method:method,request:message,response:message]
     message  =    (content-type:string,content-encoding:string,headers:string*,type:type)
      method  ::=  get | post | put | delete | head | options | trace | connect

   component  ::=  atomic | composite
      atomic  =    (name:string,inputs:variable*,outputs:variable*,parameters:variable*)
   composite  ≻    abstract[components:parametrized*,processing:processing]
  processing  ::=  sequence | failover
parametrized  =    (component:component,parameters:parameter*)

    resource  =    (name:string,parameters:parameter*)

   parameter  ≻    variable[term:term]
   attribute  ≻    variable[required:boolean,reference:parameter]
        term  ::=  variable | constant
    variable  =    (name:string,type:type)
    constant  =    (type:type,value:object)

      entity  ≻    type[name:string,stored:boolean,entity:entity?,aspects:aspect*,properties:property*]
      aspect  ≻    type[name:string,features:property*]
    property  =    (name:string,type:type,required:boolean)
        type  ::=  string | boolean | integer | float | date | class | aspect | entity
```

Figure 3: Meta-Model of Web Services.

Instrument transformation consist of introducing an aspect defined by the name of this instrument and which features are provided by the instrument attributes. An additional feature issued is added to this aspect in order to time-stamp any received data. This aspect corresponds to the common data structure among every instances of this instrument. Finally, it consists of introducing a component whose signature is composed of the specified instrument parameters and three other parameters: *type*, *input* and *output*. The first one *type* corresponds to the data type this component handles. The last two correspond to the name of, respectively, the component input and the component output. These inputs and outputs should verify the type provided by the value of the *type* parameter. Moreover, this type should also comply to the aspect generated from this instrument transformation.

Whereas, place transformation merely consist of introducing instances of the entity Place with the appropriate features, instance transformation consist of introducing:

- an instance of the entity Instrument with the appropriate properties i.e. its type, name and place;
- an entity that complies the instrument aspect i.e. with properties that correspond to the instrument aspect features which instances are stored into a dedicated table of a database;
- a compound component composed of the instrument component that provides instrument data, the TimeStamper component, the HibernateCreator component;
- a service that wraps the previous compound component;
- a service that receives a message that corresponds to a TimeSpan object data and that provides the list of instances from the dedicated database table of the instrument entity for the requested span of time.

Finally, process transformation consists either of introducing the process component at the end of the compound component transformed from the instance if this process is triggered on an event, or of introducing a resource that wraps the same compound component within a Java thread if this process is triggered at fixed-rate.

6 CODE GENERATION

Code generation of SCADA applications is ensured by the SMART REIFIER tool. The latter is a set of modules for the REIFIER tool which is a Maven plugin that generates JEE applications from models of web services. The SMART REIFIER code generation behaves as follows:

1. it compiles a web service model from a XML file `src/main/reifier/model.xml`;
2. it compiles a SAN model from a XML file `src/main/reifier/network.xml`;
3. it tranforms the previous SAN model into a web service model according to the rules explained in section 5 and merges with the first web service model;

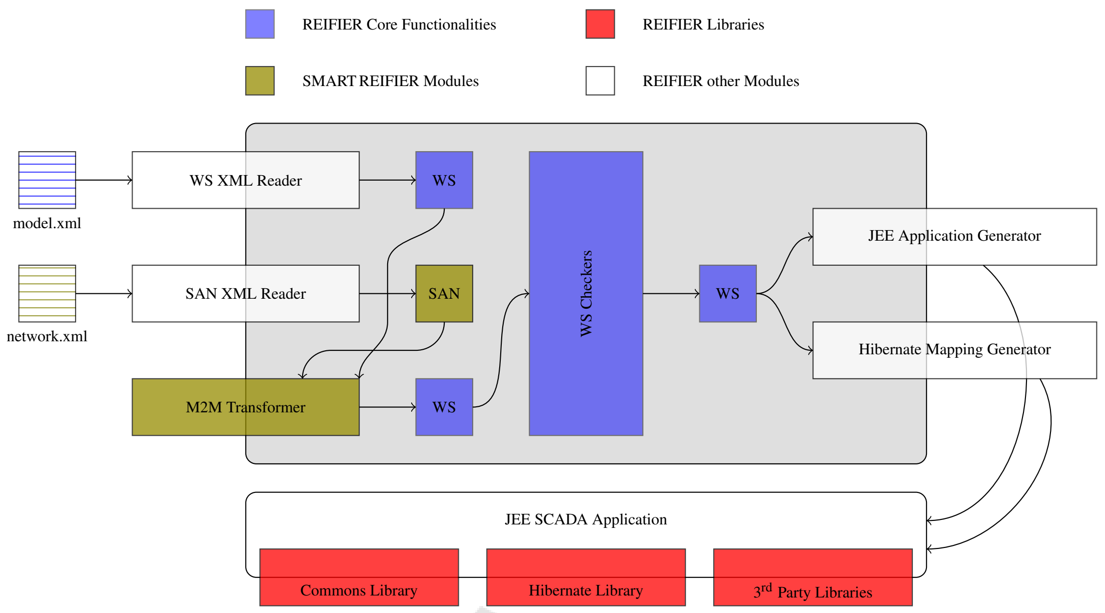

Figure 4: Architecture of the REIFIER tool and the SMART REIFIER modules.

4. it then applies REIFIER for model verification and code generation on the merged web service model in order to obtain the service-oriented SCADA application.

This code generation is illustrated by figure 4 where SMART REIFIER elements correspond to REIFIER modules. Firslty, figure 4 points out that the code generation relies on other modules that can be used in other contexts. Secondly, it points out that the generated SCADA application depends on Java libraries. Finally, it illustrates that the REIFIER tool merely verifies the consistency of web service models.

The code generation also provides a client-side application based on HTML, Bootstrap CSS and AngularJS technologies as figure 5 shows. This client-side application is unique as the code generation complies to the rules explained in section 5 and offers the three services of the fixed transformation that help to navigate through out sensor and actuator networks uniformly.

SCADA Application. The generated JEE application is a basic SCADA application. In fact, data acquisition is ensured as the generated JEE application is able to store sensor and actuator data into a database by the means of the transformed instance compound component and the transformed service that wraps the previous component. Moreover, it is able to provide these data by chunks of time intervals by the means of the other transformed service. Such a service enables supervisory control as SCADA applications users then can visualize data and, if needed, then launch some operations by the means of processes whose trigger is a user one. Such controls can be automatized by processes either triggered on instance communication or at fixed-rate. The code generation powered by the REIFIER tool also provides trendy SCADA application features such as service-oriented application, generic client-side application interfaces as figure 5, relational database persistence as well as NoSQL management for big data processing. Nevertheless, it doesn't allow to manage SQL and NoSQL databases alongside. However, the generated JEE application is not a complete SCADA application in the sense that it lacks of auxiliary functionalities. Users, roles and grants are not taken into account. Indeed, this approach is devoted to one sort of users: administrators of sensor and actuator networks. That should be required in order to adapt SAN views according to user roles.

SCADA Engineering. The code generation carries out most of the SCADA application components. It carries out application architecture and logic as well. This makes then possible to focus more on model design than application development. Development merely consists either in communication modules with sensors or actuators or in control operations for specific application behaviour. For example, listing 2 illustrates the Java implementation of a HTTP-based sensor communication component that retrieves data from a luminosity sensors. It's a pull communication component. In fact, this sensor component implementation rely on a underlying data type, here hidden, called LuxSensorMeasurement. It consists of an in-

Listing 2: The atomic component LuxSensor.

```
@Parameter(name="uri", type="uri") private URI uri;
@Parameter(name="type", type="type") private String type;
@Parameter(name="data", type="string") private String data;

public void doProcess(HttpServletRequest request, HttpServletResponse response) throws ServletException {
  Integer lux = this.getLux(this.uri);
  @Output(name="${data}", type="${type}")
  LuxSensorMeasurement measurement = new LuxSensorMeasurement();
  measurement.setLux(lux);
  request.setAttribute(this.data, measurement);
}

private Integer getLux(URI uri) throws Exception {
  CloseableHttpClient client = HttpClients.createDefault();
  try {
    HttpGet request = new HttpGet(uri);
    CloseableHttpResponse response = client.execute(request);
    try {
      InputStream input = response.getEntity().getContent();
      String string = IOUtils.toString(input, UTF8);
      return Integer.valueOf(string);
    } finally {
      response.close();
    }
  } finally {
    client.close();
  }
}
```

teger feature called lux that is directly drawn out from the sensor specification of the figure 1. Moreover, this data type is enriched by another feature called issued that corresponds to the time at which such data are retrieved. The latter will be provided by the TimeStamper component which is inserted just after such a communication component. This component implementation LuxSensor define two common parameters: a first type parameter called type – useful for the component formal verification from the REIFIER tool – and a second string parameter called data that corresponds to that attribute name of measurements Moreover, the LuxSensor component owns a third parameter called uri that is defined within its different instances in the model of the figure 1 and that stands for sensor instance IP addresses. This atomic component complies the Component interface and can be embedded into a composite component. In fact, the SCADA application code geneartion of ICAM smart lighting demonstrator provides a composite component called LuminosityAComponent that corresponds to the model instance luminosity-a of the place c115-a and that embeds the atomic component LuxSensor. Listing 3 illustrates how such atomic components are compound into a composite component.

Other types of communication can be handled: the pull and pooling communications. Pull communication component merely consists in retrieving data from sensors and in populating an instance of the data structure associated to the sensor. Pooling communication consists, on the one hand, in a push-like communication component and, on the other hand, in a resource that initializes the connection with the sensor. Such resources are launched as background tasks when SCADA applications are deployed.

The model-driven engineering of SCADA applications from SAN models then makes possible to focus more on design as it provides a strong support by the means of the JEE application code generation. Moreover, it provides a support for formal verification thanks to the REIFIER tool. However, integration of heterogeneous components still remains flexible as different communication protocols can be managed.

7 RELATED WORK

Applications of model-driven engineering to SCADA applications or sensor and actuators networks are twofold: Firstly, it has been used for SAN modelling. Secondly, model-driven engineering has been used for testing SCADA applications. These two domains are investigated before the relevance of our approach is discussed.

Listing 3: The composite component LuxSensor.

```
@Parameter(name="uri", type="uri") private URI uri;
@Parameter(name="type", type="type") private String type;
@Parameter(name="data", type="string") private String data;

private LuxSensor aLuxSensor1;
private TimeStamper aTimeStamper2;
private HibernateCreator aHibernateCreator3;

public void setUp(ServletContext context, ServletConfig config) throws ServletException {
  aLuxSensor1 = new LuxSensor();
  aLuxSensor1.setUri(this.uri);
  aLuxSensor1.setType(this.type);
  aLuxSensor1.setData(this.data);
  aTimeStamper2 = new TimeStamper();
  aTimeStamper2.setType(this.type);
  aTimeStamper2.setName("issued");
  aHibernateCreator3 = new HibernateCreator();
  aHibernateCreator3.setHibernate("hibernate");
  aHibernateCreator3.setType(this.type);
  aLuxSensor1.setUp(context, config);
  aTimeStamper2.setUp(context, config);
  aHibernateCreator2.setUp(context, config);
}

public void doProcess(HttpServletRequest request, HttpServletResponse response) throws ServletException {
  aLuxSensor1.doProcess(request, response);
  aTimeStamper2.doProcess(request, response);
  aHibernateCreator3.doProcess(request, response);
}

public void tearDown() {
  aLuxSensor1.tearDown();
  aTimeStamper2.tearDown();
  aHibernateCreator3.tearDown();
}
```

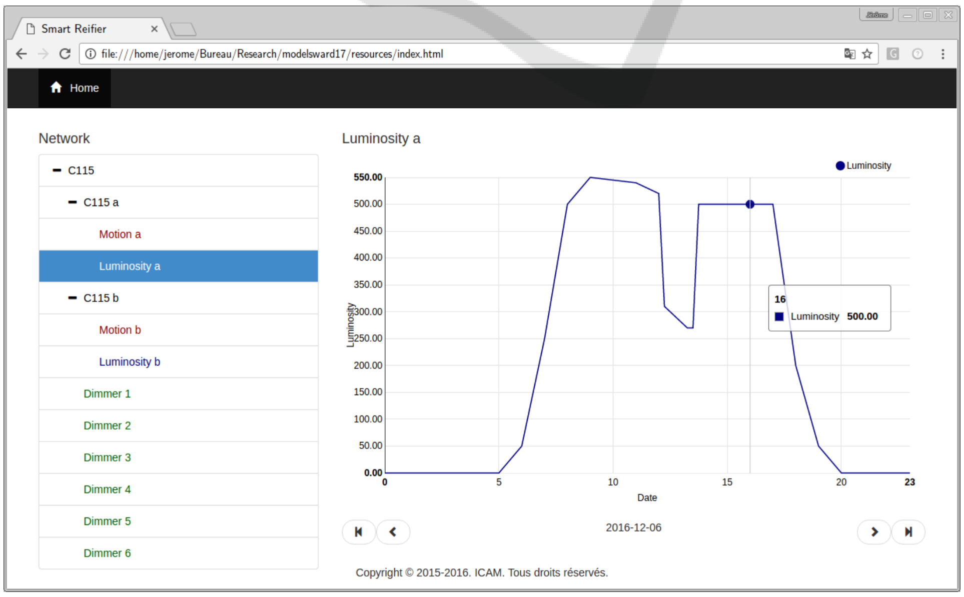

Figure 5: Screenshot of the Smart Lighting Demonstrator SCADA Application.

Design. The application of model-driven engineering to sensor and actuator networks has already been investigated and this approach share several features with those found in the scientific literature (Rodrigues et al., 2011; Priego et al., 2016). In fact, most of the experiments about SAN modelling present different solutions for assisting their design and development as noticed in (Rowe et al., 2010; Kowal et al., 2014). In (Vidal et al., 2015), MindCPS solution is a solution that aims at facilitating SAN development. To do so, the idea is to provide modelling primitives for explicitly specifying the autonomic behaviour of the system and to model transformations for automatically generating part of the code. The main advantages of this automated code generation is to allow a rapid configuration and development, even for a newcomer user. Adopting a model-driven development approach facilitate the developing of SAN applications and promote a clear and synergetic separation between the specification of the requirements at the application level and such specification in a given sensor platform (Rodrigues et al., 2013). Models — statistical or otherwise -– are used for describing, simplifying or abstracting various components of sensor data acquisition and management (Sathe et al., 2013). The best way to ease developer job seems to model separately the software architecture of sensor and actuator networks, the low-level hardware specification of the SAN nodes and the physical environment where nodes are deployed in (Doddapaneni et al., 2012). This separation of concerns is needed since hardware and software aspects are locked and tied down to specific types of nodes, hampering the possibility of reuse across projects and organizations.

Several articles tackle the use of domain-specific languages for the development of sensor and actuator networks. LWiSSy (Dantas et al., 2013) has been promoted as a domain language that considers three levels of programming granularity which may have distinct characteristics regarding the used platform or data processing. The idea is to divide the responsibility in dividing the expert skills by requiring from them only their specific knowledge. Thus, one large model is difficult to visualize, maintain, and analyze taking into account evolution. Different views onto the feature model or the combination of two or more features models, one for each domain, may solve some of the existing problems.

Testing. The step of verification and validation of distributed systems is another considerable challenge (Yang et al., 2014). In (Süß et al., 2008) approach to the problem of testing a SCADA thanks to Modelica, an object-oriented mathematical modelling language for component-oriented modelling of complex physical systems. It is an open standard and implementation, and provides a rendering of its input language in Ecore, the metalanguage of the Eclipse Modeling Framework (EMF). This tooling allows a test engineer to model all aspects of a SCADA test within one workbench and enjoy full traceability between the proprietary test model, and its surrounding environment simulation.

ITEA 2 projects IMPONET and NEMO & CODED, focused on supporting complex and advanced requirements of smart grids, specifically supporting enhanced efficiency through sensing and metering technologies, as well as automated control and management techniques based on energy availability and the optimization of power demand (Vidal et al., 2015).

Discussion. The starting point of this work was to provide a scalable modelling approach covering variability and evolution of the smart lighting system at ICAM. Indeed, the model-driven development allows us to apply this approach to every other networks of sensors and actuators that could be modelled according to the SAN meta-model of figure 2. However, it has not been investigated to which extend this approach can be applied to: internet of things, edge computing, etc. In addition, several works have pointed out the close relationship between application requirements and SAN performance, and demonstrated that application-specific optimization can increase overall system performance, mainly regarding the energy consumption. (Yang et al., 2014) points out an inherent intertwining between modelling in the control sense and model-driven software engineering.

The major drawback of this approach lies in resources or components that are embedded in processes of sensor and actuator models. In fact, they mainly refer to compound components as the DimmerAdjust and SwitchWatch ones from the smart lighting demonstrator model in the figure 1. The latter should be defined in underlying models of web services (i.e. in the file `model.xml`) aside of those of sensor and actuator networks (i.e. in the file `network.xml`). This forces SCADA developers to define the most complex system elements within technical models of web services instead of business ones of sensor and actuator networks whereas such components should belong to the latter.

Moreover, some common patterns of SCADA applications are encoded For instance, a pooling communication sensor or actuator implementation, such as serial port communication, is implemented twofold. The first part, corresponds to a push communication component. The second part corresponds

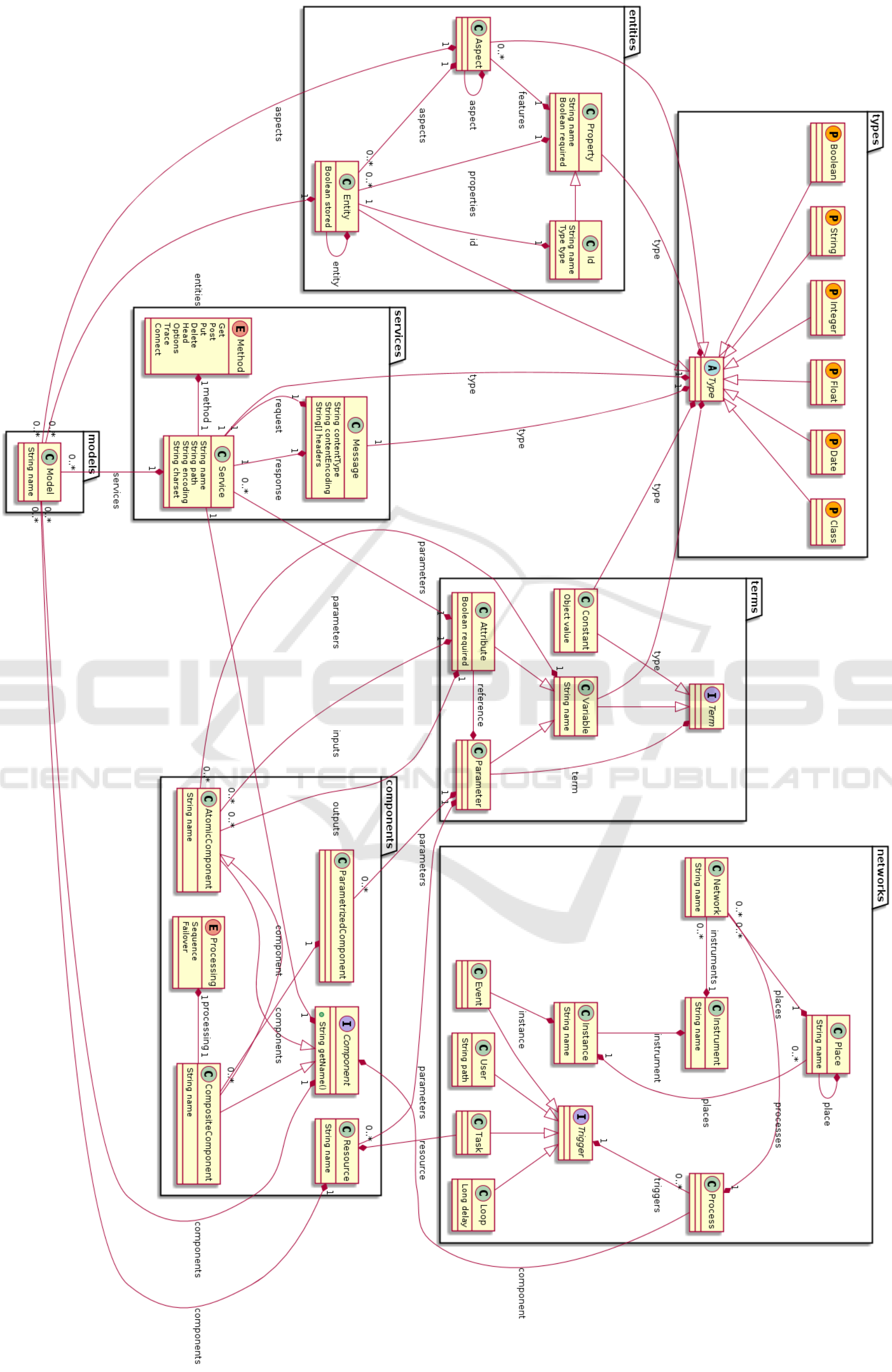

Figure 6: Meta-Models of Web Services and Sensor & Actuator Networks.

to a task process whose resource launches a thread for reading on the serial port and that processes the specified component that each time a data is read. Indeed, some components whose implementations have to be provided by developers should refer to generated components. This is a shortcoming of this approach.

8 CONCLUSION

The current work presents a model-driven development for automatically generating source code of service-oriented JEE applications that corresponds to SCADA applications from a model of sensor and actuator networks. This approach consists of a model-to-model transformation from a domain-specific metamodel of sensor and actuator networks to a metamodel of component-based web services; the code generation is delegated to the REIFIER tool that provide JEE application source code from web service models. It makes possible to develop customized SCADA applications for given sensor and actuator networks and to reverse SCADA application engineering. In fact, the traditional approach consists in customizing an existing SCADA application.

Prospects are threefold. Firstly, it aims at ensuring smart network security and modelling roles and grants of users in order to generate industrial-like SCADA applications and fine-grained user management. Secondly, it aims at designing a top-down engineering of sensor and actuator networks by incremental refinement steps from sensor and actuator specifications to implementations. Thirdly, it aims at integrating common patterns of data analysis directly as artifacts of the model transformation instead of embedding into SCADA applications them as user triggered processes of sensor and actuator network models.

ACKNOWLEDGMENTS

The research leading to these results was partially funded by the ITEA 2 project 13023 FUSE-IT.

REFERENCES

Dantas, P., Rodrigues, T., Batista, T., Delicato, F. C., Pires, P. F., Li, W., and Zomaya, A. Y. (2013). Lwissy: A domain specific language to model wireless sensor and actuators network systems. In *4th International Workshop on Software Engineering for Sensor Network Applications*, pages 7–12. IEEE.

Doddapaneni, K., Ever, E., Gemikonakli, O., Malavolta, I., Mostarda, L., and Muccini, H. (2012). A model-driven engineering framework for architecting and analysing wireless sensor networks. In *Proceedings of the 3rd International Workshop on Software Engineering for Sensor Network Applications*, pages 1–7. IEEE Press.

Kowal, M., Prehofer, C., Schaefer, I., and Tribastone, M. (2014). Model-based development and performance analysis for evolving manufacturing systems. *at-Automatisierungstechnik*, 62(11):794–802.

Priego, R., Armentia, A., Estévez, E., and Marcos, M. (2016). Modeling techniques as applied to generating tool-independent automation projects. *at-Automatisierungstechnik*, 64(4):325–340.

Rocheteau, J. and Sferruzza, D. (2016). REIFIER: Model-Driven Development of Component-Based and Service-Oriented JEE Applications. In *ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint Malo, France*.

Rodrigues, T., Batista, T., Delicato, F., Pires, P., and Zomaya, A. (2013). Model-driven approach for building efficient wireless sensor and actuator network applications. In *4th International Workshop on Software Engineering for Sensor Network Applications*, pages 43–48. IEEE.

Rodrigues, T., Dantas, P., Pires, P. F., Pirmez, L., Batista, T., Miceli, C., and Zomaya, A. (2011). Model-driven development of wireless sensor network applications. In *IFIP 9th International Conference on Embedded and Ubiquitous Computing*, pages 11–18. IEEE.

Rowe, A., Bhatia, G., and Rajkumar, R. (2010). A model-based design approach for wireless sensor-actuator networks. *AVICPS*, page 1.

Sathe, S., Papaioannou, T. G., Jeung, H., and Aberer, K. (2013). A survey of model-based sensor data acquisition and management. In *Managing and Mining Sensor Data*, pages 9–50. Springer.

Süß, J. G., Pop, A., Fritzson, P., and Wildman, L. (2008). Towards integrated model-driven testing of scada systems using the eclipse modeling framework and modelica. In *19th Australian Conference on Software Engineering*, pages 149–159. IEEE.

Vidal, C., Fernández-Sánchez, C., Díaz, J., and Pérez, J. (2015). A model-driven engineering process for autonomic sensor-actuator networks. *International Journal of Distributed Sensor Networks*, 2015:18.

Yang, C.-H., Vyatkin, V., and Pang, C. (2014). Model-driven development of control software for distributed automation: a survey and an approach. In *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, volume 44, pages 292–305. IEEE.