A fault Detection and Diagnosis Framework for Ambient Intelligent Systems

Ahmed Mohamed / Christophe Jacquet
SUPELEC Systems Sciences (E3S)
Department of Computer Science
3 rue Joliot-Curie, 91192 Gif-sur-Yvette Cedex, France
{Ahmed.Mohamed; Christophe.Jacquet}@supelec.fr

Yacine Bellik
LIMSI-CNRS
Bât 508, B.P 133
91403 Orsay Cedex, France
Yacine.Bellik@limsi.fr

***Abstract*—Ambient intelligence (AmI) systems are smart interactive systems that perceive their surroundings using sensors and act upon them using actuators. One of the most common applications of such systems is Smart Homes. In this context, the ambient system can offer a great level of dependability if it is able to exploit available sensor data in order to autonomously perform diagnosis. However, ambient environments are dynamic in a sense that components, in general, and actuators and sensors, in particular, can be added or removed from the system at run-time. This dynamicity raises new challenges not addressed in the state of the art of fault detection and diagnosis techniques. Unlike classical control theory methods, control-loops between ambient system components cannot be pre-determined at design time. In this paper we propose a new approach based on the modeling of physical phenomena, allowing one to use available resources to predict the values that are supposed to be read by sensors. Comparing the predictions and the real readings allows us to detect potential faults. Fault detection may be followed by fault isolation, which tries to identify the faulty component precisely.**

Keywords-Ambient intelligence; ubiquitous systems; sensor; actuator; fault detection; diagnosis; ontology; physical law; Smart Home; Pervasive Computing.

I. INTRODUCTION

Ambient intelligence (AmI) refers to interactive systems in which the processing and interaction capabilities are embedded into everyday objects. Such systems act upon the environment using actuators and they perceive their surroundings using sensors. The main objective of an AmI environment is to address the needs and preferences of the user. Applications range from enhancing everyday life tasks to monitoring and guaranteeing patients' safety in hospitals. To ensure the achievement of their goals, ambient systems depend strongly upon the proper conduct of tasks that are performed by actuators.

In this context we want to endow such systems with tools allowing them to check autonomously whether or not systems tasks are performed properly. As a matter of fact, when an ambient system sends out orders to an actuator, the proper way to verify whether an order has been executed properly is to exploit the sensors' readings in order to ensure that the state of the environment has changed as expected. For instance, when the system activates a light bulb, the hardware infrastructure and communication capabilities allow the system to verify whether the order has been transmitted properly and that the electric circuit of the light bulb has been closed. However, the light bulb could have been damaged and so it would not be lit properly. So to verify that the light has really been switched on, the readings of the proper light sensors must be considered. New challenges arise: first selecting relevant sensors (here light sensors), second selecting only sensors that are exposed to the actions of specific actuators (here light sensors exposed to the light emitted by a given light bulb). A solution to that from control theory consists in pre-determining closed control loops using ad-hoc sensors. However, one of the main particularities of ambient systems is that, unlike traditional systems, physical resources (mainly sensors and actuators) are not necessarily known at design time. In fact they are dynamically discovered and may appear and/or disappear at run-time, so the solution using pre-determined control loops cannot be adopted in such open environments.

We propose a solution that allows the automatic and dynamic construction of links between actuators and sensors in ambient systems by exploiting available resources at a given time, and using them to perform fault detection and diagnosis (FDD) at run-time. The approach is based on the modeling of the physical phenomena (that we call *effects*) expected to occur in the environment when a given actuator is activated. Effects are characterized by physical laws that can be modeled at various levels of details. These laws depend on physical parameters associated with actuators and sensors types. By exploiting modeled information and physical laws, the system is able to automatically create associations between actuators and sensors. Then by performing the proper calculations, the system deduces the measurement expected from a given sensor when a certain action is performed by an actuator (for instance, an increased temperature level may be expected within a certain time lapse when a heating system is activated).

This way, the system is able, first by comparing these calculated values with the actual sensors readings, to detect the existence of faults (fault detection), then by reasoning over a diagnosis model, to produce an accurate diagnosis (finding the fault source, which could come either from the actuators or from the sensors themselves) at run-time without requiring the explicit coupling of actuators and sensors at design time. The relations between the actual components are entirely deduced at run-time from the characteristics of actuator and sensor types. Therefore it is well adapted to the openness of ambient systems.

This paper is organized as follows. Section 2 is a state of the art of some existing diagnosis techniques, in the fields of

Reference: Ahmed Mohamed, Christophe Jacquet and Yacine Bellik, "A fault Detection and Diagnosis Framework for Ambient Intelligent Systems". *The Ninth IEEE International Conference on Ubiquitous Intelligence and Computing (IEEE UIC 2012)*, Sep 2012. pp. 394-401. The IEEE.

automatic control and AmI. Section 3 describes our fault detection and diagnosis approach. Section 4 shows how this approach may be applied to a complete fault detection example illustrated using our diagnosis simulator. Finally, the conclusion highlights some directions for future work.

II. STATE OF THE ART

Fault detection and diagnosis first appeared in the field of automatic control, in which systems are modeled mathematically in the form of differential equations, or their equivalent transformed formulations [1]. Many works have been done in the field of automatic control to improve system reliability; generally the resulting systems are fault tolerant systems. However, the systems in automatic control are usually pre-defined: manufacturing systems composed of machine tools, robots, transportation systems or well-defined household appliances. These systems do not exhibit some of the particularities of ambient systems such as dynamicity (devices are continuously changing states, reading values, positions, adapting to the context etc.), high heterogeneity of devices, and openness (adding or removing devices at run-time) [2]. In the field of automatic control, fault detection and diagnosis consist of three main tasks:

- Fault detection: finding out if something is not working as expected in the diagnosed system.
- Fault isolation: finding the cause of a detected fault.
- Fault identification: determining the nature and magnitude of a fault.

Fault detection and isolation are the most important tasks in fault detection and diagnosis systems. Fault isolation and fault identification are usually referred to together as fault diagnosis. Fault identification, even though useful, is sometimes ignored as the effort it requires is not worth the resulted information. In this paper we introduce a framework that focuses mainly on fault detection and isolation in the field of AmI systems.

Note that since AmI systems are user-centered, diagnosis can refer to two kinds of tasks: (i) user-behavior diagnosis, which corresponds to verifying whether the user has properly done his/her expected task, (ii) system-behavior diagnosis, consisting of verifying whether the system actuators have performed their task properly. Many techniques are proposed for user-behavior diagnosis, especially in the field of Ambient Assisted Living (AAL); the approach consists in gathering user data (behavior, preferences, etc.) in order to apply machine learning techniques [3] to detect anomalies in user behavior. In this paper we focus on fault detection and isolation of system behavior only.

In addition to the particularities of AmI systems mentioned earlier, one of the main challenges of ambient environments is that services, whose goal is generally to satisfy user's preferences by performing a specific task, are executed in the background such that they are unnoticeable by the user. This requirement causes some difficulties for fault detection because a non-intrusive system cannot decently flood the user with a large number of fault detection data. Conversely, users uninformed of detected faults may continue to rely on failed services without noticing. This can be very critical as AmI systems are becoming increasingly autonomous and complex. That is why many infrastructures for pervasive computing, incorporate fault-tolerant mechanisms. Examples include the Context Toolkit [5], Aura [6], Solar [7], ConFab [8] and Gaia [9]. These systems provide system-level mechanisms for monitoring application components and address particular issues that arise in pervasive computing contexts such as management of heterogeneous resources and distributed computing. In particular the prototype of Gaia implements some fault-tolerance mechanisms, and it has been extended with some fault handling techniques [10]. These mechanisms include heart-beat-based status monitoring, redundant provisioning of alternate services and/or applications, and restarting failed application components. Different failure reasons were identified [9] and classified [10]: components and/or services failing due to low battery power, physical damage, network disconnections or Quality of Service (QoS) problems, and Byzantine failures caused by deliberate attacks on the application. [11] introduces a recovery model for context-aware environments. However it focuses on recovering from design errors at the application level (namely object binding failures), and not on resilience to physical failures.

In our approach we are more interested in a model-based FDD technique, which is a technique based on a system description that is used to define the behavior of each component within the system and the connections between these components [12]. The technique consists in simulating the system's behavior and reasoning over the system model. Obtained information is used to compare the expected system behavior with the actual system behavior, and thus to detect faults. The major challenge of this technique is combinatorial explosion which makes the approach useless for devices composed of a considerable number of components [13]. We claim that we can overcome this problem by describing, at the fault-detection task, only system components and structure but not the behavior. The system's behavior, however, could be used at the fault diagnosis task to isolate the component whose behavior caused the fault. Another technique that is based on the description of the system's behavior is [14], which deals with context aware adaptive applications where adaptation is defined via a set of rules. The technique consists in transforming the rule set into a formal finite-state model. Algorithms are then proposed to analyze the finite-state model in order to detect adaptation faults. The approach is different from ours since they compare system state to an already established set of adaptation fault patterns, whereas we dynamically deduce faults by comparing the real world state to the model dynamically at run-time.

In other works, such as [2], there have been attempts to address the diagnosis problem in AmI using sensor networks. The technique consists in detecting faults within a sensor network by applying a fuzzy logic data fusion approach using a Statistical Process Control and a clustered covariance method [15]. We adopt another approach in which we do not rely on machine learning, instead we base our approach on the proper modeling of the diagnosed system. Models are

then used at run-time to dynamically draw conclusions about the proper operation of the system.

In general, we notice that regardless of the approaches proposed in existing work, it is always supposed that sensors and actuators, whether they are represented in a model or not, are somehow directly linked. In other words the diagnosis system explicitly contains the relationships between actuator actions and sensor states. We claim that building such explicit links is poorly adapted to highly dynamic and open ambient systems. Indeed, as devices are added to and removed from an ambient environment at run-time, it is very difficult for the system designer to thoroughly describe, at design time, how the system will be structured at runtime. For these reasons, we introduce our approach allowing the decoupling of actuators and sensors in the model, while enabling the deduction of the links between them at run-time.

III. OUR APPROACH

A. Overview

In this section we detail our Fault Detection and Diagnosis framework for ambient environments. We start by introducing the context of use of the framework. In Figure 1, the FDD framework is situated within the context of a real ambient system. The latter's most important components, that are necessary to the operation of the FDD framework, are actuators and sensors. These components, and other entities described later, are modeled in order to perform the Fault Detection and Diagnosis Tasks. As illustrated in Figure 1, the FDD framework relies mainly on an environment abstract model, an environment concrete model, and instances of the latter.

The *environment abstract model* is detailed in Figure 2. It defines the structure of the environment model in a way that enforces the decoupling of sensors and actuators at all levels. This is achieved by introducing the concept of effect, which is a modeling of the physical consequence(s) of the actions of actuators onto the environment. The Abstract Model is further discussed in Subsection C.

The *environment concrete model* follows the general structure of the abstract model and defines sensor and actuator types, the expected physical effects, the appropriate physical laws and the relations between all these entities.

An *environment instance* is created at runtime by the *context engine* that intercepts system events and signals. It contains the actual sensors and actuators as well as the actual values of effects produced by actuators and read by sensors.

Because the models of a particular AmI system follows a common abstract model, it is exploitable by the *prediction engine*, responsible for deducing the values expected to be read by the sensors. Comparing these values with the actual sensor readings makes it possible to perform Fault Detection. Then, using the *diagnosis model*, the *diagnosis engine* is responsible for isolating these faults and determining exactly what components are responsible for them. In the following subsections, we discuss the different models composing our FDD framework and the fault detection tasks.

To better explain our approach we will look at the FDD framework from two perspectives; (i) the FDD Framework Models (see Figure 3 and 4), which describes the way the ambient environment and its components are modeled within the framework, and (ii) the general structure and operations of the FDD framework (see Figure 5), in particular how the models defined in (i) are exploited by the FDD framework.

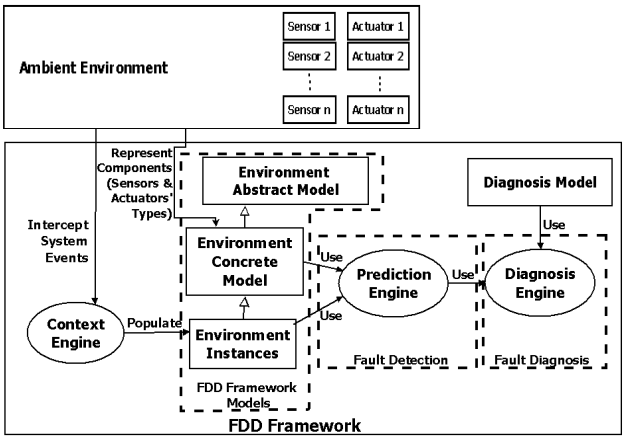

Figure 1. The FDD Framework Achitecture in the AmI context

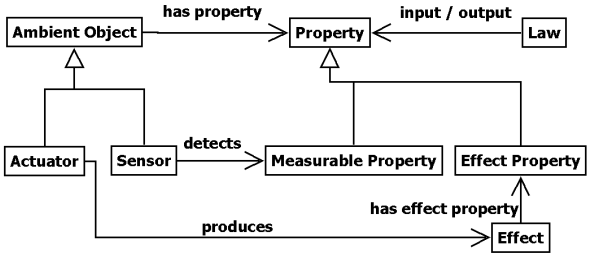

Figure 2. Abstract Model

B. The FDD Framework General Architecture

This part describes the FDD Framework according to two perspectives; a conceptual point of view (Figure 3 and 4), in which we describe the models used by the FDD framework, and an architectural point of view, Figure 5, in which we describe the FDD framework's architecture, operations and use of models.

1) Models used by the FDD framework

In order to the FDD Framework to perform the Fault Detection and Diagnosis tasks it uses information from the following models:

- The Environment's Static Model
- The Environment's Dynamic Model
- The Diagnosis Model

In Figure 3, the "use" relation between the models describes in fact the way the FDD framework uses information from one model to construct the other. For example, as explained earlier, the FDD framework uses information from the static model in order to populate the dynamic model (create environment instances).

The Diagnosis Model is used to achieve fault isolation. Therefore its nature is completely dependent on the type of Diagnosis Engine used. We neither restrict the range of fault isolation techniques, nor the nature of the diagnosis model to be used. In all cases however, the FDD framework uses the static model to build or complete the Diagnosis Model.

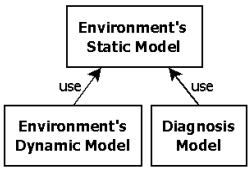

Figure 3. The FDD Framework Models

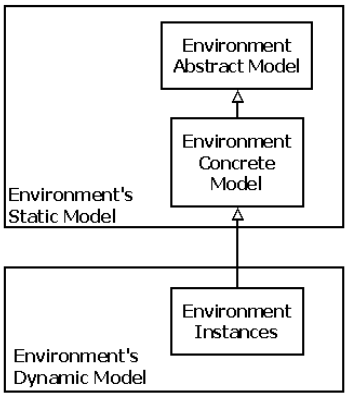

Figure 4. The FDD Framework Models' Hierarchy

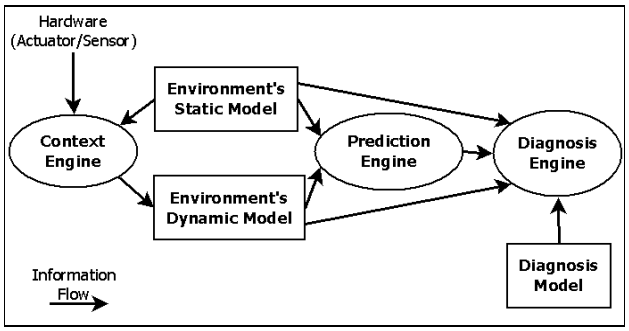

Figure 5. Run-time Architecture of the FDD framework

2) Architecture of the FDD framework

The operations of the FDD Framework depend on which model from the FDD Framework Models is handled. These models are exploited by engines in order to deduce fault detection and diagnosis conclusions. The general run-time behavior of the framework, as shown in Figure 5, can be summarized by these steps:

i) The Context Engine uses information from the hardware layer and from the Environment's Static Model to properly instantiate the real world objects.

ii) Information contained in the Environment's Static Model (Physical Laws to apply and/or Deduced Links between Different Types of Actuators and Sensors) and information contained in the Dynamic model (Actual Instances and their values) are used by the Prediction Engine to calculate the expected values of sensors. Simple comparison between predicted values and real readings of sensors allows us to detect probable faults.

iii) These conclusions (probable faults) with the calculated values, information from the Static Model, information from the Dynamic Model and information from the Diagnosis Model are exploited by the Diagnosis Engine to perform Fault Isolation. This completes diagnosis.

C. The FDD Framework Models

In this part we show how the abstract model allows one to model the ambient environment while enforcing the decoupling of actuators and sensors at design time.

As shown in Figure 4, the environment model can be divided into two main parts: a static one and a dynamic one. The *static model* contains (i) the abstract model composed of generic entities, namely Actuator, Sensor, Effect, etc. and (ii) the concrete model that specializes and concretizes these entities (Light Sensor, Sound Actuator, etc.). Actuators produce Effects, which have Effect Properties (Figure 2). Sensors detect Measurable Properties. Laws relate all these kinds of Properties in order to model physical phenomena. Using laws it is possible to estimate the values detected by the Sensors. The *dynamic model* contains the actual instances of sensors and actuators present in the physical environment. It stores the current state of the environment (sensor values, actuator commands) and it is kept updated at run-time. Section 0 explains how this model is populated and updated.

Let us see how this works on a concrete environment model, corresponding to a lighting system (Figure 7). The abstract Sensor entity is concretized as a Light Sensor entity (or a specific Light Sensor Type), the abstract Actuator entity as a Light Bulb (or a specific Light Bulb Type). Light Sensors and Light Bulbs share an Ambient Property which is the Zone in which they are located (for example the name of the room). A Light Sensor can detect a light level (Ambient Light concretizes Measurable Property). Likewise, a Light Bulb produces a Light Effect (concretization of Effect) which is characterized by a Light Intensity (concretization of Effect Property). A corresponding set of Laws is instantiated in order to calculate the value of the Light Intensity around the Light Sensor entity.

The calculations will use properties such as the position of the Light Bulb and the Light Sensor to determine the distance between the two components, the light intensity emitted by the Light Bulb to determine the received light intensity. A combination law can be used if there is more than one Light Bulb emitting light toward the Light Sensor. It is important to note here that our approach does not impose a level of detail for the physical laws. It is up to the designer to choose the relevant level of granularity. Indeed one can imagine a different modeling for our example, in which the effect of light is represented by a Boolean value (light absent – light present). This freedom to choose the level of granularity is well adapted to AmI systems since their use in real world varies according to context. We can

imagine a smart home design for people with hearing impairment in which the modeling of the effect of sounds is very detailed in order to enhance the perception of sound.

IV. LIGHT SYSTEM FDD EXAMPLE

In this section we present an illustrative example in which we show how our FDD framework could be integrated into a real AmI environment. We suppose that our ambient environment is a smart home, in which we focus only on the lighting system. The living room and the bathroom are equipped with the following devices:

- In the living room:
  - Three light bulbs: two are 23 W fluorescent light bulbs generating 1500 lm each, and one is a 60 W incandescent light bulb generating 800 lm.
  - Two light sensors: one is a photo transistor (with accuracy of ±75% [16]; it is from these accuracy values that we determine the tolerance value for each sensor type) and the other is a photo diode with a current amplifier (with accuracy of ±33%; called "Photo Diode" for short in the rest of the example).
- In the bathroom:
  - A 100 W incandescent light bulb of 1750 lm.
  - A light sensor of type photo resistor (accuracy not guaranteed).

The approximate positioning of these components is illustrated in Figure 6 (the exact x,y coordinates are defined later in the dynamic model; origin is at the top left corner and distances are in centimeters). The transparent circles around actuators have diameters proportional to the produced Light Intensity. Their only purpose is to allow one to quickly compare between actuator effects values. The actual intensity value is written under the actuator type picture. The intensity values read by the sensors are written after their names.

A. The Simulator

To implement the example we use a simulator that we have developed in Java (see Figure 6 for a screenshot).

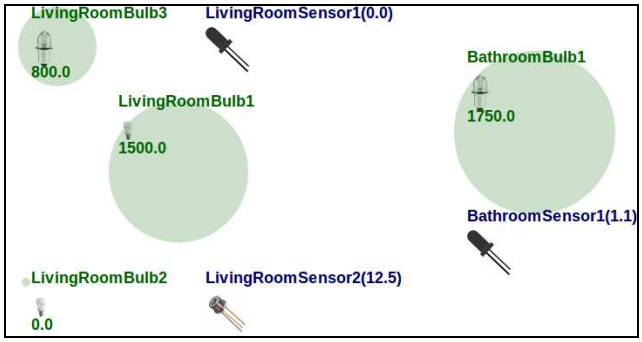

Figure 6. The AmI Environment as presented in the simulator

B. Environment Model

On Figure 4, the Environment's Static Model was described as composed of an Abstract Model and a Concrete Model. Figure 7 shows how the Concrete Model is structured, based on the Abstract Model, in the context of the lighting system. The Concrete Model is defined using a textual triplet-based syntax. For instance to define the sensor type "Photo Transistor" the syntax is:

```
PhotoTransistor is-a Sensor;
```

Where "Sensor" is the Entity defined in the Abstract Model. And to describe the relation between "Photo Transistor" and the detected entity "Ambient Light Intensity" the syntax is the following:

```
AmbientLightIntensity is-a MeasurableProperty;
PhotoTransistor detects AmbientLightIntensity;
```

Where "Ambient Light Intensity" is defined as a "Measurable Property" and where "detects" is a relation defined in the abstract model as the link between entities of type "Sensors" and entities of type "Measurable Property".

This syntax is used to describe all of the Concrete Model depicted on Figure 7. The entities defined in the Concrete Model are the following (in bold, types from the abstract model):

Sensors: *Photo Transistor*, *Photo Diode*, *Photo Resistor*.

Actuators: *Fluorescent-*, *Incandescent Light Bulb*.

Properties:

Tolerance: A property for Sensors. It will be instantiated for Photo Transistors and Photo Diodes, which have tolerance values, but not for Photo Resistors. The value of tolerance of each sensor will be considered at fault detection stage as the threshold for error margin.

Zone: is a property for all actuators and sensors that indicates the name of the room in which they are located. Two objects not in the same room are not supposed to affect each other as far as light is concerned.

2D Position: is a property for all actuators and sensors that is represented by the coordinates of each component; however some actual instances may be without coordinates.

Measurable Property: *Ambient Light Intensity*, an entity that models the expected readings of each Light Sensor.

Effect: *Light Effect*, which is the main effect that is produced by the light actuator. By definition this Entity is a description of the physical phenomena observed as a consequence of the actions of actuators on the ambient environment. In this context it is a description of the light emission phenomena observed when a Light Bulb is on.

Effect Property: *Light Intensity*, which is a property of the Light Effect. It contains the specific value of the light intensity of the Light Effect produced by each light actuator.

Law:

Ambient Light Law Set: this entity contains a set of laws, expressed as mathematical functions, that allows us to estimate the value of "Ambient Light Intensity" that each sensor is supposed to detect. The functions use values from other entities and results from other functions within the same law set to perform calculations. The functions are:

$$\mathbf{S}\text{ame}\mathbf{Z}\text{one}_{(s,a)} = (\mathbf{Z}\text{one}_{(s)} == \mathbf{Z}\text{one}_{(a)}) \quad \textbf{(1)}$$

$$\mathbf{D}\text{istance}_{(s,a)} = \begin{cases} +\infty & \text{if } \mathbf{S}\text{ame}\mathbf{Z}\text{one}_{(s,a)} == false \\ \text{Sqrt}[(\mathbf{X}_{(s)}-\mathbf{X}_{(a)})^2+(\mathbf{Y}_{(s)}-\mathbf{Y}_{(a)})^2] & otherwise \end{cases} \quad \textbf{(2)}$$

Sqrt is the square root function

$$\mathbf{D}\text{irect}\mathbf{L}\text{ight}\mathbf{E}\text{xposure}_{(s,a)} = \mathbf{L}\text{ight}\mathbf{I}\text{ntensity}_{(a)} \,/\, \mathbf{D}\text{istance}_{(s,a)}^{2} \quad \textbf{(3)}$$

$$\mathbf{A}\text{mbient}\mathbf{L}\text{ight}\mathbf{I}\text{ntensity}_{(s)} = \sum_{(a)}[\mathbf{D}\text{irect}\mathbf{L}\text{ight}\mathbf{E}\text{xposure}_{(s,a)}] \quad \textbf{(4)}$$

(1) verifies whether or not a sensor and an actuator are in the same zone.

(2) uses the x,y coordinates to calculate the distance (in centimeters) between an actuator and a sensor that are in the same zone. This function returns an infinite distance value when the two objects are not in the same room.

(3) estimates the light intensity value at a light sensor when exposed to a single light source positioned at a certain distance (calculated from **(2)**) and generating a certain luminous flux. The input parameter luminous flux is the effect property that ensures that **(3)**, and consequently the whole ambient light law set, only considers actuators that produce light effect.

(4) calculates the sum of all the results from **(3)**, which is the sum of the light intensities caused by each single light source on this paticular light sensor. **(4)** is the function that calculates the final theoretical value of the measurable property ambient light intensity around a sensor. The comparison of this value with the actual reading of the sensor is the basis of the fault detection task.

In reality when instantiating actual objects from the hardware layer into the model, it is possible to fail to obtain the coordinates of the object (if no location service is available). In that case the "Ambient Light Law Set" is unusable. For that case we define another instance of Law that is adaptable for the new (lower) level of details in which some components are described:

Ambient Light On Off Law Set: is a set of laws that is used when the coordinates of an object involved in the fault detection and diagnosis are unknown. In that case we apply Boolean functions to determine the expected sensor reading. The functions are:

$$\mathbf{S}\text{ame}\mathbf{Z}\text{one}_{(s,a)} = (\mathbf{Z}\text{one}_{(s)} == \mathbf{Z}\text{one}_{(a)}) \quad \textbf{(1)}$$

$$\mathbf{B}\text{oolean}\mathbf{D}\text{irect}\mathbf{L}\text{ight}\mathbf{E}\text{xposure}_{(s,a)} = \mathbf{S}\text{ame}\mathbf{Z}\text{one}_{(s,a)} \,\&\, \text{isON}(\mathbf{L}\text{ight}\mathbf{I}\text{ntensity}_{(a)}) \quad \textbf{(5)}$$

$$\mathbf{B}\text{oolean}\mathbf{M}\text{ultiple}\mathbf{L}\text{ight}\mathbf{E}\text{xposure}_{(s)} = \text{OR}(\mathbf{B}\text{oolean}\mathbf{D}\text{irect}\mathbf{L}\text{ight}\mathbf{E}\text{xposure}_{(s,a)}) \quad \textbf{(6)}$$

(5) estimates whether or not a sensor is exposed to a single light source that is turned on. The function isON converts the value of light intensity into a true false value.

(6) applies a logical OR function on all results from **(5)**, which are the states of all light sources visible by this sensor. This means that one light source that is ON is enough to activate the light sensor.

It is relevant to note that the prediction engine is able to use, at the same time, entities described in different level of details. As a matter of fact, at run-time the prediction engine uses a matching technique to affect values to their proper parameters, in order to evaluate, first the "Ambient Light Law Set", which is the most detailed law and thus with the highest priority. If it fails (for instance there are no coordinates defined for a sensor), it tries then to evaluate the law set with the next priority value, which is "Ambient Light On Off Law" by exploiting the available information. It is to be noted that this is one possible, and simplified, way to design the environment model for the light context. In fact we can imagine a more realistic definition of the Light Effect entity, which in addition would describe the heat emitted by the light source. There would be a "Heat Emission" effect that would contribute to a model determining the current temperature in the room. Another possible solution would be to add another effect such as "Heat Effect" that will have as property "Heat Emission" the latter will be used in the same way by the set of laws for ambient temperature fault detection. This possibility to have multiple solutions shows a flexibility provided by our approach that might be useful in other context of use.

C. Instance Model

Once the Concrete Model is properly set by the designer, the framework is ready to start the real time fault detection. Before that the model called "Environment Instances" composing the Environment's Dynamic Model (as showed in Figure 4) must be populated with the actual devices from the ambient environment, represented as instances of entities from the Concrete Model. The same triplet syntax is used by the context engine to create instances. Operator "is-a" allows the instantiation of an entity from the Concrete Model; entities of type Property in the Concrete Model are used as predicates to associate them to an instance and give them a value. For instance to declare a fluorescent light bulb (like the actual one in the living room) we use:

```
LivingRoomBulb1 is-a FluorescentLightBulb;
```

This instantiation links automatically "Living Room Bulb 1" to "Light Effect", so by adding:

```
LivingRoomBulb1 value 1500;
```

The value of "Light Intensity" of the "Light Effect" produced by this bulb is set to 1500 lm. The declaration of the sensors also links them directly to the corresponding "Measurable Property" that is "Ambient Light Intensity".
To define the value of the zone in which the latter bulb is in, the entity "Zone" defined in the environment model is used as a predicate:

```
LivingRoomBulb1 Zone 'LivingRoom';
```

The model is then completed following the same logic. The complete dynamic model for our example, generated by the context engine, is the following:

```
LivingRoomBulb1 is-a FluorescentLightBulb;
LivingRoomBulb2 is-a FluorescentLightBulb;
LivingRoomBulb3 is-a IncandescentLightBulb;
BathRoomBulb1 is-a IncandescentLightBulb;

LivingRoomSensor1 is-a PhotoTransistor;
LivingRoomSensor2 is-a PhotoDiodeCA;
BathRoomSensor1 is-a PhotoResistor;
```

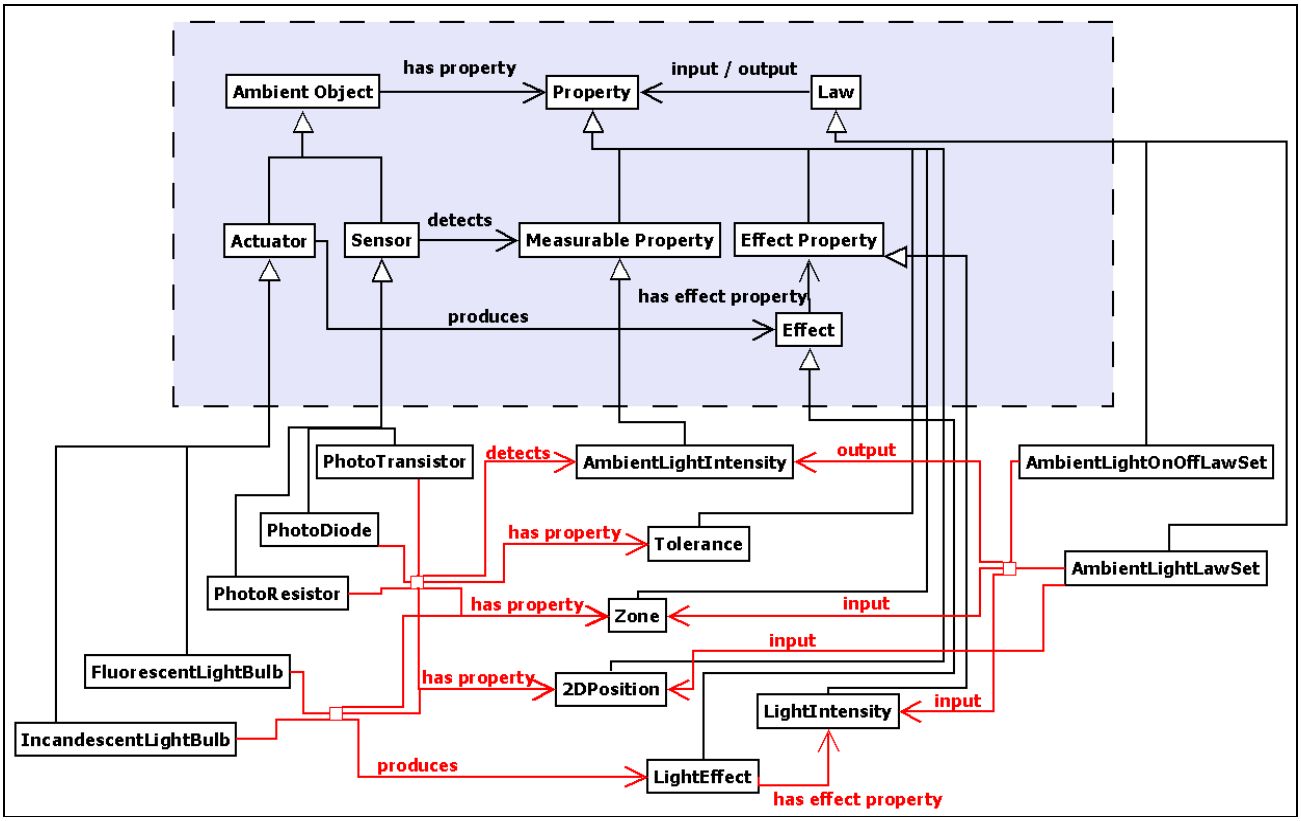

Figure 7. Concrete Model (down) created from the Abstract Model (top) in the context of Light FDD

```
LivingRoomBulb1 2DPosition (150,150);
LivingRoomBulb1 Zone 'LivingRoom';
LivingRoomBulb1 value 1500;
LivingRoomBulb2 2DPosition (50,350);
LivingRoomBulb2 Zone 'LivingRoom';
LivingRoomBulb2 value 0;
LivingRoomBulb3 2DPosition (50,50);
LivingRoomBulb3 Zone 'LivingRoom';
LivingRoomBulb3 value 800;
BathroomBulb1 Zone 'BathRoom';
BathroomBulb1 value 1750;

LivingRoomSensor1 2DPosition (250,50);
LivingRoomSensor1 Zone 'LivingRoom';
LivingRoomSensor1 Tolerance 75.0;
LivingRoomSensor1 value 0.0;
LivingRoomSensor2 2DPosition (250,350);
LivingRoomSensor2 Zone 'LivingRoom';
LivingRoomSensor2 Tolerance 33.0;
LivingRoomSensor2 value 0.0;
BathRoomSensor1 Zone 'BathRoom';
BathRoomSensor1 Tolerance 75.0;
BathRoomSensor1 value 0.0;
```

Note that the previous model is constantly updated via the context engine so a new instance can be introduced at any moment of the execution of the FDD task. It is also possible to add instances manually (by appending the previous model) by a user that can be the designer, an expert or the final user of the system. Note also that "Bath Room Bulb 1" and "Bath Room Sensor 1" does not have valid coordinate values; therefore the prediction engine will resort to using the ambient On/Off Law Set laws in order to predict the sensor's readings.

Now that all the instances are well defined, we can perform real-time fault detection. The values defined previously are in fact updated at run-time via the context engine, which, in general, plays the role of a gateway between the hardware devices and the FDD framework in order to continuously update what we called on Figure 4 the Environment's Dynamic Model.

D. The Fault-Detection Task

The fault-detection task consists in evaluating the calculated (theoretical) value of every "measurable property" supposed to be detected by a sensor. This is done via the prediction engine using the law sets defined in the environment model. The values are then compared to the values actually read from the sensors. If the values are outside the sensor's tolerance margin then an inconsistency is detected. This inconsistency is most likely due to a faulty component. The faulty component is not identified at this stage; we only detect the existence of possible faults.

Using our simulator we create the scenario described in Table 1. The simulator generates approximate readings for the sensors. These readings are compared with the predicted values (calculated via the prediction engine via the law sets) throughout a short scenario (15 seconds). In the table we trace the values every 5 seconds.

We suppose that at Time=10s the "Living Room Bulb 2" is turned on generating a "light effect" having the property "light intensity" of 1500 lm. Theoretically this should have increased the readings of "Living Room Sensor 1" from 0.95 lm to 0.106 lm, and the readings of "Living Room Sensor 2" from 0.036 lm to 0.073 lm. Calculations from the prediction engine clearly reflect this rise in light intensity around the two light sensors exposed to the light emitted by "Living Room Bulb 2". However, at 10 s the values of sensor readings did not change and stayed at 0.090 lm for "Living Room Sensor 1" and 0.034 lm for "Living Room Sensor 2". Thus, by comparing the predicted value to the readings (considering tolerance values of the corresponding sensors), a fault is detected.

In the Bathroom, since the only light sensor and the only light bulb there do not have coordinates, the prediction engine can only estimate the On/Off state of the sensor based on the On/Off state of the Bulb. The comparison between the actual On/Off state of the sensor and the calculated On/Off state do not detect any differences.

TABLE I. FAULT DETECTION SCENARIO

Time [s]	**Sensors Readings [lm]**			**Prediction Engine Calculations [lm]**		
	LivingRoomSensor1	***LivingRoomSensor2***	***BathRoomSensor1***	***LivingRoomSensor1***	***LivingRoomSensor2***	***BathRoomSensor1***
0	0.090	0.034	0.55 (ON)	0.095	0.036	ON
5	0.090	0.034	0.55 (ON)	0.095	0.036	ON
10	**0.090**	**0.034**	0.55 (ON)	0.106	0.073	ON
15	**0.090**	**0.034**	0.55 (ON)	0.106	0.073	ON

V. CONCLUSION

In this paper, we introduced an original approach for the Fault Detection and Diagnosis of AmI systems; the method is based on the definition of the physical phenomena and exploiting the resulting models to simulate the system behavior; the comparison between the real system and the simulated system is the basis of the Fault Detection and Diagnosis approach. The FDD framework is composed of static models that are defined by the designer, following a predefined effect-based abstract model, to describe the diagnosed environment, dynamic models that represent the environment at run-time and two engines: the context engine that populates the dynamic models with the appropriate instances and the prediction engine that evaluates the expected readings of sensors. The approach is adapted to the dynamicity and openness of AmI systems since there is no predetermined relations between actuators and sensors (they are indeed deduced at run-time via the laws defining the physical phenomena), and objects can be added to the model at run-time. In addition this framework offers the freedom to choose the level of granularity in which the system is described, simply by using descriptions of physical phenomena at various levels of details.

The Fault Detection part of the framework is complete, however Diagnosis (isolating the fault and finding out its cause) is yet to be detailed. As future work we envision to fully describe the diagnosis part of the FDD framework. Moreover, the current framework uses models describing mainly the structure (entities and relations between them) of the diagnosed environment, so to improve FDD results we plan de describe the behavior of system components, for instance using finite state machines. Finally, real-scale tests in an experimental ambient environment will be carried out in order to validate the diagnosis framework in real scale.

ACKNOWLEDGMENT

This work has been performed within the CBDP project, a project co-funded by the European Union. Europe is involved in Région Île-de-France with the European Regional Development Fund.

REFERENCES

[1] J. Gertler, Fault Detection and Diagnosis in Engineering Systems, Marcel Dekker, New York, 1998.

[2] Iliasov, A., Romanovsky, A., Arief, B., Laibinis, L., and Troubitsyna, E. "On Rigorous Design and Implementation of Fault Tolerant Ambient Systems". In Proceedings of ISORC. IEEE Computer Society, Washington, DC, USA, 2007 141-145

[3] JC. Augusto, P. McCullagh, V. McClelland, and J-A. Walkden. "Enhanced Healthcare Provision through Assisted Decision-Making in a Smart Home Environment", proceedings of the 2nd Workshop on Artificial Intelligence Techniques for Ambient Intelligence, 2007.

[4] D. Estrin, D. Culler, K. Pister, and G. Sukhatme, "Connecting the Physical World with Pervasive Networks", IEEE Pervasive Computing, 2002, pp.59 69.

[5] Salber, D., Dey, A.K., Abowd, G.D. "The Context Toolkit: Aiding the Development of Context-Enabled Applications". In Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '99), ACM Press, New York, NY, 434- 441.

[6] J. De Sousa and D. Garlan, "Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments" Proc. IEEE-IFIP Conf. Software Architecture, 2002.

[7] Chen, G. and D. Kotz. "Context Aggregation and Dissemination in Ubiquitous Computing Systems". In Fourth IEEE Workshop on Mobile Computing Systems and Applications. pp. 105-114. 2002

[8] J. I. Hong and J. A. Landay. "An Architecture for Privacy-Sensitive Ubiquitous Computing". In MobiSys, 2004.

[9] M.Roman, C.Hess, R.Cerqueira, A.Ranganathan, R.H.Campbell and K.Nahrstedt, "Gaia: A Middleware platform for active spaces", ACM SIGMOBILE Mobile Computing and Communications Review, vol. 6, no. 4, pp.65-67, 2002

[10] S. Chetan, A. Ranganathan, and R. Campbell, "Towards fault tolerant pervasive computing" in Pervasive 2004 Workshop on Sustainable Pervasive Computing, pp. 38-44, Linz/Vienna, Austria, Apr. 2004

[11] D. Kulkarni and A. Tripathi. "A framework for programming robust context-aware applications". IEEE Trans. Softw. Eng., 36:184–197, 2010

[12] Kitts, C., "Managing Space System Anomalies Using First Principles Reasoning". IEEE Robotics and Automation Magazine, Special Issue on Automation Science, v 13, n 4, 2006, pp. 39-50.

[13] J.D. Kleer, "Focusing on Probable Diagnoses", in Proc. AAAI, 1991, pp.842-848.

[14] M. Sama, D. S. Rosenblum, Z. M. Wang, and S. Elbaum, "Model-based fault detection in context-aware adaptive applications," in Proceedings of 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pp. 261-271, 2008.

[15] Shell, J., Coupland, S., and Goodyer, E. N. (2010). "Fuzzy data fusion for fault detection in wireless sensor networks". Computer. IEEE. Retrieved from http://hdl.handle.net/2086/4492.

[16] Tamara A. Papalias and Mike Wong, "Making Sense of Light Sensors", Application notes, CA: Intersil Americas Inc. 2007.