Design and Implementation of Ambient Intelligent Systems using Discrete Event Simulations

Souhila Sehili
University of Corsica
SPE UMR CNRS 6134
Corte, France
sehili@univ-corse.fr

Laurent Capocchi
University of Corsica
SPE UMR CNRS 6134
Corte, France
capocchi@univ-corse.fr

Jean-François Santucci
University of Corsica
SPE UMR CNRS 6134
Corte, France
santucci@univ-corse.fr

***Abstract*—The Internet of Things (IoT) project enables rapid innovation in the area of Internet connected devices and associated cloud services. An IoT node can be defined as a flexible platform for interacting with real world objects and making data about those objects accessible through the Internet. Communication between nodes is discrete Event-oriented and the simulation process play an important role in defining assembly of nodes in such ambient systems. One of today's challenges in the framework of ubiquitous computing concerns the design of such ambient systems. The main problem is to propose a management adapted to the composition of applications in ubiquitous computing. In this paper, we propose the definition of a modeling and simulation scheme based on a discrete-event 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 that can be implemented in the execution machine. A pedagogical example concerning a concurrent access to a switchable on/off light has been modeled into the Python DEVSimPy environment in order to validate our approach.**

Keywords–IoT; Discrete-event; Simulation; Formalism; Assembly; Smart; Environment.

I. INTRODUCTION

Technological advances in recent years around mobile communication and miniaturization of computer hardware have led to the emergence of ubiquitous computing. In our previous paper [1] we have presented how the DEVS formalism can be used in order simulate the behavior of ambient intelligent 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. The word "ubiquitous computing"was first used in 1988 by Mark Weiser to describe his vision of future [2] - computing at the twenty-first century as he had imagined. In his idea, computing tools are embedded in objects of everyday life. The objects are used both at work and at home. The user has at its disposal a range of small computing devices such as smartphone or PDA, and their use is part of ordinary daily life. These devices make access to information easier for everyone, anywhere and anytime. Users then have the opportunity to exchange data easily, quickly and effortlessly, regardless of their geographic position. The definition of such complex systems involving sensors, smartphones, interconnected objects, computers, etc. results in what is called ambient systems.

One of today's challenges in the framework of ubiquitous computing [3] 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 [4]. Ambient systems applications design involves the management of many varied devices integrated in objects of everyday life. The unpredictability of availability of the features of these devices makes the need for explicit adaptation for this type of system. The specificity of this adaptation is that it will meet all the constraints imposed by the context of ambient 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. Mechanism to address this concern must then be proposed by middleware for ubiquitous computing.

We have being focused on the WComp environment, which is a prototyping and dynamic execution environment for Ambient Intelligence applications. WComp [5] is 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 devices, like UPnP [6] or DPWS services [7], discovered in the software infrastructure. In the framework of the WComp, it has been defined a management mechanism allowing extensible interference between devices. This is particularly important in the context definition of new coordination logic. In WComp it has been proposed a methodology for interference management mechanism to be dynamically and automatically extensible. In order to deal with the asynchronous nature of the real world, WComp has defined an execution machine for complex connections. In a real case, the assumption of zero reaction time is not realistic. It is essential to check that the system is fast enough according to the dynamics of the environment. It is also essential to make the link between the logical time and physical time and the relationship between the actual events of the environment and those used in the definition of synchronous processes [8]. The entity that is responsible for ensuring these approximations is the execution machine and is used to treat the interface between synchronous and asynchronous process environments [9].

In this paper, we propose the definition a modeling and simulation 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. The interest of such an approach is twofold: (i) the behavior will be used to write the methods required; (ii) to check the different strategies (to be implemented in the execution machine) before implementation. The rest of the paper is as follows: Section II concerns with the background of the study by presenting the traditional approach for the design of IoT systems. It briefly introduces a set of middleware framework before focusing on the WComp Framework. The DEVS formalism and the DEVSimPy environment are also presented. In Section III, the proposed approach based on the DEVS formalism is given. An overview of the approach as well as the interest in using DEVS simulation is detailed. Section IV deals with the validation of the approach through a case study The conclusion and future work are given in Section V.

II. RELATED WORK

There have been a some approaches dedicated to the management of ubiquitous systems. In this section, we highlight several kinds of middleware tools have been proposed in the recent years such as:

Roman et al. [10] proposed a middleware software infrastructure Gaia, which assist humans in the development of applications for ubiquitous computing buildings and homes intelligent by interacting with devices simultaneously.

Seung et al. [11] proposed a new approach in middleware architecture HOMEROS, which adopts a hybrid-network model to efficiently manage enormous resources, context, location, allowing high flexibility in the environment of heterogeneous devices and users.

Lopes et al. [12] proposed a middleware software infrastructure EXEHDA, which manages and implements the follow-me semantics in which the applications code is installed on-demand on the devices and this installation is adaptive to context of each device.

Ferry et al. [13] proposed a middleware WComp based on a software infrastructure, a service composition architecture, and a compositional adaptation mechanism used in prototyping and executing the Ambient Intelligence applications.

III. BACKGROUND

A. IoT Design and WComp

The ubiquitous computing is a new form of computing that has inspired many works in various fields such as the embedded system, wireless communication, etc. Embedded systems offer computerized systems having sizes smaller and integrated into objects everyday life. An ambient system [14] is a set of physical devices that interact with each other (e.g., a temperature sensor, a connecting lamp, etc.). The design of an ambient system should be based on a software infrastructure and any application to be executed in such an ambient environment must respect the constraints imposed by this software infrastructure.

Devices and software entities provided by the manufacturers are not provided to be changed: they are black boxes. This concept can limit the interactions to use the services they provide and prevents direct access to their implementation. The creation of an ambient system can not under any circumstances pass by a modification of the internal behavior of these entities but simply facilitate the principle reusability, since an entity chooses for its functionality and not its implementation. In the vision of ubiquitous computing, users and devices operate in an environment variable and potentially unpredictable in which the entities involved appear and conveniently disappear (a consequence of mobility, disconnections, breakdowns, etc.). It is not possible to anticipate the application design when we do not have information about availability of any devices. As a result a set of tools have been dedicated in developing software infrastructure allowing the design of applications with the constraint unpredictability availability of component entities [15].

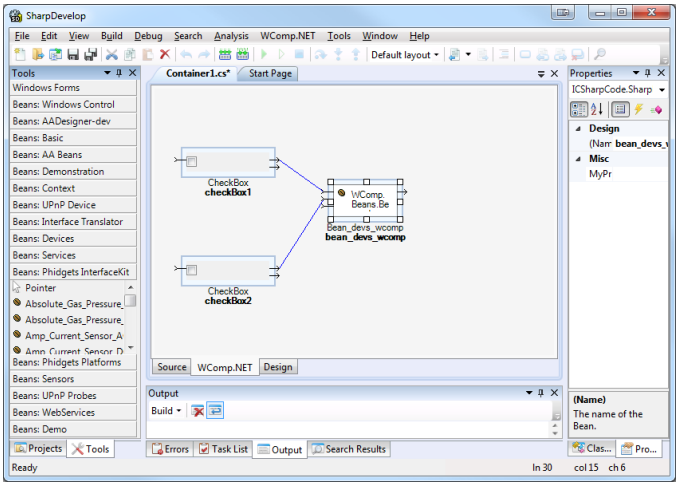


Figure 1. WComp platform.

In this paper, we deal with the WComp framework, which is used in order to design ambient systems. The WComp architecture is organized around containers and designers [16] (Figure 1). The purpose of containers is to take over the management of the dynamic structure such as instantiation, destruction of components and connections.

The Designer runs the Container for instantiation and for the removal of components or connections between components in the Assembly, which has to be created. A component belonging to the WComp platform is an instance of the Bean class implemented in a hight level object language [17] to use properties at runtime and to calibrate some variables to refine the interaction.

An application is created by a WComp component assembly in a container, according to SLCA model [18]. WComp allows to implement an application from an orchestration of services available in the platform and/or other off-the-shelves components.

Whatever the tool that may be used, the design of a IoT component leans on the definition of:

- A set of methods allowing to describe the behaviors of the component
- The execution machine associated with the considered component

The design of ambient computing systems involves a different technique from those used in conventional computing. Applications are designed dynamically by *smart*

devices (assembly components) of different nature.

The smart device is an identified component, which is generalized as a class of objects defining a data as property and containing distinct logic sequences that can manipulate it, known as methods that are executed when the component receives an event from others components. The manner of executing these methods (*state automaton* [19]) depending on some inputs is called the *execution machine* (Figure 2).

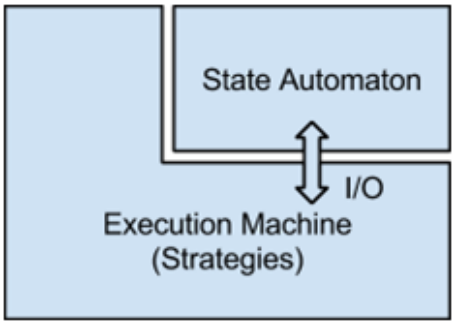


Figure 2. Component state automaton with execution machine.

The construction of an ambient system requires the definition of:

- The state automaton (methods)
- The execution machine

Several ways to manage the execution machine are known as strategies; the description of the strategies are defined manually in the methods of the Bean class (object oriented class) of WComp framework. Figure 3 describes the traditional way to design an ambient system using WComp. The behavior and the components involved in the ambient system, as well as the Bean classes describing the execution machine, are coded using the C# language.

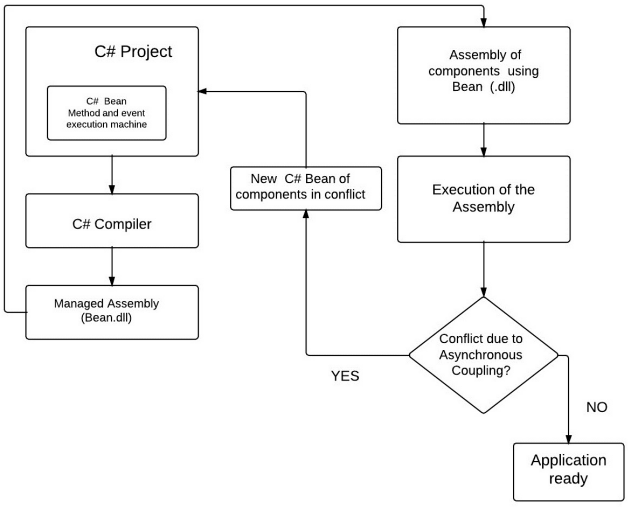


Figure 3. Traditional IoT component design with WComp.

The compilation allows to derive the corresponding dynamic assembling binary files (.dll) of the Bean classes involved in the resulting Assembly [20]. The Assembly can then be executed. Conflicts are checked: if conflicts (generally due to asynchronous couplings) are detected the designer has to write a new behavior of the execution machine by recoding the Bean classes in order to solve the coupling conflicts while if no conflict are detected the application is ready. In this paper, we choose to propose a new approach for a computer aided design of ambient systems using the DEVS formalism by developing DEVS simulation concepts and tools for the WComp platform. The goal is to use the DEVS formalism and the DEVSimPy framework in order to perform DEVS modeling and simulations: (i) to detect the potential conflicts without waiting to implementation and execution phases as in the traditional approach of Figure 3; (ii) to offer the designer to choose between different executions strategies and to test them using DEVs simulations; (iii) to propose a way to automatically generate the coded of the methods involved in the execution machine strategies. The DEVS formalism and the DEVSimPy environment are briefly introduced in the next two sub-sections while the proposed approach is introduced in Section III.

B. The DEVS formalism

Since the seventies, some formal works have been directed in order to develop the theoretical basements for the modeling and simulation of dynamical discrete event systems [21]. DEVS (Discrete EVent system Specification) [22], [23] 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.

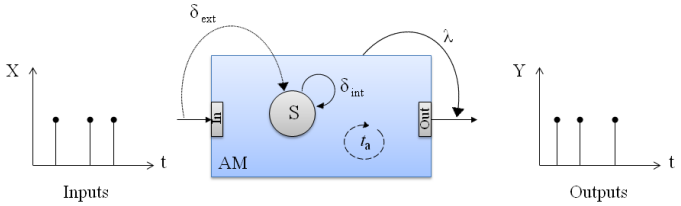


Figure 4. Atomic model in action.

A DEVS atomic model AM (Figure 4) with the behavior is represented by the following structure:

$$AM =< X, Y, S, \delta_{int}, \delta_{ext}, \lambda, t_a >$$

where:

- $X : \{(p, v)|(p \in inputports,\ v \in X_p^h)\}$ is the set of input ports and values,

- $Y : \{(p, v)|(p \in outputports,\ v \in Y_p^h)\}$ is the set of output ports and values,
- S: is the set of states,
- $\delta_{int} : S \rightarrow S$ is the internal transition function that will move the system to the next state after the time returned by the time advance function,
- $\delta_{ext} : Q \times X \rightarrow S$ is the external transition function that will schedule the states changes in reaction to an external input event,
- $\lambda : S \rightarrow Y$ is the output function that will generate external events just before the internal transition takes places,
- $t_a : S \rightarrow R_{\infty}^{+}$ is the time advance function that will give the life time of the current state.

The dynamic interpretation is the following:

- $Q = \{(s, e)|s \in S^h, 0 < e < t_a(s)\}$ is the total state set,
- e is the elapsed time since last transition, and s the partial set of states for the duration of $t_a(s)$ if no external event occur,
- δ_{int} : the model being in a state s at t_i , it will go into s' , $s' = \delta_{int}(s)$, if no external events occurs before $t_i + t_a(s)$,
- δ_{ext} : when an external event occurs, the model being in the state s since the elapsed time e goes in s', The next state depends on the elapsed time in the present state. At every state change, e is reset to 0.
- λ : the output function is executed before an internal transition, before emitting an output event the model remains in a transient state.
- A state with an infinite life time is a passive state *(steady state),* else, it is an active state *(transient state)*. If the state s is passive, the model can evolve only with an input event occurrence.

The DEVS coupled model CM is a structure:

$$CM =< X, Y, D, \{M_d \in D\}, EIC, EOC, IC >$$

where:

- X is the set of input ports for the reception of external events,
- Y is the set of output ports for the emission of external events,
- D is the set of components (coupled or basic models),
- M_d is the DEVS model for each $d \in D$,
- EIC is the set of input links that connects the inputs of the coupled model to one or more of the inputs of the components that it contains,
- EOC is the set of output links that connects the outputs of one or more of the contained components to the output of the coupled model,
- IC is the set of internal links that connects the output ports of the components to the input ports of the components in the coupled models.

In a coupled model, an output port from a model $M_d \in D$ can be connected to the input of another $M_d \in D$ but cannot be connected directly to itself.

The DEVS abstract simulator is derived directly from the model. A simulator is associated with each atomic model and a coordinator is associated with each coupled model. In this approach, simulators allows to control the behavior of each model, and coordinators allows the global synchronization between each of them. The communication between all these elements is performed using four kinds of messages. The initialization messages (i, t) are used to achieve an initial temporal synchronization between all actors. The internal transition messages $(*, t)$ allow the processing of an internal event, while the external transition messages (x, t) allow the processing of an external event. Finally, the output messages (y, t) allow the transportation of the output values to the parent elements and is the result of an $(*, t)$ message.

C. The DEVSimPy environment

DEVSimPy [24] (DEVS Simulator in Python language) is an open Source project (under GPL V.3 license) supported by the SPE team of the university of Corsica Pasquale Paoli. This aim is to provide a GUI for the modeling and simulation of PyDEVS [25] models. PyDEVS is an Application Programming Interface (API) allowing the implementation of the DEVS formalism in Python language. Python is known as an interpreted, very high-level, object-oriented programming language widely used to quickly implement algorithms without focusing on the code debugging [26]. The DEVSimPy environment has been developed in Python with the wxPython [27] graphical library without strong dependences other than the Scipy [28] and the Numpy [29] scientific python libraries. The basic idea behind DEVSimPy is to wrap the PyDEVS API with a GUI allowing significant simplification of handling PyDEVS models (like the coupling between models or their storage).

Figure 5 depicts the general interface of the DEVSimPy environment. A left panel (bag 1 in Figure 5) shows the libraries of DEVSimPy models. The user can instantiate the models by using a drag-and-drop functionality. The bag 2 in Figure 5 shows the modeling part based on a canvas with interconnection of instantiated models. This canvas is a diagram of atomic or coupled DEVS models waiting to be simulate.

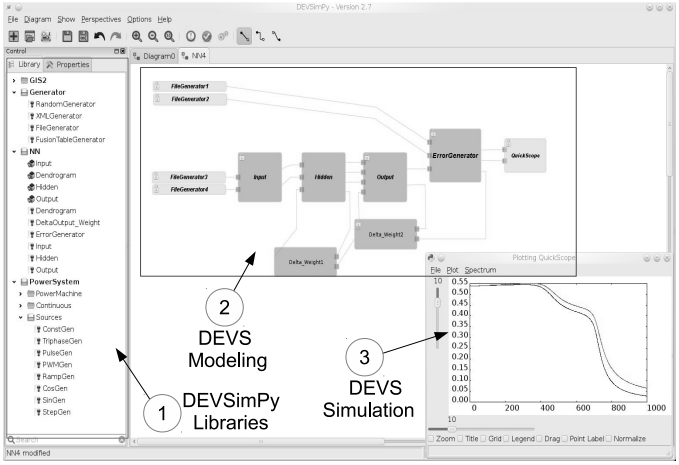


Figure 5. DEVSimPy general interface.

A DEVSimPy model can be stored locally in the hard disk or in cloud through the web in the form of a compressed file including the behavior and the graphical view of the model

separately. The behavior of the model can be extended using specific plug-ins embedded in the DEVSimPy compressed file. This functionality is powerful since it makes it possible to implement new algorithms above the DEVS code of models in order to extend their handling in DEVSimPy (exploit behavioral attributes, overriding of DEVS methods, etc.). A plug-in can also be global in order to manage several models through an generic interface embedded in DEVSimPy. In this case, the general plug-in can be enabled/disabled for a family of selected models. An interesting global plug-in called Blink has been implemented to facilitate the debugging in DEVSimPy. This plug-in is based on successive steps of the simulation and blink the models to indicate their activity with a color code corresponding to the nature of the DEVS transition function (internal, external, time advance, output).

DEVSimPy capitalizes on the intrinsic qualities of DEVS formalism to simulate automatically the models. Simulation is carried out in pressing a simple button, which invokes an error checker before the building of the simulation tree. The simulation algorithm can be selected among hierarchical simulator (default with the DEVS formalism) or direct coupling simulator (most efficient when the model is composed with DEVS coupled models). A plug-in manager is proposed in order to expand the properties of DEVSimPy allowing their enabling/disabling through a dialog window. For example, a plug-in called ”Blink” is proposed to visualize the activity of models during the simulation. It is based on a step by step approach and illuminates each active model with a color, which depends on the executed transition function. In this paper, a plug-in is used to allow the transposition of the execution machine strategies validated with DEVS simulation to WComp environment.

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

IV. PROPOSED APPROACH

As pointed in Section II-A, the traditional way to design ambient systems described in Figure 3 has the following drawback: the creation of Bean class components using the WComp Platform is performed by the definition of methods (both implementing the behavior of a device and its execution machine) in the object oriented language C#. The compilation allows to obtain a set of library components, which are used in a given Assembly (which corresponds to the designed ambient system). However, eventual conflicts due to the connections involved by the Assembly can be detected only after execution. This means that the Designer has to modify the execution machine of some components and restart the design at the beginning. We propose a quite different way to proceed, which is described in Figure 6.

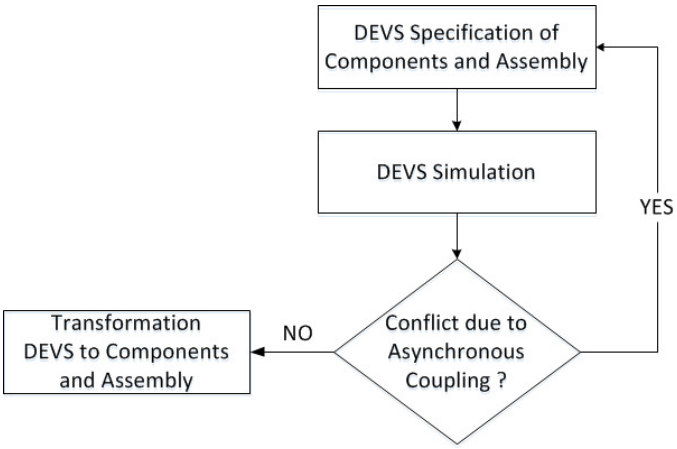


Figure 6. IoT component design using DEVS.

The idea is to use the DEVS formalism in order to help the Designer to:

- Validate different strategies for execution machines involved in an Assembly.
- Write the methods corresponding to the strategy of the execution machine he wants to implement.

For that the Designer has first to write the specifications the components as well as the coupling involved in an Assembly (corresponding to an ambient system to implement). Then simulations can be performed. According to the results of the simulation, conflicts can be highlighted: if some conflicts exist the DEVS specifications have to be modified if not the design process goes on with C# implementation as in Figure 3. The DEVS specifications can be used to help the Designer to write the methods of the Bean classes in the C# language Figure 6 and then compile them and execute the resulting Assembly being assured that there will be no coupling conflict. Section IV details the proposed approach using a pedagogical example. Two different execution machine strategies will be implemented using WComp and using the DEVS formalism. We will point out how DEVS can be used to simulate execution machines strategies before compilation and execution of the C Bean classes. Furthermore, we also point out how the designer can use the DEVS specifications in order to write the methods involved in an execution machine strategy.

V. CASE STUDY: SWITCHABLE ON/OFF LIGHT

A. Description

We choose to validate the proposed approach on a pedagogical case study: realization of an application to control the lighting in a room. The case study involved three components to be assembled: a light component with an input (ON / OFF) and two switches components with an output (ON / OFF) as shown in Figure 7.

Two different behaviors concerning the connections between the switch and the light component are envisioned (corresponding to the implementation of two different execution machines):

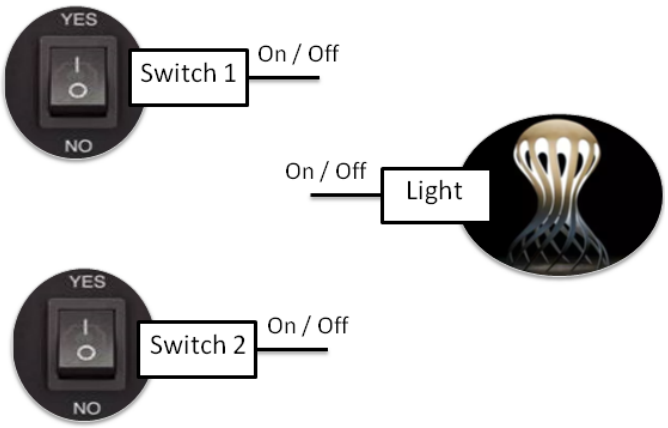


Figure 7. Assembly of Light and Switches components.

- First behavior: the light is controlled by toggle switches, which rest in any of their positions
- Second behavior: the light is controlled by push button switches, which have two-position devices actuated with a button that is pressed and released

In this part, we will present first how we have implemented these two previous behaviors using the WComp platform. Then we will give the DEVS approach involving the DEVS specifications of the two behaviors of the case study and the way DEVS can be used for WComp design of the ambient components.

B. WComp implementation

The behaviors corresponding to the toggle switch and push button switch have been implemented using two different Bean classes in WComp platform in order to be assembled separately with a light component.

The Bean class (Figure 8) in WComp platform is a self-contained class enabling the reuse of the component and facilitate the sharing of it component by other systems. This class is introduced in a specific category of the graphic interface (Container WComp) and the references (#Category in Figure 8) are added in the class. The implementation of the Bean class requires the definition of the name of the Bean class, which is the name of the component in the Designer (#Bean Name in Figure 8). The Properties of the Bean class contain the setter and getter of the class attributes. The Methods implement the behaviors of the component (#Propriety, #Methods in Figure 8) and the EventHandler activate the methods when events are emitted.

Looking at the structure of the Bean class we identify the part that involves a set of actions to follow in a given situation (Methods). These actions define the behaviours of the component that have been identified by the programmer early in the design process.

To illustrate this point, we choose to clarify the observed behaviors by implementing them in the methods of different classes.

```
using System;
using WComp.Beans;
[Bean(Category=" ")]                                   #Category

public class MyBean                                    #Bean Name

public int Myprop {                                    #Propriety
        get {
            return myprop; }
        set {
            myprop = value; }

public void MyMethod() {                               #Methods
        # A method sending an event

        }

public delegate void stringValueEventHandler(string val);

public event stringValueEventHandler PropertyChanged;  #Events

private void MyEvent(int i) {
        if (PropertyChanged != null)
            PropertyChanged(i);
}
```

Figure 8. Bean class structure in WComp.

1) First behavior implementation: corresponding to the toggle switch described in the Figure 9, the line 2 is used to check the position of the toggle switch: if ON is true the line 3 ensures that there are subscribers before calling the event PropertyChanged. In the lines 4 and 5 the event is raised and a resulting string is transmitted. The Bean class returns the String once The ControlMethod method is invoked.

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

Figure 9. First light method implementation in WComp.

2) Second behavior implementation: Corresponding to the push button switch described in the Figure 10. The initialization of the lightstate variable of component Light is performed through line 1. Line 3 allows to switch the value of the lightstate variable while line 4 allows to initialize the message to be returned. Line 5 is dedicated to check the lightstate variable and to eventually change to returned message. Lines 6 and 7 allow to ensure that there are subscribers before calling the event Property-Changed and transmit the returned message.

The compilation step is performed for each Bean class. The compiler produces modules that are the traditional executable files (DLL) reusable and manipulated in the WComp platform. After this process each bean class is instantiated and connected with two check-box representing the respective switches in order to realize the required assembly in the WComp platform (Figure 11).

```
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);
```

Figure 10. Second light method implementation in WComp.

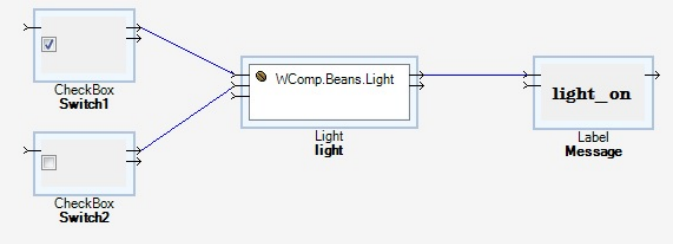


Figure 11. WComp assembly components.

```
Import os
Import platform
Import DomainInterface                          #Specific import

class MyComponent                               #DEVS Class

Def __init__():                                 #Initialization

Def extTransition(self):                        # DEVS external transition function
    pass

Def outFonction(self):                          # DEVS output function
    pass

def intTransition(self):                        # DEVS internal transition function
    pass

def timeAdvance(self):                          # DEVS Time Advance function
    return self.state['sigma']
```

Figure 13. Aotmic Model structure in DEVSimPy.

C. DEVS Specifications

In order to highlight the interest of the DEVS formalism in the management of conflicts between the interconnected components in WComp platform, we defined a DEVS atomic model for each component in DEVSimPy Framework. The behaviors of the light component are implemented in the atomic model Light (Figure 12). The assembly is a DSP diagram (DSP stands for DEVSimPy) and is easy to reuse in DEVSimPy.

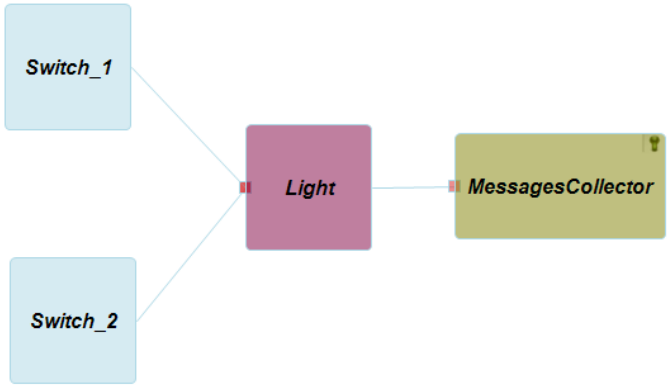


Figure 12. Object interaction diagram for the light component.

Figure 13 depicts the template of an atomic model class in Python language into DEVSimPy.

The implementation of this class needs some specific imports when the model inherits another module or library (#Specific import in Figure 13). The class has a constructor (__init __ ()) with a particular attribute "self.state" that allow to define the state variables (#Initialization in Figure 13). The transition functions like δ_{int} and δ_{ext} are implemented through intTransition(self) and extTransition(self) methods (#DEVS external transition function and #DEVS internal transition function in Figure 13). The output function λ is implemented in the outFunction(self) method and the time advance function t_a in timeAdvance(self) method.

In the structure of the atomic model class, the different actions related to the component behaviors are defined in the external transition that we have chosen to clarify below in the two cases defined in the Section IV-B.

The specifications of the behaviors are achieved using finite-state automaton (Figures 14 and 16) that allows to specify the component behaviors formally [30] and facilitate the deployment in DEVSimPy as an atomic model.

1) The toggle switch behavior: in the transition graph "automaton" given in Figure 14, each state is represented by a pair (state/output). This means that the states are "state1 and state2" and the associated output are "Set_On and Set_Off". The input value is given by the transition between one state and the next state. The system can remain in the same state (loop) as stationary state.

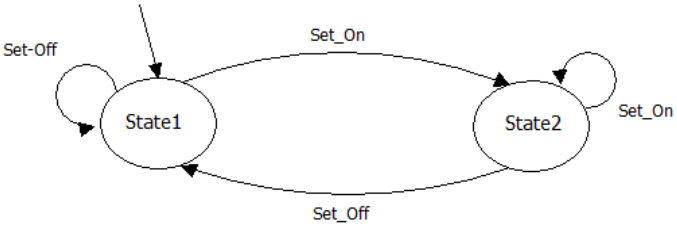


Figure 14. Automaton of the toggle switch behavior.

The corresponding DEVSimPy implementation of the automaton is given in Figure 15 and expressed through the external transition of the atomic model *Light* (Figure 12).

```
1     self.intstate= "OFF"
2     def extTransition(self):
3     for i in xrange(len(self.IPorts)):
4       msg=self.peek(self.IPorts[i])
5       if msg :
6         self.result[i]=msg.value[1]
7       if self.result[i]==self.intstate :
8           self.finstate=self.intstate
9           else:
10          self.finstate=self.result[i]
11            self.state['sigma']=0
```

Figure 15. External transition function of the light atomic model.

The initialization of the state variable *instate* is done in line 1 (initial value is OFF). Line 3 and line 4 allow to assign the variable *msg* with the value of the events on the input ports. From line 5 to line 10 the code allows to assign the value of the state variable *instate* according to the value of the variable *msg*: if the message on the port is equal to the initial state then the state variable remains on the same state else the value of the *instate* variable is changed. By setting the variable *sigma* to 0, line 11 allows to activate the output function.

2) The push-button switch behavior: the transition graph "automaton" given in Figure 16 is a different one that in Figure 14, where the system cannot remain in the same state for each input. The system move to the other state.

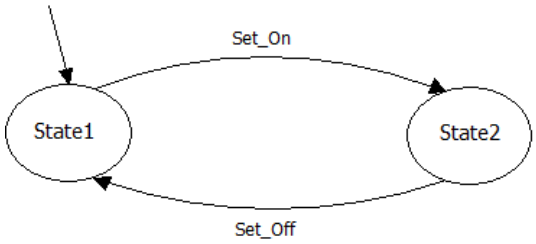


Figure 16. Automaton of the push-button switch behavior.

The corresponding DEVSimPy implementation of the automaton is given in Figure 17 and expressed through the external transition of the atomic model *Light* (Figure 12).

```
1  def extTransition(self):
2      for i in xrange(len(self.IPorts)):
3        msg=self.peek(self.IPorts[i])
4  if msg :
5    self.result[i]=msg.value[1]
6    if self.intstate == "ON":
7    self.finstate= "OFF"
8    else:
9    self.finstate="ON"
10   self.intstate=self.finstate
```

Figure 17. External transition function of the atomic model Light.

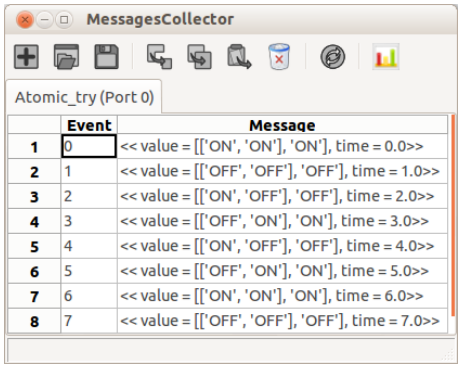
Line 2 and line 3 allow to assign the variable msg with the value of the events on the input ports. From line 4 to line 10, the code allows to switch the value of the state variable *intstate* and *finstate* from ON to OFF or OFF to ON according to the values of the input ports.

D. Simulation results

In both cases, once the modeling scheme has be realized using the DEVSimPy environment, we are able to perform simulations that correspond to the behavior of the ambient system according to the two different execution machines that have been defined. The simulation results obtained with DEVSimPy are illustrated in a *MessageCollector* model, which is often used to store messages received during the simulation. The *MessageCollector* model organizes its results in a table (see Figure 18 and Figure 19).

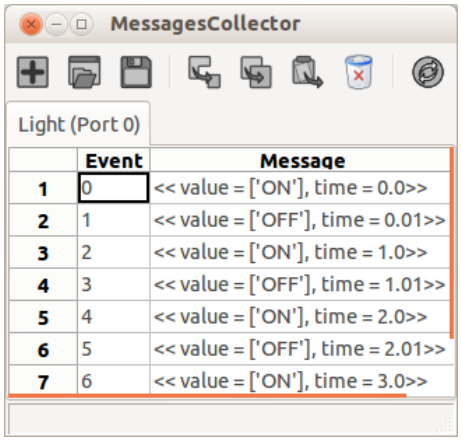
In Figure 18, we show several lines which highlight the result of events from two toggle switches, in the first line we describe the position of toggle switches ['ON', 'ON'] the resulting event is the Lamp 'ON'. The simulation results of the first case express the fact that the execution machine allows the ambient system under study remains in the initial position ("ON" or "OFF") until we will actuate another position using one of the switches.

In Figure 19, we show a several lines which highlight the result of events from two push button switches. The simulation results of the second case express the fact that the execution machine allows the ambient system under study to alternately "ON" and "OFF" with every push of one of the switches.



	Event	Message
1	0	<< value = [['ON', 'ON'], 'ON'], time = 0.0>>
2	1	<< value = [['OFF', 'OFF'], 'OFF'], time = 1.0>>
3	2	<< value = [['ON', 'OFF'], 'OFF'], time = 2.0>>
4	3	<< value = [['OFF', 'ON'], 'ON'], time = 3.0>>
5	4	<< value = [['ON', 'OFF'], 'OFF'], time = 4.0>>
6	5	<< value = [['OFF', 'ON'], 'ON'], time = 5.0>>
7	6	<< value = [['ON', 'ON'], 'ON'], time = 6.0>>
8	7	<< value = [['OFF', 'OFF'], 'OFF'], time = 7.0>>

Figure 18. First simulation results captured with *MessagCollector*.



	Event	Message
1	0	<< value = ['ON'], time = 0.0>>
2	1	<< value = ['OFF'], time = 0.01>>
3	2	<< value = ['ON'], time = 1.0>>
4	3	<< value = ['OFF'], time = 1.01>>
5	4	<< value = ['ON'], time = 2.0>>
6	5	<< value = ['OFF'], time = 2.01>>
7	6	<< value = ['ON'], time = 3.0>>

Figure 19. Second simulation results captured with *MessageCollector*.

E. Integration of DEVS implementation in WComp

As depicted in Figure 20, the integration of strategies in WComp starts by defining the DEVS atomic model (AM) corresponding to the component in which strategies are identified (using functions) in DEVSimPy environment (*Light* in the case study).

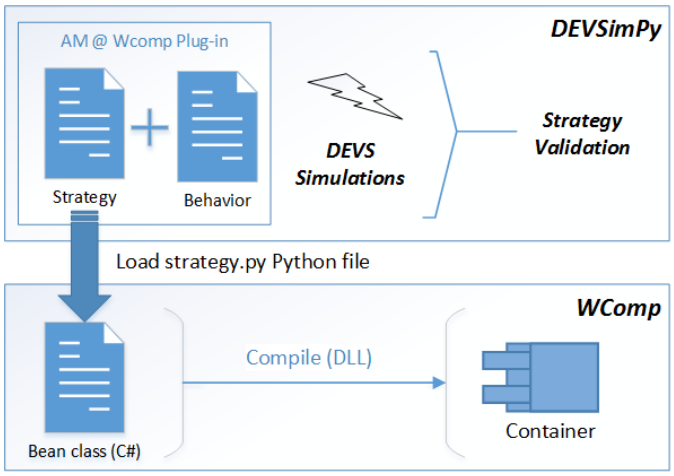


Figure 20. DEVSimPy/WComp integration process.

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 21).

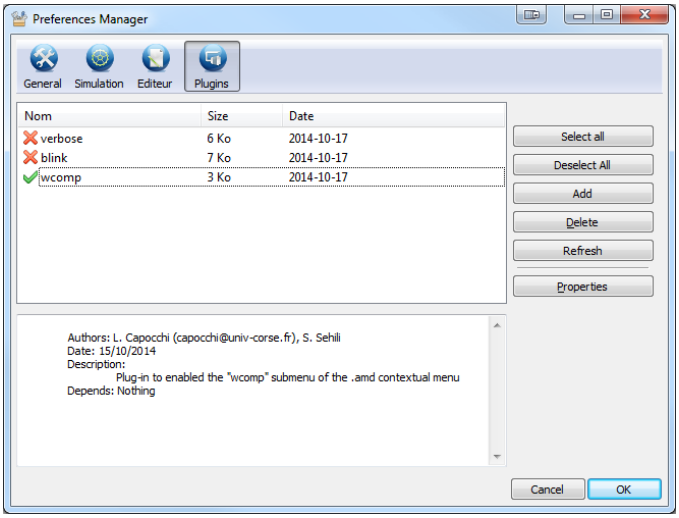


Figure 21. General plug-in *WComp* in DEVSimpy.

Once simulations are performed and strategies are validated in the DEVSimPy framework, we load the strategies file (strategy.py in Figure 20) that contains strategies into WComp. This is done to through IronPython [31], which is an implementation of Python for .NET allowing us to leverage the .NET framework using Python syntax and coding styles.

For that, the class Bean of the component *Light* has been created in WComp and the references (line 1-2) have been added as illustrated in Figure 22 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;
```

Figure 22. C# importing to use Python functions.

As illustrated in Figure 23, the IronPython runtime (line 1) and the Dynamic Type (line 2) have been created and the strategy Python file (line 3) has been loaded.

```
1 ScriptRuntime python=Python.CreateRuntime();
2 dynamic pyfile = python.UseFile(@"Path");
3 String string = pyfile.Strategies();
```

Figure 23. C# code to insert Python *Strategies* function.

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

F. Interest of the approach

As described in Sections IV-C and IV-D, the proposed approach allows to study the behavior of an ambient system using DEVS simulations before any WComp implementation. This will allow a Designer of ambient system to select the desired execution machine that adapts to the context of use before the design phase of the component under WComp platform to reduce the time and implementation cost.

In Section IV-C, we briefly introduce how the DEVS specifications can be used by an ambient system Designer to write the code of execution machine. From the two previous cases defined in Section IV-C, we can note that the method of the Bean class under WComp platform of a given ambient component present some similarities with the external transition of the corresponding DEVS atomic model of the same component (in the one part, see Figure 9 and Figure 15, and on the other part Figure 10 and Figure 17).

Furthermore, in Section IV-E, we performed simulations of strategies defined in the local plug-in of the atomic model of the component in DEVSimPy that are loaded in WComp framework through an implementation of python for .NET (IronPython) in the Bean class. This will allow us to validate and implement all the components, which WComp platform reuse them directly.

VI. Conclusion and Future Works

This paper deals with an approach for the design and the implementation of IoT ambient systems based on Discrete Event Modelling and Simulation. The traditional way leans on: (i) the definition of the behavior of IoT components in a Library; (ii) the design of the coupling of components belonging to the Library; (iii) the execution of the resulting coupling. If some errors are detected, the designer has to redefine the behavior of the components (especially by redefining the behavior of the execution machine, which allows to

describe the behavior of the ambient system in case of time conflicts).

This paper introduces a new approach based on DEVS simulations: instead of waiting the implementation phase to detect eventual conflicts, we propose an initial phase consisting in DEVS modeling and simulation of the behavior of components involved in an ambient system, as well as the behavior 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 presented approach has been applied on a pedagogical example that is described in detail in the paper: implementation of two different behaviors of a given ambient system, definition of the corresponding DEVS specification, implementation of the DEVS behavior using the DEVSimPy framework, analysis of the simulation results. Furthermore, we have also pointed out that the DEVS specifications can be used in order to help the Designer to write the behavior of the IoT components.

Our future work will consist in two main directions. Firstly, we have to work on the Design of complex IoT systems using DEVS formalism and DEVSimPy framework. Secondly, we have to propose an approach allowing to automatically write the behavior of the execution machines after their validation based on DEVS simulation. This automatic generation of the behavior will be performed from the DEVS external state transition function coding and will consist in generating the corresponding execution machine code (for example C# code in the case of the WComp framework).

References

[1] S. Sehili, L. Capocchi, and J.-F. Santucci, "Iot component design and implementation using devs simulations," in The Sixth International Conference on Advances in System Simulation (SIMUL), 2014, pp. 71–76.

[2] M. Weiser, "The computer for the 21st century," SIGMOBILE Mob. Comput. Commun. Rev., vol. 3, no. 3, Jul. 1999, pp. 3–11. [Online]. Available: http://doi.acm.org/10.1145/329124.329126

[3] M. Satyanarayanan, "Pervasive computing: Vision and challenges," IEEE Personal Communications, vol. 8, 2001, pp. 10–17.

[4] M. Zhao, G. Privat, E. Rutten, and H. Alla, "Discrete control for the internet of things and smart environments," in Presented as part of the 8th International Workshop on Feedback Computing. Berkeley, CA: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/feedbackcomputing13/workshop-program/presentation/Zhao

[5] V. Hourdin, N. Ferry, J.-Y. Tigli, S. Lavirotte, and G. Rey, "Middleware in ubiquitous computing," Computer Science and Ambient Intelligence, 2013, pp. 71–88.

[6] M. Jeronimo and J. Weast, "Upnp design by example: A software developer's guide to universal plug and play," in Intel Press, 2003.

[7] S. Unger, E. Zeeb, F. Golatowski, D. Timmermann, and H. Grandy, "Extending the devices profile for web services for secure mobile device communication," in Presented as part of the 8th International Workshop on Feedback Computing, 2013.

[8] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," Proceedings of the IEEE, vol. 79, no. 9, 1991, pp. 1270–1282.

[9] S. Schewe and B. Finkbeiner, "Synthesis of asynchronous systems," in 16th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR 2006). Springer Verlag, 2006, pp. 127–142.

[10] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, "Gaia: A middleware platform for active spaces," SIGMOBILE Mob. Comput. Commun. Rev., vol. 6, no. 4, Oct. 2002, pp. 65–67. [Online]. Available: http://doi.acm.org/10.1145/643550.643558

[11] S. W. Han, Y. B. Yoon, H. Y. Youn, and W.-D. Cho, "A new middleware architecture for ubiquitous computing environment," in Software Technologies for Future Embedded and Ubiquitous Systems, 2004. Proceedings. Second IEEE Workshop on. IEEE, 2004, pp. 117–121.

[12] J. Lopes, R. Souza, C. Geyer, C. Costa, J. Barbosa, A. Pernas, and A. Yamin, "A middleware architecture for dynamic adaptation in ubiquitous computing," Journal of Universal Computer Science, vol. 20, no. 9, 2014, pp. 1327–1351.

[13] N. Ferry, V. Hourdin, S. Lavirotte, G. Rey, M. Riveill, and J.-Y. Tigli, "WComp, a Middleware for Ubiquitous Computing", ser. . InTech, Feb. 2011, ch. 8, pp. 151–176. [Online]. Available: http://www.intechopen.com/articles/show/title/wcomp-a-middleware-for-ubiquitous-computing

[14] Y. Liu, "Design of the smart home based on embedded system," in Computer-Aided Industrial Design and Conceptual Design, 2006. CAIDCD'06. 7th International Conference on. IEEE, 2006, pp. 1–3.

[15] D. Cheung-Foo-Wo, "Adaptation dynamique par tissage d'aspects d'assemblage," Ph.D. dissertation, Université de Nice Sophia Antipolis, 2009.

[16] V. Monfort and F. Felhi, "Context aware management plateform to invoke remote or local e learning services: Application to navigation and fishing simulator," in Ambient Intelligence and Future Trends-International Symposium on Ambient Intelligence (ISAmI 2010). Springer, 2010, pp. 157–165.

[17] G. Gauffre, S. Charfi, C. Bortolaso, C. Bach, and E. Dubois, "Developing mixed interactive systems: A model-based process for generating and managing design solutions," in The Engineering of Mixed Reality Systems. Springer, 2010, pp. 183–208.

[18] V. Hourdin, J.-Y. Tigli, S. Lavirotte, G. Rey, and M. Riveill, "Slca, composite services for ubiquitous computing," in Proceedings of the International Conference on Mobile Technology, Applications, and Systems. ACM, 2008, p. 11.

[19] S. Eugene Xavier, "Theory of automata formal languages and computation," in The Engineering of Mixed Reality Systems. New Age International (P) Ltd, 2005, ISBN: 978-81-224-2334-1.

[20] V. Monfort and S. Cherif, "Bridging the gap between technical heterogeneity of context-aware platforms: Experimenting a service based connectivity between adaptable android, wcomp and openorb," in IJCSI International Journal of Computer Science Issues, vol. 8, no. 3. IJCSI, May 2011, ISBN: 978-81-224-2334-1.

[21] B. P. Zeigler, "An introduction to set theory,," ACIMS Laboratory, University of Arizona, Tech. Rep., 2003, URL: http://www.acims.arizona.edu/EDUCATION/ [Retrieved: April, 2014].

[22] B. P. Zeigler, H. Praehofer, and T. G. Kim, Theory of Modeling and Simulation, Second Edition,. Academic Press, 2000.

[23] B. Zeigler and H. Sarjoughian, "System entity structure basics," in Guide to Modeling and Simulation of Systems of Systems, ser. Simulation Foundations, Methods and Applications. Springer London, 2013, pp. 27–37. [Online]. Available: http://dx.doi.org/10.1007/978-0-85729-865-2_3

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

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

[26] F. Perez, B. E. Granger, and J. D. Hunter, "Python: An ecosystem for scientific computing," Computing in Science and Engineering, vol. 13, no. 2, 2011, pp. 13–21, URL: http://dblp.uni-trier.de/db/journals/cse/cse13.html#PerezGH11 [Retrieved: February, 2014].

[27] N. Rappin and R. Dunn, "Wxpython in action." Greenwich, Conn: Manning, 2006.

[28] E. Jones, T. Oliphant, P. Peterson et al., "SciPy: Open source scientific tools for Python," 2001, [accessed 2015-05-26]. [Online]. Available: http://www.scipy.org/

[29] T. E. Oliphant, “Python for scientific computing,” Computing in Science and Engineering, vol. 9, no. 3, May/June 2007.

[30] N. Belloir, J.-M. Bruel, and F. Barbier, “Intégration du test dans les composants logiciels,” in Workshop OCM dans lingnierie des SI during INFORSID, 2002.

[31] M. Foord and C. Muirhead, “Ironpython in action,” in Manning Publications Co., 2009.