HAL open science

Discrete Event Modeling and Simulation for IoT Efficient Design Combining WComp and DEVSimPy Framework

S Sehili, Laurent Capocchi, Jean François Santucci, Stéphane Lavirotte, Jean-Yves Tigli

To cite this version:

S Sehili, Laurent Capocchi, Jean François Santucci, Stéphane Lavirotte, Jean-Yves Tigli. Discrete Event Modeling and Simulation for IoT Efficient Design Combining WComp and DEVSimPy Framework. 5th International Conference on Simulation and Modeling Methodologies, Technologies and Applications, Jul 2015, Colmar, France. 10.5220/0005538300440052 . hal-01343553

HAL Id: hal-01343553

https://hal.science/hal-01343553v1

Submitted on 13 Jul 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License

Discrete Event Modeling and Simulation for IoT Efficient Design Combining WComp and DEVSimPy Framework

S. Sehili[1], L. Capocchi[1], J. F. Santucci[1], S. Lavirotte[2] and J. Y. Tigli[2]
[1]*SPE UMR CNRS 6134, University of Corsica, Corte, France*
[2]*I3S (UNS - CNRS), 930 Route des Colles - BP 145, 06903 Sophia-Antipolis, France*
{*sehili, capocchi, santucci*}*@univ-corse.fr, stephane.lavirotte@unice.fr, tigli@unice.fr*

Keywords: Internet of Things, Discrete-Event System, Object Oriented Modeling, Modeling, Simulation, Ubiquitous computing.

Abstract: One of today's challenges in the framework of ubiquitous computing concerns the design of ambient systems including sensors, smart-phones, interconnected objects, computers, etc. The major difficulty is to propose a compositional adaptation which aims to integrate new features that were not foreseen in the design, remove or exchange entities that are no longer available in a given context. In order to provide help to overcome this difficulty, a new approach based on the definition of strategies validated using discrete-event simulation is proposed. Such strategies make it possible to take into account conflicts and compositional adaptation of components in ambient systems. These are defined and validate using a discrete-event formalism to be integrated into a prototyping and dynamic execution environment for ambient intelligence applications. The proposed solution allows the designers of ambient systems to define the optimum matching of all components to each other. One pedagogical example is presented (switch-lamp system) as a proof of the proposed approach.

1 INTRODUCTION

The rapid progress of technology related to sensor networks leads us to a merger between the physical world and the virtual world where the interconnection of objects with a level of intelligence will lead to a revolution in the creation and the availability of service that will gradually change our way to act on the environment. The definition of such complex systems involving sensors, smart-phone, interconnected objects, computers, etc. results in what is called ambient systems. The software development of such systems belongs to ubiquitous computing domain. One of todays challenges in the framework of ubiquitous computing concerns the design of such ambient systems. One of the main problems is to propose a management adapted to the composition of applications in ubiquitous computing. The difficulty is to propose a compositional adaptation which aims to integrate new features that were not foreseen in the design, remove or exchange entities that are no longer available in a given context. We have being focused on the WComp environment (Hourdin et al., 2013) which is a prototyping and dynamic execution environment for Ambient Intelligence applications created by the Rainbow research team of the I3S laboratory, hosted by University of Nice - Sophia Antipolis and CNRS. It uses lightweight components to manage dynamic orchestrations of Web service for device, like UPnP (Universal Plug and Play), discovered in the software infrastructure. In the framework of the WComp, it has been defined a management mechanism allowing extensible interference between devices. In order to deal with the asynchronous nature of the real world, WComp has defined an execution machine for complex connections. In (Sehili et al., 2014) we have presented how the DEVS formalism can be used in order simulate the behavior of IoT components before any implementation using the WComp environment. The interest of this approach has been pointed out on a pedagogical example which allowed to show that using the DEVS formalism conflicts can be detected using simulation before any implementation.

In this paper we propose the definition a modeling and simulation (M&S) scheme based on the DEVS formalism in order to specify at the very early phase of the design of an ambient system: (i) the behavior of the components involved in the ambient system to be implemented; (ii) the possibility to define a set of strategies which can be implemented in the execution machine; (iii) the automatic generation of WComp code corresponding to a selected strategy.

The interest of such an approach is twofold: (i) the behavior involved the DEVS models are used to write the methods required in order to code the components when using the WComp environment (automatic generation of WComp code); (ii) the DEVS implementation of the different strategies are used in order to check them before implementation.

The rest of the paper is as follows: Section 2 gives the context of the study by presenting the interest in combining DEVSimpy simulation and Wcomp framework for IoT system design. Section 3 presents the background of the work including the DEVS formalism, the DEVSimPy framework and the WComp. Section 4 deals with the validation of the approach through a complex case study involving different kinds of sensors which have been used in order to manage the lighting of a house. We points out how different strategies can be compared using DEVS simulation and validated before their implementation using the WComp Framework. We also explain how the WComp code can be automatically generated from the DEVS model of the selected strategy. The conclusion and future work are given in section 5.

2 CONTEXT

A synchronous automaton of an IoT component is dissociated form its asynchronous execution machine that manages its I/O using strategies (see Figure 1).

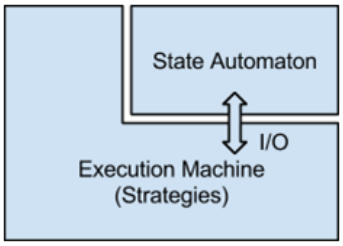

Figure 1: State automaton with its execution machine.

This main challenge is to define the appropriate strategies employed by the execution machine. It is the execution machine that encapsulates the automaton and that will handle asynchronous I/O. Therefore, the executing machine has asynchronous communications with its environment. This encapsulation leads to a combinatorial explosion of the potential strategies that results in complex systems. The issue is whether the execution machine behavior will remain consistent with the model of the automaton (with possible additional constraints).

There is a lack of work in the literature concerning the execution machine exhibited in a number of different strategies. The problem of managing the synchronous automaton is the management of the outbreak of the execution cycle (which should not be preempted). There are many possible strategies which could be adopted: (i) triggering a cycle when an event happens in the automaton; (ii) triggering a cycle on a certain type of event; etc. Ideally, the modeler should have the choice of the execution machine he wants to use.

This paper shows how the DEVS formalism is suitable to model synchronous automatons and check the strategies of the execution machine in a context of IoT system design. It also presents the strong ability of WComp to design IoT component based on the strategies defined with DEVSimPy which is a framework dedicated to DEVS M&S. Furthermore, the strategies defined using DEVSimPy are fully integrated in WComp.

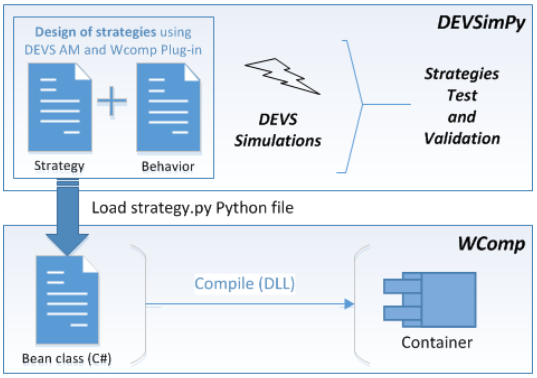

Figure 2: Design and test of WComp execution machines (strategies) using DEVSimPy framework.

Figure 2 depicts the use of DEVSimPy framework to perform the design and test of strategies involved in WCoimp execution machines. The behavior of a DEVS model is expressed through specifications of a finite state automaton. However, this DEVS specifications represent both the state automation and the execution machine. The interest of using DEVS is the ability to define as many strategies as DEVS model specifications. DEVS simulations are used to test and validate the strategies defined by the user in a Strategy.py Python file. This file can be dynamically loaded in the Bean class of WComp and then can be used in order to change the properties of the C# component obtained after compilation.

In the following section, background information as the DEVS formalism, DEVSimPy framework and WComp are outlined.

3 BACKGROUND

3.1 DEVS and DEVSimPy Framework

Since the seventies some formal work have been directed in order to develop the theoretical basements for the M&S of dynamical discrete-event systems (Zeigler, 2003). DEVS (Discrete EVent system Specification) (Zeigler et al., 2000) has been introduced as an abstract formalism for the modeling of discrete-event systems, and allows a complete independence from the simulator using the notion of abstract simulator.

DEVS defines two kinds of models: *atomic models* and *coupled models*. An atomic model is a basic model with specifications for the dynamics of the model. It describes the behavior of a component, which is indivisible, in a timed state transition level. Coupled models tell how to couple several component models together to form a new model. This kind of model can be employed as a component in a larger coupled model, thus giving rise to the construction of complex models in a hierarchical fashion. As in general systems theory, a DEVS model contains a set of states and transition functions that are triggered by the simulator.

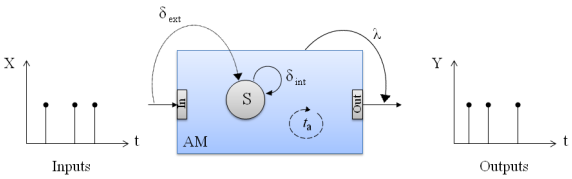

Figure 3: DEVS atomic model in action.

The Figure 3 describes the behaviour of a discrete-event system as a sequence of deterministic transitions between sequential states (S). The atomic model (AM in Figure 3) reacts depending on two types of events: external and internal events. When an input event occurs (X), an external event (coming from another model) trigger the external transition function $\delta_{ext}(X,S)$ of the atomic model in order to update its state. If no input event occurs, an internal event trigger the internal transition $\delta_{int}(S)$ of the atomic model in order to update its state. Then, the output function $\lambda(S)$ of the atomic model is executed to generate the outputs (Y). $t_a(S)$ is the time advance function which determine the life time of a state.

DEVSimPy (Capocchi et al., 2011) is an open Source project (under GPL V.3 license) supported by the SPE (Science pour l'Environnement) team of the UMR CNRS 6134 Lab. of the university of Corsica "Pasquale Paoli". This aim is to provide a GUI for the M&S of PyDEVS and PyPDEVS (Li et al., 2011) models. PyDEVS is an Application Programming Interface (API) allowing the implementation of the DEVS formalism in Python language (Perez et al.,). PyPDEVS is the parallel version of PyDEVS based on Parallel DEVS formalism (Chow and Zeigler, 1994) which is an extension of DEVS formalism. The DEVSimPy environment has been developed in Python with the wxPython (Rappin and Dunn, 2006) graphical library without strong dependencies other than the Scipy (Jones et al., 2001) and the Numpy (Oliphant, 2007) scientific python libraries. The basic idea behind DEVSimPy is to wrap the PyDEVS API with a GUI allowing significant simplification of handling PyDEVS/PyPDEVS models (like the coupling between models or their storage into libraries).

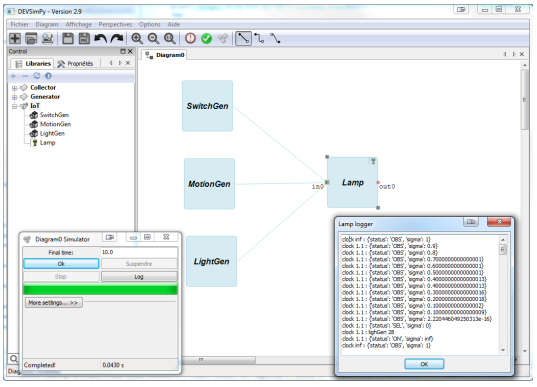

Figure 4: DEVSimPy with simulation dialogue window.

Figure 4 shows the DEVSimPy interface with four interconnected DEVS atomic models (Lamp, MotionGen, LightGen and SwitchGen on the right of the interface) instantiated from the "IoT" library frame (left of the interface) into a coupled model called "Diagram0". The simulation is performed by using the "Diagram0 Simulator" dialogue windows and the simulation trace is printed in the "lamp logger" window. A plug-in manager is proposed in order to expand the functionalities of DEVSimPy allowing their enabling/disabling through a dialog window. In this paper, a plug-in is used to allow the transposition of the execution machine strategies validated with DEVS simulation to WComp environement.

3.2 WComp for IoT

Ubiquitous computing involves collaboration between all kinds of components to build dynamically new applications that are adapted to the physical environment. The complexity of such a system to sense and adapt to the environment involves solving problems related to memory management, time constraints, etc., between all communicating objects.

Mechanism to address this concern must be proposed by middleware for ubiquitous computing. Several variety of middleware tools have been defined in the recent years in order to perform ubiquitous computing : (a) HOMEROS (Seung et al., 2004) middleware architecture which allows high flexibility in the environment of heterogeneous devices and user ; (b) EXEHDA (Lopes et al., 2014) middleware which manages and implements the follow-me semantics in which the applications code is installed on-demand on the devices used and this installation is adaptive to context of each device ; (c) WComp (Hourdin et al., 2013) middleware based on a software infrastructure, a service composition architecture and a compositional adaptation mechanism. In this paper we are interested in the WComp middleware.

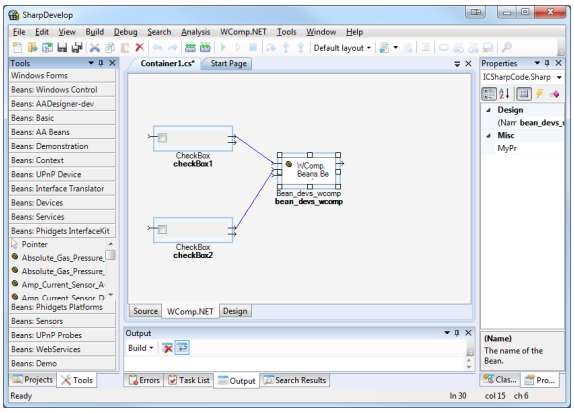

Figure 5: WComp framework.

The architecture of WComp framework is organized around containers and designers (Figure 5). Containers manage the instantiation, designation and destruction of components. A Designer runs containers for instantiation and destruction of components or connections between components in the assembly using an adequate formalism. WComp uses the .NET framework which has a container implemented in C# language.

The components in WComp are implemented with a object oriented approach in the Bean classes. The component is a self-contained class that contains properties and methods needed to communicate with other components. Methods are executed when the component receives an event from others components. The manner of executing these methods (*state automaton*) depending on some inputs is called the *execution machine*. The application needs to be context awareness. For this, the components need to have an adaptive behavior and several ways to manage the execution machine are defined as strategies.

In WComp, the implementation of the execution machine is done manually in the methods of the class Bean. To illustrate this point, we reuse the pedagogical example of a lamp connected with two switches presented in (Sehili et al., 2014). The lamp can be enabled/disabled simultaneously from the two switches and a strategy for the execution machine is needed to prevent and deal with any possible conflicts. We can observe two behaviors according to the assembly that involves the definition of two execution machines (strategies) of the behaviors. The two code blocks below represent the two strategies corresponding to the behaviors of the switches implemented in the Control method of the Bean class. This method is written for each class Bean. In WComp framework, we define each execution machine in a separated class.

The lines [3-7] in the first code block below alter the received event as follow: if the lamp state is ”light_ON”, it changes to ”light_OFF”. This implementation represents the push button switch behavior.

```
1 public bool lightstate = false;
2 public void ControlMethod() {
3  lightstate =! lightstate;
4  string msg = "light_OFF";
5  if (lightstate) msg = "light_ON";
6  if (PropertyChanged != null)
7    PropertyChanged(msg);
8 }
```

The lines [2-5] in the second code block below concern the modification of the lamp status depending on the old state. If the lamp status is ”light_ON”, we alter the state to ”light_OFF” else we keep the initial state (”light_ON”). This implementation represents a toggle switch behavior.

```
1 public void ControlMethod(bool on) {
2    if(on){
3      if (PropertyChanged != null)
4        PropertyChanged("light_ON");
5    }else{ PropertyChanged("light_OFF"); }
6 }
```

After the implementation of the methods in each class, we have to compile the Bean classes to obtain two executable files (DLL) which are inserted in the resulting assembly to be instantiated and connected in applications.

The question that arises is how to choose between the two Bean classes before any instantiation and execution of the assembly? With the help of DEVS simulation, we are able to defined an approach to check differente strategies to be instantiated in the Bean class early in the desing process.

4 CASE STUDY: SMART LIGHTING

4.1 Description of the System

To validate our approach, we chose to deal with an application dedicated to control home lighting using sensors and actuators.

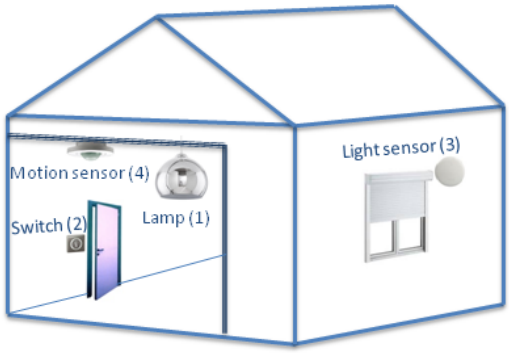

Figure 6: Description of the system.

The case study involves four components (Lamp, Switch, Light sensor, Motion sensor in Figure 6) to be assembled:

1. The *Lamp* component has two states "On" and "Off" with a level of intensity i ($i \in [0-100]$)
2. The *Switch* component sends the activation/desactivation messages "On"/"Off" to the lamp. It is connected to a preemptive port and it generates random events
3. The *Light sensor* component sends messages corresponding to a brightness levels in percent (%) such as a value greater than 50 involves activation for the connected object ("on" for the lamp) otherwise deactivation ("off" for the lamp)
4. The *Motion sensor* component sends messages corresponding to a proximity level compared to the sensor such as a level between 1 and 5 involves the detection of a person in the sensor's range of action this implies that the connected object receiving these values can be activated in this range ("on" for the lamp) or disable if the level is zero (off for the lamp)

We present in Figure 7 the interconnection between the various components (Lamp, Switch, Light sensor, Motion sensor) which represents our modeling approach. Three event generators are connected with the Lamp model which embeds strategies. Then we define four strategies associated with the previously presented assembly. The strategies are coded in

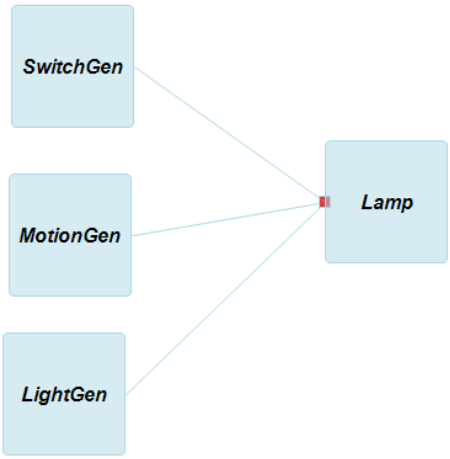

Figure 7: Assembly of the system.

the "Lamp" model with features allowing to perform the proposed approach.

4.2 Description of the Strategies

The interconnection between all components is feasible in several ways. In this part we have defined the possible cases for this interconnection to further illustrate the applications of ambient computing. Two scenarios are presented in the following sections: (i) buffer management of "Lamp model", (ii) preemption management.

4.2.1 First Scenario: Buffer Management

The component *Lamp* has a buffer of events and is connected with the components *LightGen* and *MotionGen* (Figure 7). The behavior of *MotionGen* is to generate a message every 0.1 units of time. The behavior of *LightGen* is to generate a message every 0.7 units of time. The buffer will have several messages *MotionGen* and at least one message of *LightGen*. We define two strategies according to the first scenario.

Strategy I

The first strategy is the basic one:

- If it is bright (*LightGen* > 50) whatever presence is detected or not detected, if the *Lamp* is 'on' it passes 'off' and if it is 'off' it remains 'off'
- If it is dark (*LightGen* > 50) and presence is detected, if the Lamp is 'on' is remains 'on' and if it is 'off' it passes 'on'
- If it is dark (*LightGen* > 50) and presence not detected, if the lamp is 'on' it passes 'off' and if it is 'off' it remains 'off'

Strategy II

The second strategy allows us to improve the energy management.

- If *MotionGen* decreases (person going away) and *LightGen* > 50 (it is bright) then : (i) if *Lamp* is 'on' it passes 'off' ; (ii) if the Lamp is 'off' it remains 'off'
- If *MotionGen* decreases (person going away) and *LightGen* < 50 (it is dark) then : (i) if *Lamp* is 'on' its intensity decreases with distance; (ii) if *Lamp* is 'off' it remains 'off'
- If *MotionGen* increases (person approaching) and *LightGen* > 50 (it is bright) then : (i) if *Lamp* is 'on' it passes 'off' ; (ii) if *Lamp* is 'off' it remains 'off'
- If *MotionGen* increases (person approaching) and *LightGen* < 50 (it is dark) then : (i) if *Lamp* is 'on' its intensity increases with distance; (ii) if *Lamp* is 'off' it passes 'on' with a proportional level to the proximity given by *MotionGen*

4.2.2 Second Scenario: Preemption Management

The component *Lamp* is connected with the three components *LightGen*, *MotionGen* and *SwitchGen* of the Figure 7. The connection port of *SwitchGen* component will be considered as preemptive in the first case and non-preemptive in the second case both for the component *Lamp*. *SwitchGen* generates random events. The Lamp component use its buffer only for the non-preemptive case. We define two strategies according to the second scenario.

Strategy III

"SwitchGen" is preemptive:

- If the component *Lamp* is 'on' and receives 'on' it remains 'on' and inversely
- If the component *Lamp* is 'on' and receives ' off it passes to' off' and inversely

Strategy IV

"SwitchGen" is not preemptive:

- If no *SwitchGen* event in the buffer, then the strategy I is applied.
- If at least one *SwitchGen* event is in the buffer, then the events of *MotionGen* are not considered and the events of *LightGen* override the *SwitchGen* events (energy management):
  - If it is bright (*LightGen* > 50) and the lamp is 'on' it passes to 'off'
  - If it is bright (*LightGen* > 50) and the lamp is 'off' it remains 'off'
  - If it is dark (*LightGen* < 50) and the lamp is 'on' it remains 'on'
  - If it is dark (*LightGen* < 50) and the lamp is 'off' it passes to 'on'

4.3 DEVS Modeling

The modeling of the previous system has been realized using the DEVS formalism. In order to highlight the different scenarios of the section 4.2 in the Lamp model we have defined the DEVS specifications using the state automaton of the Figure 8. In this automaton we define five states: the lamp states ("ON", "OFF"), the observation state "OBS", the selection state "SEL" and the preemption state "PRE".

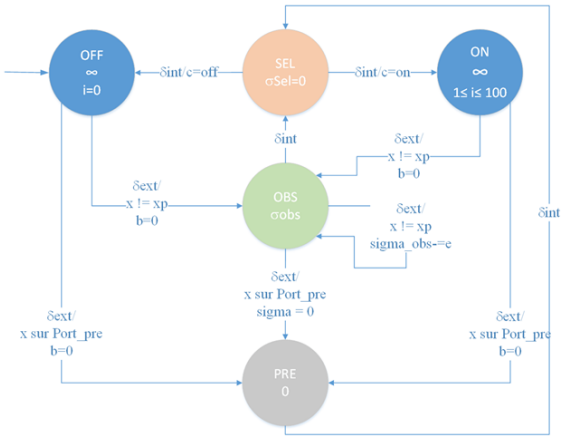

Figure 8: *Lamp* state automaton integrating strategies.

The behavior of the Lamp are described in a sequence of transitions (internal transition and external transition) between sequential states ("ON", "OFF", "OBS", "SEL", "PRE") as described below:

- In the external transition δ_{ext} we define two cases:
  - The first case is when the received message is preemptive, it will be inserted into the preemptive message list; if the initial state of the message is "on" or "off" then the system passes to the preemption state "PRE" and the time advance is set to 0.
  - The second case is when the received message is not preemptive, it will be inserted in the buffer of messages "b"; if the initial state in "on" or "off", the system passes to the observation state "OBS" and the time advance is set to "sigma_obs" else if the initial state is "OBS" the time advance is elapsed.

- In the internal transition δ_{int}: If the initial state is in "PRE" or "OBS" then the system passes to the selection state "SEL" and the time advance t_a is set to "sigma_sel" else if the state is "SEL" and the preemptive list is not empty and the preemptive value is "on" then the lamp state is "on" else "off"; if the preemptive list is empty we apply one of the four strategies written before in section 4.2 and the time advance is set to INFINITY.

We have used the DEVSimPy framework to implement the IoT component library including the four atomic models *SwitchGen*, *MotionGen*, *LightGen* and *Lamp*. The Lamp model has been implemented using the preceding DEVS automaton (Figure 8). The Figure 4 depicts the modeling of the case study with DEVSimPy.

4.4 DEVS Simulation

The representation of the system using the notions of hierarchy and modularity offered by the DEVS formalism allows us to manipulate and reuse the system that can be quickly simulated. In our work we performed the simulations of the described strategies in section 4.2 using DEVSimPy framework.

Figure 9 depicts the simulation results obtained using strategy II which improves the management of energy. For that we defined parameters sigma_obs = 1, sigma_sel = 0 with an interval of simulation equal to 10.

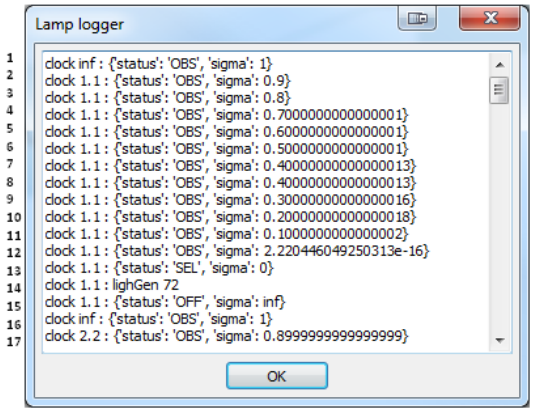

Figure 9: Simulation results obtained with strategy II.

The results shown in the logger (Figure 9) represent the transitions between the state of the model *Lamp* when the strategy II is applied. The model *Lamp* is in the observation state "OBS" (line 1). During 0.1 units of time, the models *MotionGen* and *LightGen* generate messages every 0.7 units of time (line 2-12). Once the time advance is achieved (sigma=1) the system move to selection state "SEL" (line 13) and *Lamp* state change to "OFF" depending of its previous state (line 14). This results allows us to validate the strategy II and then generate the strategy Python file to be integrated in a Bean class into WComp:

```
1 def Strategy2(lightGen, L_MotionGen, previous_state, power):
2  if (L_MotionGen[0].value[1] < L_MotionGen[-1].value[1]):
3    if (lightGen > 50): state = 'OFF'
4    else:
5      if (previous_state == 'ON'):
6       state = 'OFF'
7       power -=(L_MotionGen[-1].value[1]-L_MotionGen[0].value[1])*20
8      else: state = 'OFF'
9  else:
10  if (lightGen > 50): state = 'OFF'
11  else:
12    power += (L_MotionGen[-1].value[1]-L_MotionGen[0].value[1])*20
13    state = 'OFF' if previous_state == 'ON' else 'ON'
14 return (state, power)
```

In the previous code, the test of the motion is considered with a conditional statement (line 2,9) that considers the last value of the array *L_MotionGen*. The test of the level of light is assumed by the *lightGen* variable (line 3,10). Depending on the values of *L_MotionGen*, *lightGen* and *previous_state* variables, the *power* and *state* variables are updated (line 14).

4.5 Integration of Strategies in WComp

The integration of strategies in WComp starts by defining the DEVS atomic model corresponding to the component in which strategies are identified (using functions) in DEVSimPy environment (Lamp in the case study). These strategies will be defined in a dedicated interface from a DEVSimPy local plug-in. The access to the local plug-in will be through the context menu of the atomic model only when the general plug-in called "WComp" of strategies is activated (Figure 10).

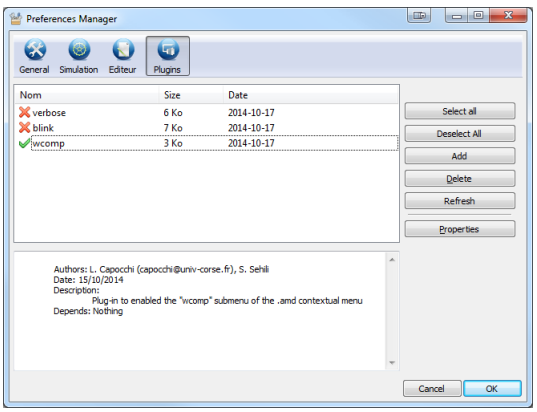

Figure 10: General plug-in *WComp* in DEVSimpy.

Once simulations are performed and strategies are validated in the DEVSimPy framework, we load the strategies file that contains strategies in WComp (Figure 11). This is done due to through IronPython which

is an implementation of Python for .NET allowing us to leverage the .NET framework using Python syntax and coding styles.

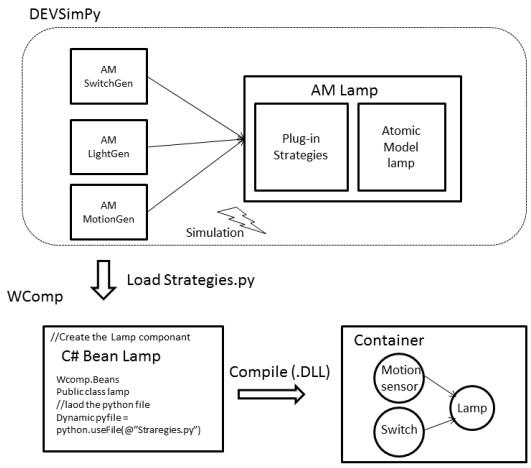

Figure 11: Integration of strategies validated with DEVSimPy into WComp.

For that, the class Bean of the component Lamp has been created in WComp and the references (line 1-2) have been added as illustrated below in order to insert Python statements into C# code.

```
1 using IronPython.Hosting;
2 using IronPython.Runtime;
3 using Microsoft.Scripting.Hosting;
4 using Microsoft.CSharp;
```

As illustrated by the code below, the IronPython runtime (line 2), the Dynamic Type and the strategy Python file (line 6) have been created. Depending on the content of the Strategy.py file, a function can be called (Strategy1 in line 8) in order to invoke the selected strategy that return a new state.

```
1 public void Strategie1(string val) {
2  ScriptRuntime py = Python.CreateRuntime();
3
4  if (PropertyChaned != null) {
5  // Create a Dynamic Type for Python File
6   dynamic pyf = py.UseFile("Strategy.py");
7
8   PropertyChanged(pyf.Strategie1(val))
9  }
10 }
```

After the compilation of the Bean class, the corresponding binary file (dll) is inserted in the resulting assembly to be interconnected with the other components.

5 CONCLUSION

This paper deals with an approach for the design and the implementation of IoT ambient systems based on discrete-event M&S. A new approach based on DEVS simulations is proposed: instead of waiting the implementation phase of an IoT ambient system to detect eventual specification problem, we describe an initial phase consisting in DEVS M&S of the behavior of components involved in an ambient system as well as the behavior of different strategies corresponding to different behaviors of execution machines. Once the DEVS simulations have brought successful results, the Designer can implement the behavior of the given ambient system using an IoT framework such as WComp. The code corresponding to selected strategy can be fully inserted in the WComp environment in such a way that the Designer has no code to write in the WComp environment when implementing the selected strategy.

The presented approach has been validated on a test-case example which is described in detail in the paper: DEVS implementation of four different strategies which can be used for a given ambient system, definition of the corresponding DEVS specification, implementation of the DEVS behavior using the DEVSimPy framework, analysis of the simulation results.

Our future work will consist in two main directions: (1) we have to work on the Design of more complex IoT systems using DEVS formalism and DEVSimPy framework; (2) we also plan to use the DEVSimPy framework in order to manage discrete-event simulations obtained from DEVS models associated with connected objects such as board computers, sensors, controllers or actuators (the interest is to strongly associate simulations and connected objects). The result will be the ability to manage connected objects (sensors, computer boards, actuators, controller) from the DEVSimPy framework while providing intelligent decisions based on simulations.

From a M&S aspects the following tasks are performed (Sehili et al., 2015):

- Development of a DEVSimPy Phidget components library for manipulating Phidget board computers, sensors and actuators
- Implementation of a Web Server in order to run the simulations involving components of the DEVSimPy Phidgets Library which allows to manage connected objects and provision of web services allowing to dynamically interact from the mobile application with the DEVSimPy framework which is installed on the Web Server

REFERENCES

Capocchi, L., Santucci, J. F., Poggi, B., and Nicolai, C. (2011). DEVSimPy: A Collaborative Python Software for Modeling and Simulation of DEVS Systems. In *WETICE*, pages 170–175. IEEE Computer Society. URL: http://code.google.com/p/devsimpy/ [Retrieved: Dec 2014].

Chow, A. C. H. and Zeigler, B. P. (1994). Parallel DEVS: A parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th Conference on Winter Simulation*, WSC '94, pages 716–722, San Diego, CA, USA. Society for Computer Simulation International.

Hourdin, V., Ferry, N., Tigli, J.-Y., Lavirotte, S., and Rey, G. (2013). Middleware in Ubiquitous Computing. pages 71–88.

Jones, E., Oliphant, T., and Peterson, P. (2001). Scipy: Open source scientific tools for python. URL: http://www.scipy.org/Citing_SciPy [Retrieved: February, 2014].

Li, X., Vangheluwe, H., Lei, Y., Song, H., and Wang, W. (2011). A testing framework for devs formalism implementations. In *Proceedings on the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, TMS-DEVS '11, pages 183–188, San Diego, CA, USA. Society for Computer Simulation International.

Lopes, J. a., Souza, R., and Geyer, C. (2014). A middleware architecture for dynamic adaptation in ubiquitous computing. *Journal of Universal Computer Science*, 20:1357–1351.

Oliphant, T. E. (2007). Python for scientific computing. *Computing in Science and Engineering*, 9:10–20.

Perez, F., Granger, B. E., and Hunter, J. D. Python: An ecosystem for scientific computing. *Computing in Science and Engineering*, (2):13–21.

Rappin, N. and Dunn, R. (2006). *WxPython in action*. Manning.

Sehili, S., Capocchi, L., and Santucci, J.-F. (2014). Iot component design and implementation using devs simulations. The Sixth International Conference on Advances in System Simulation SIMUL 2014.

Sehili, S., Capocchi, L., and Santucci, J.-F. (2015). Management of ubiquitous systems with a mobile application using discrete event simulations (WIP). London, UK. Accepted in ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS).

Seung, W. H., Yeo, B. Y., and Hee, Y. Y. (2004). A new middleware architecture for ubiquitous computing environment. IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems.

Zeigler, B. P. (2003). An introduction to set theory. Technical report, ACIMS Laboratory, University of Arizona. URL: http://www.acims.arizona.edu/EDUCATION/ [Retrieved: April, 2014].

Zeigler, B. P., Praehofer, H., and Kim, T. G. (2000). *Theory of Modeling and Simulation, Second Edition*. Academic Press.