Building Heterogeneous Models at Runtime to Detect Faults in Ambient-Intelligent Environments

Christophe Jacquet, Ahmed Mohamed, Frédéric Boulanger, Cécile Hardebolle, Yacine Bellik

Building Heterogeneous Models at Runtime to Detect Faults in Ambient-Intelligent Environments

Christophe Jacquet[1], Ahmed Mohamed[1,2], Frédéric Boulanger[1], Cécile Hardebolle[1], and Yacine Bellik[2]

[1] SUPELEC E3S, Gif-sur-Yvette, France
[2] LIMSI-CNRS, Orsay, France

Abstract. This paper introduces an approach for fault detection in ambient-intelligent environments. It proposes to compute predictions for sensor values, to be compared with actual values. As ambient environments are highly dynamic, one cannot pre-determine a prediction method. Therefore, our approach relies on (a) the modeling of sensors, actuators and physical effects that link them, and (b) the automatic construction at run-time of a heterogeneous prediction model. The prediction model can then be executed on a heterogeneous modeling platform such as ModHel'X, which yields predicted sensor values.

Keywords: Heterogeneous modeling, models@run.time, models of computation, ambient intelligence

1 Introduction

Ambient intelligence is a vision in which physical environments are equipped with electronic devices that make them sensitive and responsive to the presence of people [3]. Overall, it comprises systems that activate some actuators based on data provided by some sensors, applying reasoning and decision techniques.

However, software and hardware may suffer failures, and ambient intelligent systems are at risk because of their vast number of sensors and actuators. As people are expected to rely on these systems more and more in the future, being able to detect faults is a major concern. This paper focuses specifically on the *detection* of *hardware* faults. It does not investigate the precise identification of the faulty component(s), nor does it deal with software faults.

In software, mechanisms such as exceptions can report failures. Likewise, an actuator can provide a return code, but this reflects only the transmission of orders to the hardware, not the final result. For instance, when the system activates a light bulb, it receives an acknowledgement that confirms the switch-on of the electrical circuit, but this does not necessarily mean that the bulb is really on (the bulb may be damaged for instance). Therefore, a reliable ambient-intelligent application needs to assess at run-time the real status of its actuators.

To address the issue, the designer could pre-determine control loops using designated sensors. However, the particularity of ambient systems is that physical

resources, mainly sensors and actuators, are not necessarily known at design time: instead, they are dynamically discovered at run-time. In consequence, such control loops cannot be pre-determined. The diagnosis strategy, the links between sensors and actuators, need to be automatically determined at run-time.

We have proposed a fault-detection approach that relies only on sensors discovered at run-time, thereby not requiring the addition of specific devices for diagnosis purposes. To achieve this, we have introduced a metamodel for describing ambient-intelligent systems, in which actuators and sensors are modeled independently, and are loosely coupled. The links between them are deduced automatically at run-time thanks to the thorough modeling of the *physical effects* involved, i.e. the physical laws such as light propagation or heat diffusion.

More precisely, we build a *prediction model* automatically at run-time. Executing this model, the system determines what values are expected to be read on sensors. Comparing the output of this prediction model with the actual output of sensors, it is able to detect faults. As the prediction model applies physical laws, it contains calculations. Besides, it must also model the behavior of objects. For example, an actuator such as a light bulb has *state*: it may be switched off, switched on but warming up, or switched on with full brightness. Therefore, the prediction model must also contain state, for instance state machines. In consequence the prediction model is *heterogeneous*, so we use a heterogeneous modeling tool, ModHel'X, to represent it and to execute it.

This overall approach for diagnosis has already been presented; see [6, 5] for more details. The contribution of this paper is the explicit construction and execution of a *heterogeneous* prediction model at run-time. We show that a heterogeneous modeling tool such as ModHel'X is well adapted for this.

The paper is structured as follows. Section 2 summarizes our approach. It relies on an example to show the need for building a heterogeneous model. The automated methodology for generating such a model is detailed in Section 3. Section 4 discusses the benefits and limitations of our current approach; Section 6 summarizes the paper and gives some perspectives.

2 Our Approach

As seen above, ambient intelligent environments being dynamic by nature, the designers cannot pre-determine links between actuators and sensors. Therefore we introduce an approach in which such links can be determined automatically.

Throughout this paper, we assume that all objects can communicate with the system. They send notifications to an *object management* process when entering or leaving the environment and they can describe themselves. We suppose that an object identifies itself as an instance of a given class, which completely describes its features and possible behaviors (see Sections 2.1 and 2.2). In addition, an object may report its coordinates each time it moves. A system such as Ubisense [9] can provide such communication and localization capabilities off-the-shelf.

2.1 A Loosely Coupled Meta-Model for Ambient Systems

In the physical world, actuators and sensors are linked through the phenomena generated by the former and detected by the latter. To reflect this, we have created a metamodel in which we model actuators, sensors and physical phenomena (called *effects*). We model the fact that (a) a certain category of actuators produce a given effect, and that (b) a certain category of sensors detect a given physical property, consequence of a given effect.

For instance, we model that any light actuator (e.g. a lamp) produces light, more precisely an effect that we call *luminous flux*. Conversely, any light sensor detects a physical property called *illuminance*. Illuminance at a given position can easily be calculated knowing the light sources and their luminous fluxes.

This approach is translated into an abstract metamodel (see Fig. 1 left), which is specialized for every kind of effect, for instance for light (see Figure 1 right). The formulae that describe an effect are structured as a set of functions attached to the effect. For the propagation of light they are as follows:

$$\texttt{distance}(A, B) = \sqrt{(A.\texttt{x} - B.\texttt{x})^2 + (A.\texttt{y} - B.\texttt{y})^2} \tag{1}$$

$$\texttt{illum}(A, B) = \frac{A.\texttt{flux}}{(\texttt{distance}(A, B))^2} \tag{2}$$

$$B.\texttt{illuminance} = \sum_{A \in \texttt{LightActuators}} \texttt{illum}(A, B) \tag{3}$$

The notation $Obj.\texttt{prop}$ refers to the property *prop* of the object *Obj*. Some properties *define* an effect (e.g., the flux emitted by a light bulb, its position in space); we suppose that they are known. Some are *observable* by a sensor (e.g., the illumination *received by a light sensor*) and we wish to predict them.

In the formulae above, A and B are free variables that stand for any object. `LightActuators` is the set of light actuators. Formula (1) defines a function, called `distance`, that computes the 2D distance between two ambient objects A and B. In formula (2), $\texttt{illum}(A, B)$ is the illuminance contributed by object A onto object B. Formula (3) states that the total illuminance on object B is the sum of the individual contributions of every light actuator onto object B.

In this example, the light effect is defined quite precisely. More generally, depending on the application's needs, an effect can be defined at various levels of granularity. For instance, we could have used a simple boolean rule ("if a light bulb is on in a room then the light sensors in that room normally detect light").

2.2 Using the Models to Predict Expected Behavior

At run-time, the specialized metamodel is instantiated to reflect the actual objects present in the environment. For instance, let us suppose that at some point a room contains two light actuators `la1` and `la2`, and a light sensor `ls1`. Initially, there is no link between the actuator and sensors. The links will be deduced automatically using available information about their types. For a given

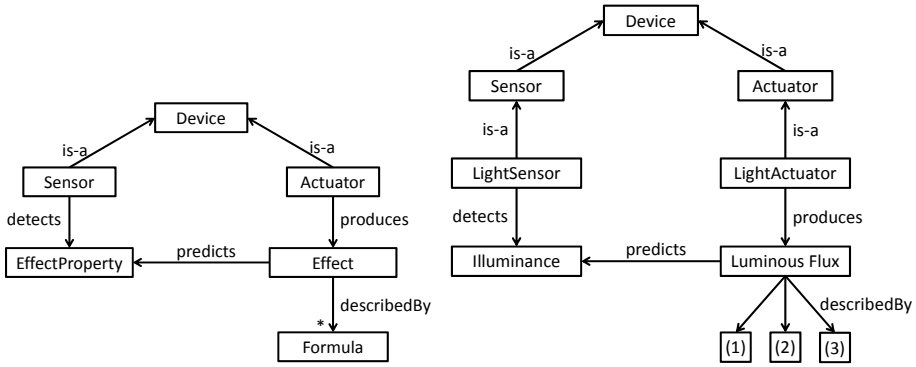

Fig. 1. Abstract metamodel describing the objects of the ambient environment (left), and a specialization of it to account for a specific effect: the emission of light (right). (1), (2) and (3) refer to the corresponding formulae defined in the text.

configuration of the environment, these links make up a *prediction model* that may be used at will to compute new expected sensor output when some known properties change.

In this section, we show on the example with the light sensors and actuators how to build a prediction model, before studying the general automatic method in Section 3. We want to predict the output of a sensor, so we look for the formula that can calculate it. It gives us an expression that may contain function calls. So we recursively use the formulas to apply functions and to eventually determine a simple mathematical expression that gives the result. We need to determine the illuminance perceived by `ls1`, so we start by applying formula (3) symbolically:

$$\texttt{ls1.illuminance} = \texttt{illum}(\texttt{la1}, \texttt{ls1}) + \texttt{illum}(\texttt{la2}, \texttt{ls1}) \tag{4}$$

Then we process the two function calls to `illum` recursively using formula (2), and after that we do the same with `distance` using formula (1):

$$\texttt{ls1.illuminance} = \frac{\texttt{la1.flux}}{(\texttt{distance}(\texttt{la1}, \texttt{ls1}))^2} + \frac{\texttt{la2.flux}}{(\texttt{distance}(\texttt{la2}, \texttt{ls1}))^2} \tag{5}$$

$$= \frac{\texttt{la1.flux}}{(\texttt{la1.x} - \texttt{ls1.x})^2 + (\texttt{la1.y} - \texttt{ls1.y})^2} + \frac{\texttt{la2.flux}}{(\texttt{la2.x} - \texttt{ls1.x})^2 + (\texttt{la2.y} - \texttt{ls1.y})^2} \tag{6}$$

We obtain a symbolic expression that contains no function call, and that depends only on object properties. These properties are of two kinds: (a) structural properties supposed to be known, such as the x and y coordinates of objects, (b) properties depending on the current state of objects, such as `la1.flux` and `la2.flux`. If at each instant we know the fluxes emitted by `la1` and `la2`, then

we can use formula (6) to predict the value reported by `ls1`. An inconsistency between the expected value and the actual value would indicate a fault.

If at some point the expansion of a formula cannot eliminate remaining free variable, this means that the set of formulae is not complete. It denotes an issue in the design of the model of an effect.

We note that actuator characteristics such as the luminous fluxes are not necessarily known directly at all times. For instance, a Compact Fluorescent Lamp (CFL) has warming up and cooling periods that have an impact on its flux. To account for this, a *behavioral model* can be associated with the actuators. For example, Figure 2 is a timed finite state machine that coarsely models a CFL. Upon entry in the system, an actuator is supposed to be in its initial state.

Now that we have seen our approach on an example, let us formalize it.

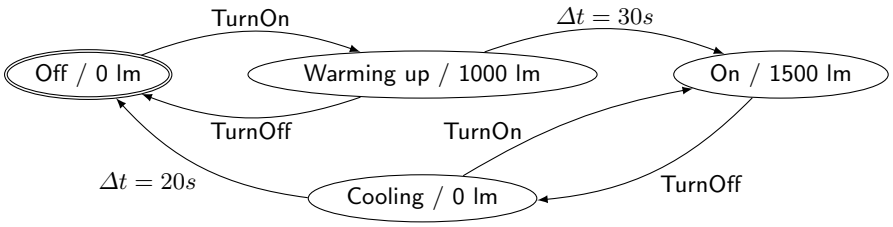

Fig. 2. Simple timed finite state machine for modeling the behavior of a CFL.

2.3 A Workflow for Fault Detection

We can picture diagnosis as a process that runs in parallel with the actual system. This process must have access to all the events of the process responsible for *object management*: notifications for object addition or removal, object movements, and commands sent to objects such as actuators. In response, the diagnosis process can build, update and execute a *prediction model*.

We have just seen that a prediction model may contain parts of different natures: (a) formulae, for instance formula (6), (b) behavioral sub-models, for instance the state machine of Figure 2. Therefore, a prediction model is a *heterogeneous model*, and the adaptation between its heterogeneous parts must be defined carefully. For this reason, we have chosen to model it explicitly using ModHel'X, a heterogeneous modeling tool that is being developed at Supélec (see Section 3.1). The whole run-time workflow may be as follows (see Figure 3):

(1) The object management process maintains the run-time, dynamic model of the system's objects of interest, that conforms to the metamodel in Figure 1.
(2) The diagnosis process deduces links and builds or updates the (heterogeneous) prediction model each time the structure of the environment changes.
(3) The prediction model is executed by a heterogeneous model execution environment.

(4) The results of the execution, predicted measured values, are compared to actual values so as to draw a conclusion: is there a fault?

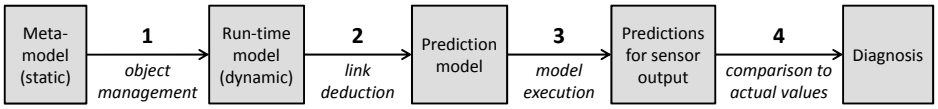

Fig. 3. Overall workflow of the approach.

The main focus of this paper is step (2), described in Section 3.2. First the heterogeneous modeling tool ModHel'X is introduced in Section 3.1, which gives some insights on step (3). Steps (1) and (4) are out of the scope of this paper.

3 Generation of a Heterogeneous Prediction Model

3.1 Heterogeneous Modeling with ModHel'X

ModHel'X [1] is an experimental framework for heterogeneous model execution. ModHel'X allows one (a) to describe the structure of heterogeneous models, (b) to define the semantics of each modeling language used, and (c) to define the semantic adaptation between heterogeneous parts of a model.

ModHel'X relies on a metamodel that defines a common abstract syntax for the structure of all models. As we can see on Figure 6, a model contains *blocks* (gray rectangles with rounded corners) that are connected through *relations* (arrows) between their *pins* (black circles).

One attaches semantics to a model using *models of computation* (MoCs). A MoC is a set of rules that define the nature of the components of a model and how their behaviors are combined to produce the behavior of the model. ModHel'X comes with off-the-shelf MoCs including Timed Finite State Machines (TFSM), Discrete Events (DE) and Synchronous Data Flow (SDF). TFSM are state machines with timed transitions. DE and SDF work like their implementations in Ptolemy II [4]: in DE, blocks are processes that exchange timestamped events that can contain data; in SDF, blocks are data-flow operators that consume and produce a fixed number of data samples on their pins each time they are activated. On Figure 6, the MoC of the overall model is DE, represented by a diamond-shaped label.

Heterogeneity is handled through hierarchy: ModHel'X introduces special blocks called *interface blocks* whose internal behavior is described by a model obeying a specific MoC. Figure 6 contains three interface blocks. They act as *adapters* between the outer MoC and their inner MoC, which may differ. Three aspects must be adapted [1]: *data* (which may not have the same form in the inner and outer models), *time* (the notion of time and the time scales may differ in the inner and outer models) and *control* (the instants at which it is possible or necessary to communicate with a block through its interface).

3.2 Using Effects to Deduce a Prediction Model

In input, the prediction model receives updates about non-structural changes in the environment: movements of objects, commands sent to actuators, etc. In contrast, structural changes, such as the removal or the addition of an object, trigger a re-build of the prediction model. In output, the prediction model sends expected sensor values to a fault detection layer that compares them to actual values and decides which differences are abnormal.

The remainder of this section describes the fully-automated procedure used to build the prediction model.

The first step is to use the effects as pivots between sensors and actuators, as explained on the example of Section 2.2. The procedure starts at sensor outputs and recursively applies functions so as to build a symbolic expression without function calls. This symbolic expression constitutes a computational model to be evaluated at each instant. We use Synchronous Dataflow (SDF) as its model of computation. SDF treats the computational model as a sampled system, computing its new outputs regularly, for instance each second. When recursively applying the functions, two cases can occur:

- If a function contains iterating operators, such as the sigma notation for a sum, then this operator must be expanded depending on the current situation of the environment. For instance, when processing the summation operator that iterates over "all the light sources" in formula (3), we actually perform this iteration and create as many branches in the computational model as there are light sources. So a function with an iterating operator is decomposed into elementary operators, each represented by an SDF block.
- If a function contains no iterating operator, then we choose to translate it into a single SDF block. We could further decompose it into elementary operators, but our choice generates simpler models.

Figure 4 depicts the SDF model obtained for the running example. The upper part deals with light actuator `la1`, the lower part deals with `la2`. This shows how the current situation of the environment determines the structure of the model.

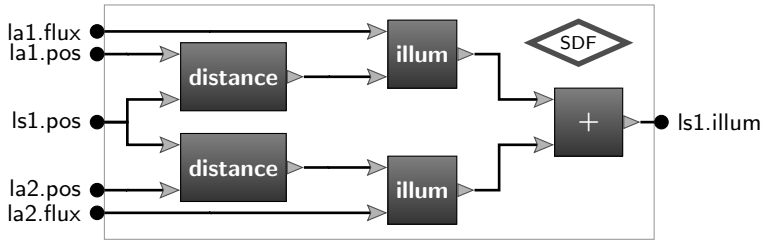

Fig. 4. SDF computational model for the running example.

The second step is to include the actuators' behavioral models into the larger prediction model. This is necessary because some input of the computational

model may depend on the current state of some actuator. For instance, the luminous flux of a light bulb depends its current state, as shown on Figure 2.

The state machines are modeled using the TFSM MoC. Using timed state machines is necessary, because some transitions fire spontaneously after some time (for instance on Figure 2, the state machine transitions to state "on" 30 seconds after entering state "heating up").

At this point we have two parts in the model: a computational dataflow model using the SDF MoC, and a number of state machines using the TFSM MoC. These two parts must be interconnected. The discrete events (DE) MoC is well-suited to compose SDF and TFSM models. Discrete events allow state changes to be propagated to the SDF model. Therefore, in all cases we build automatically a top-level model conforming to the DE MoC. The TFSMs and the SDF model are embedded into this DE model, and semantic adaptation is performed using interface blocks generated from standard patterns.

At the boundary between DE and TFSM, in input, events such as "switch on this lamp" have to be forwarded to the TFSM. In output, the TFSM produces events such as "Light flux now at 1500 lm" that also just have to be forwarded to DE. ModHel'X provides a generic DE/TFSM adapter, to be parameterized with a mapping between DE and TFSM events. Figure 5 shows how the state machine for `la2`, an incandescent lamp, is embedded into a DE interface block (in gray). A similar interface block is added around `la1`'s state machine.

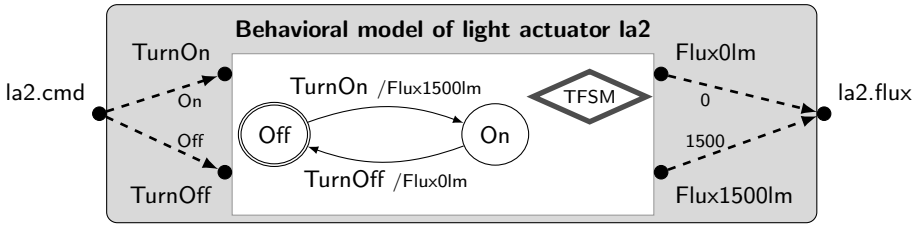

Fig. 5. The TFSM model of an incandescent lamp, embedded into a DE interface block.

A similar adaptation is added around the SDF model of Figure 4. However, in input, an event such as "Light flux now at 1500 lm" cannot be taken into account immediately in general, because an SDF model is a sampled system that reacts at specific instants. Therefore, events are translated into values that are memorized so as to be provided to the model at its next activation instant. For example, "Light flux now at 1500 lm" creates a new value of 1500 on the pin corresponding to the actuator's light flux, and after receiving this event, this new value is emitted at each SDF instant. In output, the adapter sends an event in DE only when a value changes in SDF, so as not to create irrelevant events.

The overall prediction model for our example is depicted on Figure 6. It uses data from an *object management* service to update the computations. These data

may be simple values, fed directly to the computations, or events that trigger state changes of some of the TFSMs, in turn generating new values fed to the computations. The results of the computational model are provided to a fault detection layer that is not described here.

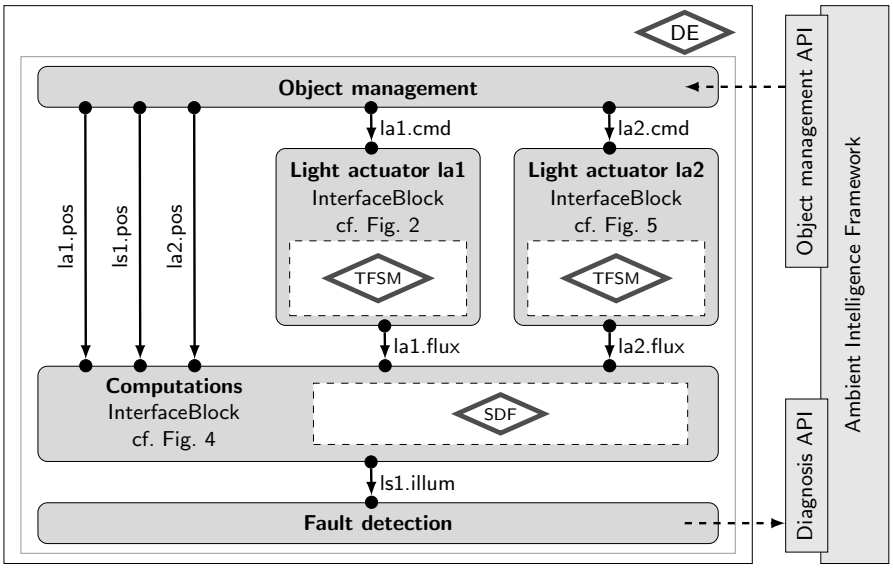

Fig. 6. Overall prediction model for the example.

4 Discussion

We have shown a method for generating a prediction model automatically. The model is regenerated each time the structure of the environment changes, namely when a new object is introduced in the ambient environment, and each time an object is removed. These events are relatively infrequent compared to values changes. When only property values change, it is not necessary to regenerate the model because the structure does not change; we must only continue model execution with new input values.

Another problem arises here: how can we maintain the state of the model when its structure changes? For instance, if in our example at some point a third light actuator `la3` appears in the room, the prediction model is regenerated. While doing so, the states of `la1` and `la2` should be preserved. ModHel'X does not support model evolution currently; this issue has not been addressed yet.

We used ModHel'X to build and run the prediction model because it allows one to specify semantic adaptation explicitly at the border between heterogeneous

models. However, our theoretical approach is not linked with ModHel'X specifically; it could use other modeling tools such as Ptolemy II or Matlab/Simulink as well.

One could argue that the behavioral models used in the example are very rudimentary: for instance a real CFL does not suddenly switch to full light after the warming up time. A more accurate model would represent a continuous increase of the luminous flux from switch on. Integrating such a model in our framework is possible indeed: as the behavioral models of actuator are executed onto a versatile heterogeneous modeling platform, one can use any MoC and not just TFSM, one could use for instance the continuous time (CT) MoC.

Currently, uncertainties are not taken into account. However for the approach to scale to real-world scenarios it will be necessary to do so, as sensors and actuators have limited accuracy. Their margin of error is generally known, so a possible approach would be to calculate uncertainty explicitly along with the calculation of the expected results. Also, the model of a physical effect may be too simplistic. For instance the simple light model described in this paper does not model reflections, transmission of light from one room to another, etc. Such perturbations can have an impact on the actual values measured by the sensors, adding to the uncertainties.

5 Related Work

[2] underlines the importance of failure detection for ambient (or pervasive) systems, especially when they are used in to support healthcare. Devices that cease to function are easy to detect using heartbeat messages, but detecting Byzantine faults (devices yielding erroneous results) is deemed much harder. This is the kind of faults that we try to detect here. Fault detection is an important matter in the related field of autonomic computing too. Most methods learn to recognize faults from data sets: some of them recognize patterns characteristic of faults, others construct a set of normal behaviors [8]. Here, instead of learning techniques, we use automatic model transformations.

Some approaches use static analysis to detect defects in the adaptation logic of context-aware systems [7]. Contrariwise, we perform fault detection at runtime, in order to detect hardware failures. To account for the highly dynamic nature of ambient environments, we do not rely on a pre-existing model of system behavior.

6 Conclusions and Perspectives

This paper has introduced an approach for generating predictions of sensor values in an ambient intelligent environment. These predictions are to be compared with actual values, in order to detect faults. As ambient environments are highly dynamic, one cannot pre-determine a prediction method. Instead, the system must determine at run-time how predictions can be made. Our approach relies on (a) the modeling of sensors, actuators and physical effects that link them, and (b) the automatic construction of a heterogeneous prediction model. This

model can then be executed on the heterogeneous modeling platform ModHel'X. We have shown that this approach is well adapted for taking into account both physical laws and behavioral models of objects, namely actuators.

We have implemented the automatic procedure described in Section 3.2 in a demonstrator that has enabled us to validate this approach in simulated conditions. The simulator uses the ModHel'X API to directly instantiate the prediction model. A next step would be to test it in real-scale in an ambient intelligent environment.

Other perspectives include a focus on managing the evolution of the prediction model when its structure changes, and using various MoCs for finer-grained behavioral models, beyond simple state machines. The uncertainties stemming from limited component accuracy and simplifications made when modeling physical effects will need to be dealt with too. Future work could also investigate the decisional process involved when eventually making a diagnosis.

References

1. Boulanger, F., Hardebolle, C., Jacquet, C., Marcadet, D.: Semantic adaptation for models of computation. In: Proc. Int'l Conf. Application of Concurrency to System Design (ACSD). pp. 153–162. IEEE (2011)
2. Chetan, S., Ranganathan, A., Campbell, R.: Towards fault tolerance pervasive computing. IEEE Technology and Society 24(1), 38–44 (2005)
3. Ducatel, K., Bogdanowicz, M., Scapolo, F., Leijten, J., Burgelman, J.C.: Scenarios for ambient intelligence in 2010. Tech. rep., European Commission (2001)
4. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S.R., Xiong, Y.: Taming heterogeneity - the Ptolemy approach. Proc. IEEE 91(1), 127–144 (2003)
5. Jacquet, C., Mateos, M., Bretault, P., Jean-Bart, B., Schnepp, I., Mohamed, A., Bellik, Y.: An Ambient Assisted Living Framework with Automatic Self-Diagnosis. International Journal in Advances in Life Sciences 5(1/2) (Jul 2013), to appear
6. Mohamed, A., Jacquet, C., Bellik, Y.: A fault Detection and Diagnosis Framework for Ambient Intelligent Systems. In: Proceedings of the Int'l Conference on Ubiquitous Intelligence and Computing (UIC). pp. 394–401. IEEE (September 2012)
7. Sama, M., Rosenblum, D.S., Wang, Z., Elbaum, S.: Model-based fault detection in context-aware adaptive applications. In: Proceedings of the 16th International Symposium on Foundations of software engineering. pp. 261–271. ACM (2008)
8. Shevertalov, M., Lynch, K., Stehle, E., Rorres, C., Mancoridis, S.: Using search methods for selecting and combining software sensors to improve fault detection in autonomic systems. In: SBSE. pp. 120–129. IEEE (2010)
9. Steggles, P., Gschwind, S.: The Ubisense smart space platform. In: Adjunct Proceedings of the Third International Conference on Pervasive Computing. vol. 191, pp. 73–76 (2005)