Formal Abstract Modeling of Dynamic Multiplex Networks

Cristina Ruiz-Martin
Carleton University /
Universidad de Valladolid
Ottawa, ON K1S 5B6 Canada
cruiz@eii.uva.es

Gabriel Wainer
Carleton University
Ottawa, ON K1S 5B6 Canada
gwainer@sce.careleton.ca

Adolfo Lopez-Paredes
Universidad de Valladolid
Valladolid, Castilla y León 47250 Spain
aparedes@eii.uva.es

ABSTRACT

We describe an Abstract Model for Diffusion Processes to simulate diffusion processes in multiplex dynamic networks using formal modeling and simulation (M&S) methodologies (in this case, the DEVS formalism). This approach helps the users to implement diffusion processes over a network by using the network specification and the diffusion rules. The result of combining the network specifications and the diffusion rules is an Abstract Model for Diffusion Processes, which is formally defined in DEVS, and can be converted into a computerized model. Using the proposed Abstract Model for Diffusion Processes, we can study a diffusion process in multiplex networks with a formal simulation algorithm, improving the model's definition. We present a case study using the CDBoost simulation engine.

KEYWORDS

Diffusion Processes, Multiplex Networks, DEVS, Formal Modeling and Simulation

ACM Reference format:

Cristina Ruiz-Martin, Gabriel Wainer, and Adolfo Lopez-Paredes. 2018. Formal Abstract Modeling of Dynamic Multiplex Networks. In Proceedings of SIGSIM Principles of Advanced Discrete Simulation (SIGSIM-PADS' 18). ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3200921.3200922

1 INTRODUCTION

A diffusion process is a phenomenon where an element is spread from a place with high concentration to a place where the concentration is low. The study of such phenomena has been useful in various domains [1]: Medicine (e.g. studying of spreading of disease over a population), Management (e.g. analyzing how a new idea or change is accepted in a company), Social Science (e.g. checking the effect of an information disseminated in a Social Network in the behavior of the people), etc.

SIGSIM-PADS '18 , May 23–25, 2018, Rome, Italy

ACM ISBN 978-1-4503-5092-1/18/05...$15.00
https://doi.org/10.1145/3200921.3200922

Diffusion processes have been studied building a multiplex network (i.e. a set of interconnected entities with different types of relations) that defines the relations between the entities involved in the process and defining the rules the diffusion process follows [2]. Then, these specifications are translated into a computer program that is simulated to generate results. Unfortunately, most of the existing research about simulation of diffusion processes in multiplex or multiplex dynamic networks does not provide insight on the M&S methodology and platforms they use. For example, Xiong et al [3] run a numerical simulation to study the effect of the diffusion of innovation in social networks but no information on the simulation aspects is presented. Khelil et al [4] use their own simulator written in Java to implement a model to study information dissemination strategies in mobile ad hoc networks, but no information is provided about the simulation details. Numerous cases are similar to these.

Not only the simulation aspects are neglected: the model definition is normally informal, and difficult to analyze. To overcome the lack of formalization, and the details of implementation, we here propose a formal Abstract Model for Diffusion Processes (from here on, DAM, *Diffusion Abstract Model*) that allows us to define and implement diffusion processes in multiplex dynamic networks by using the definition of the network and the diffusion rules for the process. This model is an integral component of a formal architecture defined in [5] that allows defining and studying diffusion processes in multiplex networks.

Having the DAM formally defined allows one having the diffusion model and its implementation separated, therefore model verification and validation are simplified and development time reduced. One can think about the formal model prior to implementation, analyze different experiments that could be conducted on the model, and finding complex errors before spending valuable time in coding. In our case, we use Discrete Events Systems specifications (DEVS) [6] since diffusion process in multiplex networks can be modeled using a discrete event approach and DEVS is well suited. It provides a formal framework to study hierarchical modular models, which is well adapted for modeling diffusion models. Since the models, simulators, and experiments are independent, the same model can be implemented on different platforms and the verification process can be improved. As the simulation algorithms for DEVS have been formally defined, this helps in achieving separation of concerns and building complex applications that can be verified with ease, focusing only on the modeling aspects. To simulate the DEVS models, we use the CDBoost framework. CDBoost is a cross-platform DEVS simulator implemented in C++11 that allows a direct conversion of

DEVS functions into C++ code. This is the formal methodology and the simulation tools used to define the DAM, can be used to achieve repeatable simulation studies.

Some of the advantages of this new model architecture are as follows:

- the model can be defined using formal modeling and simulation (M&S) methodologies

- we can study dynamic networks, and defined a mechanism for storing the time when the change occurs

- we can update the properties at runtime.

- we defined an XML specification for the behavior of the diffusion model, which allows us to define complex rules. This allows us to remove software dependencies.

The rest of the paper is organized as follows. In section 2, we discuss related work on diffusion processes and we briefly explain DEVS and CDBoost. In section 3, we present the DAM and the architecture where it is integrated. In section 4, we explain a general implementation of the DAM using DEVS. In section 5, we present some simulation results of the application of the DAM to study an information diffusion process inside an organization. Finally, in section 6, we present the conclusions of this work.

2 BACKGROUND

Many diffusion processes have been specified using differential equations or other types of rules such as if-then rules. For example, many diffusion processes in medicine have been studied using Susceptible-Infected-Recovered (SIR) models, which are formally defined using differential equations. (e.g. [7]).

In fact, much of the research work on diffusion processes is based on the definition of new algorithms invented in the field of Medicine. There are algorithms to study preventive measures to protect the population against a disease [8], [9], to study the propagation of specific diseases such as dementia [10], etc. These algorithms, although developed for medical applications, have sometimes been applied to study problems in other fields, such as communications in mobile networks [4] or the diffusion of information and opinion adoption in social networks [11]–[13].

However, as mentioned in the introduction, the diffusion algorithms are normally converted into ad-hoc computer programs that include the network. There is no formal definition of the model. Likewise, the M&S methodology used or the simulation platform where the model is implemented are not detailed.

Generalizing diffusion processes from simplex to multiplex networks is not simple. Although there have been some advances in this area, it is still an open research field [14]. For example, several diffusion processes (e.g. linear diffusion, random walks, etc.) have already been generalized [2], [15]. In [16], the authors proposed a match between the elements of diffusion processes in social networks and the concepts used in Agent Based Modeling (ABM) techniques. They also proposed to use ABM to study the diffusion problem in social networks as a method to obtain empirical results and to connect theoretical and empirical research.

Some recent research has focused on formalizing the study of diffusion processes in multiplex networks [17], [18] using ABM, Network Theory and DEVS. The authors presented an architecture to simulate information diffusion processes in multiplex dynamic networks. They defined a model using a server-proxy architecture where the servers represent the behavior of the nodes in the network model (i.e. the rules to transmit and assimilate the information), and the proxies define the diffusion rules for each type of link (i.e. the different types of connections between nodes) in the network model. Both servers and proxies are modeled using DEVS, and they are coupled to represent network nodes. The connections between the nodes in the networks are used to build the DEVS Top Model that represents multiplex networks. In order to be able to model dynamic networks (changes on the number of nodes or the connections between them), the authors store all possible network configurations and they use Dynamic DEVS [19], and a database to store all the network configurations and the properties that define the behavior of the nodes.

In [20], the authors adapted the above-mentioned architecture to study business processes in the healthcare sector. They modified the architecture to include Business Process Model and Notation (BPMN) in order to study the impact of dynamic allocation of patients in the healthcare pathway.

Following the research line presented in [17], [18], we introduce a new architecture where the model (DAM) is abstract and generic and can be instantiated to simulate any kind of diffusion process in multiplex networks.

We used the DEVS formalism [6] as the formal basis to develop the DAM, as DEVS provides a framework to develop hierarchical models in a modular way, allowing model reuse and thus, reducing development time and testing. There are different DEVS simulators such as JAMES [21], JDEVS [22], DEVSJava [23], CDBoost [24], etc. We used CDBoost, a fast cross-platform DEVS simulator implemented in C++11. CDBoost provides simple interfaces to the modeler, who can transform a DEVS model to a DEVS simulation. At the user level, it allows defining atomic and coupled models, and since the output format of the simulation is flexible (i.e. the user can configure it), it can be defined in such way that helps the analysis of the results. Figures 4 and 5 show the CDBoost simulator definition to implement DEVS models. Figure 1 shows a template to implement an atomic model, and figure 2 a template to define coupled models.

```
1 struct AtomicName_defs{ //Input&Output Port declaration
2    struct input_port1 : public in_port< MSGi1> {};
3    struct input_portn : public in_port< MSGin> {};
4    struct output_port1 : public out_port< MSGo1> {};
5    struct output_portn : public out_port< MSGon> {};   };
6
7 template<typename TIME>
8 class AtomicName{
9  using defs=AtomicName_defs;//port definition in context
10  public:
11  struct state_type{ //Define your state variables here   };
12  state_type state;
13  AtomicName() noexcept {//parameters/initial state values}
14
15 //DEVS functions
16  void  internal_transition()  {//Define  internal  transition
17      function             }
18 void external_transition(TIME e, typename make_message_bags
```

```
19      <input_ports>::type mbs) {
20  //Define your external function here                }
21     void     confluence_transition(TIME     e,     typename
22          make_message_bags <input_ports>::type mbs) {
23  // confluence function here                  }
24   typename   make_message_bags<output_ports>::type   output()
25      const {  // Output function
26    typename make_message_bags<output_ports>::type bags;
27    //Define your output function here. Fill bags
28    return bags;                                  }
29 TIME time_advance() const {
30      //Define the time advance function              }
31 };
```

Figure 1. DEVS atomic model implementation in CDBoost.

As seen in the figure, first, we declare the model ports as a structure (lines 1-5) and the atomic model as a class (lines 7-31). Each atomic model class has the set of state variables grouped together in a structure (lines 11). It also has a model constructor to instantiate the model parameters and initial values (line 13). We implement all the DEVS functions (internal, external, confluence, output and time advance, in lines 15-31) in C++. The code in bold cannot be modified (it is part of the simulator).

The coupled models are implemented using the template provided in figure 2. We instantiate all the atomic models with their parameters (lines 1-6) and then we define the coupled models (including the top model).

```
1 //*****INSTANTIATE ATOMICS *******//
2 template<typename TIME>
3 class iestream_int : public iestream_input<int,TIME> {
4 public:
5 iestream_int(): iestream_input<int,TIME>
6 ("inputs/test_filterNetworks.txt") {};   };
7 //*****DEFINE COUPLED *******//
8 struct inp_in_1 : public in_port<int>{};
9 struct outp_out_2 : public out_port<double>{};
10 using iports_C1 = std::tuple< inp_in_1 >;
11 using oports_C1 = std::tuple< outp_out_2 >;
12 using submodels_C1=models_tuple<filterNet, iestream_int> ;
13 using eics_C1=tuple<EIC
14      <inp_in_1,iestream_int, iestream_defs::in> >;
15 using eocs_C1 =tuple< EOC
16     < filterNet, filterNet _defs::out, outp_out_2> >;
17 using ics_C1=tuple<IC
18     <iestream_int,iestream_defs::out,
19          filterNet, fiterNet _defs::in> >;
20
21 using C1=coupled_model <TIME,iports_C1,oports_C1,
22          submodels_C1,eics_C1,eocs_C1,ics_C1>;
23
24 int main(){ //Call the simulator
25   runner<NameOfTimeClass, NameOfTopModel, logger_top> r{0};
26   r.runUntil(300000); }
```

Figure 2. DEVS coupled and top model implementation in CDBoost.

Figure 2 is an implementation of the coupled model shown in figure 3. We first declare the coupled model ports (lines 8-9). We then define the top model components: input ports (line 10), output ports (line 11), submodels (line 12), external input couplings (line 13-14), external output couplings (line 15-16) and internal couplings (line 17-19). The coupled model (line 21-22) is a tuple of all these components. The last step is to call the simulator (lines 24-26). We set the name of the time class and the top model name (line 25), and simulation running time (line 26).

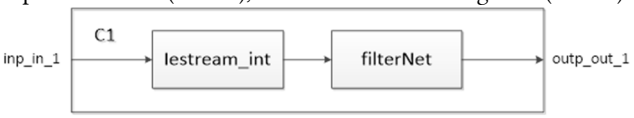


Figure 3. Example of DEVS coupled model defined in fig. 2

3 ABSTRACT MODEL FOR DIFFUSION PROCESSES DEFINITION

To model and simulate diffusion processes in multiplex dynamic networks, we proposed the architecture presented in figure 4 [5], whose main component is the diffusion abstract model.

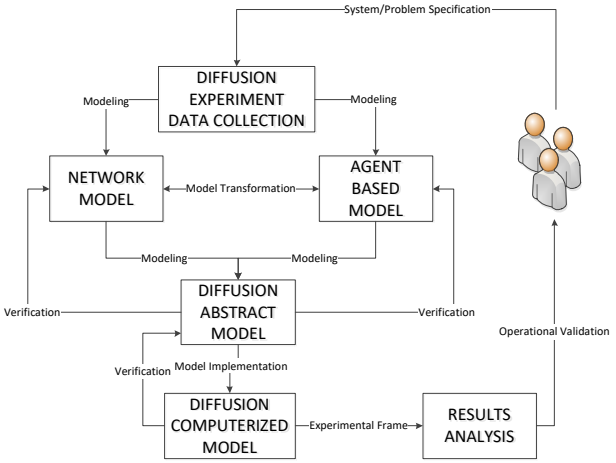


Figure 4: An architecture to simulate diffusion processes in multiplex dynamic networks

The architecture includes six components as follows:

a) Diffusion experiment data collection: we need to obtain all the requirements, specifications, and data available for the problem under study. This information can be gathered manually (e.g., through interviews or text analysis), or automatically (e.g. using different types of sensors). In general, the diffusion experiment data is provided by experts in the domain. If the information is incomplete, the domain experts should provide additional data, instructions about how to collect it, or, if not available, approve a set of assumptions.

b) Network model: it is an organized representation of the *Diffusion experiment data* using Network Theory. It provides a formal representation of the relations among the components of the system. Although this is a formal model, there are different tools like Gephi [26], Pajek [27], MuxViz [28], etc. that allows the model to be defined using a graphical interface, and it allows storing the network model in various formats (tables, graphs, XML). The model is built following the *Diffusion experiment data* document.

c) Agent-Based model: it is a representation of the behavior of those in charge of the diffusion process, the objects they use

for diffusing the element, and the properties of the relationships among these objects. It is defined using ABM techniques, and it can be implemented using different methods: specific software platforms such as NetLogo, Repast [29], using an XML definition, or a formal specification like a DEVS model.

d) DAM: the DAM, the main object of this research, is an abstract and formal representation of the *Diffusion experiment data* that matches elements in both the *Network* and the *Agent-based* models. It is a formal specification (in our case, we use DEVS, but it could be defined using other formalisms, like System Dynamics, State charts, etc.). One could also define the DAM combining different formalisms, as long as there is a way to connect them (for instance, a metamodel).

e) Diffusion Computerized Model (DCM): it is a computer implementation of the *DAM*. It can be built using different simulators; in our case, we used a DEVS simulator called CDBoost [30]. Once all the components of the *DAM* are implemented, the top-level model can be defined either manually or automatically by processing the *Network* and *Agent-based* models.

f) Results Analysis: the results provided by the *DCM* can be analyzed using different methods, statistical and data visualization tools (such as R [31], PowerBI [32], etc.).

The DAM is the central part of the architecture, which we will describe in detail. It is defined a generic container that follows the structure presented in figure 5.

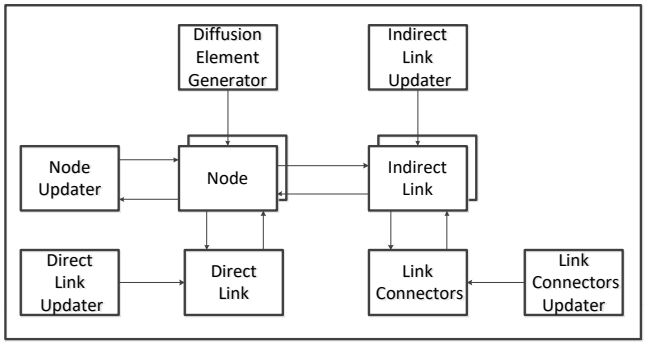


Figure 5. DAM structure

It includes nine components.

1. Node: it is a representation of a vertex in the Network model, including all its input and output connections. It also matches an Agent in the in the Agent-Based model, since the Agents are the nodes in the Network model. Here is where we define the behavior of the diffusion process on each node.

2. Indirect Link: it represents the properties of the input and output connections of a node in the Network model. It also matches the objects used by the agents to carry the diffusion process. Each of them is different. Once all the objects used by the agents to carry the diffusion process have been modeled, each *Indirect Link* will contain a different subset of them based on input and output links of the node.

3. Direct Link: it represents direct connections between *Node* models. It represents the properties of the links in the *Network* model that have a direct connection between nodes. In the *Agent-based* model, it represents the connections handled without using any additional objects. It can be defined as an atomic or coupled model based on the complexity and the level of detail needed for diffusion rules on direct connections. For example, in the case of information transmission face-to-face, it can be as simple as an atomic model that transmits the message after a delay or it can be a coupled model that combines the environmental noise and the distance between the people and introduce a disturbance in the contentment of the message.

4. Link Connectors: this single model represents how the objects used by the diffusion agents are connected. It does not have a direct match to the *Network* model. In the *Agent-based* model, it represents the properties of the relations among the *Indirect Links* and how they are connected. Link Connectors are similar to Direct Links, but they represent the rules the diffusion process follows between indirect links. For instance, in a diffusion process for communication, if indirect links represent messages through Facebook or text messaging, the Link Connectors model can be defined as a coupled model with two components: *Internet* and *Mobile Network*, with a switch to direct the messages to the appropriate network.

5. Diffusion Element Generator: it generates elements to be diffused over time. It defines the initial location of the diffusion elements in the *Network* and *Agent-based* models, and the new ones introduced over time.

6. Updaters: they modify the properties of the models at runtime. They allow us to model dynamic *Networks* where not only change the connections but also the nodes and links. With the updaters, we can modify the properties of the *Indirect Link*, *Link Connectors*, *Node* and *Direct Link* models without modifying any model in the structure.

As we have already mentioned, each of these components could be modeled and implemented using different formal methods. In this paper, we will show its definition using DEVS.

The *Indirect Link Updater*, *Link Connectors Updater*, *Direct Link Updater* and *Diffusion Element Generator* are defined as four different instances of a DEVS atomic model that is parameterized to generate updates in the properties of the other components. This is done using the information in an external file. It can also include the element to be diffused in the model. In order to instantiate these models we use any C++ type (i.e. *int*, *pair*, *struct* etc.) that matches the information stored in the external file.

The *Node Updater* can be either an atomic or a coupled model (depending on the complexity of the rules to update the properties of the node). It updates the properties of the *Nodes* like the other updaters, but in this case, the properties can be updated using both information stored in an external file and data included in the model (e.g. the properties of other nodes)

Node and *Indirect Link* are DEVS coupled models that have several filters and atomic or coupled models that represent the behavior of the diffusion process. The atomic and coupled models inside *Node* and *Indirect Links* are parameterized based on the specifications provided in the Network and the Agent-based models. We broadcast the diffusion messages, and filter the ones that should be assigned to each component. The components of the model identify if the message (i.e. a diffusion element or a property update) is for them.

Figure 6 represents the structure of the *Indirect Link* Coupled. It has one filter for each input port and an atomic or coupled model for each type of link on the *Indirect Link*. It also contains a *Sink* (added for validation purposes; if a message arrives at the *Sink*, it means that the model received a message for a link not included in the model; this means there is an implementation error or a mismatch in the model definition).

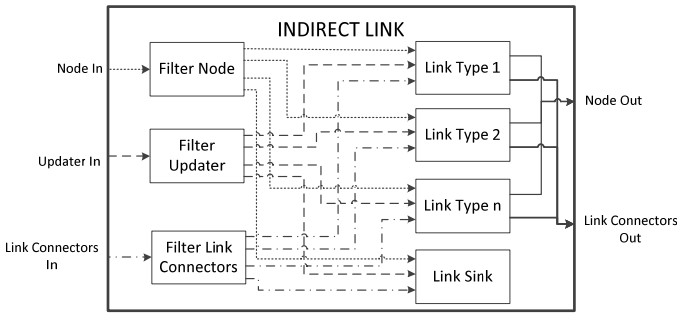


Figure 6. Indirect Link coupled model

All these models (both atomic and coupled) have been defined formally using DEVS. In Appendix I we show the formal definition of one of these DEVS atomic models: the *Switch*, used to redirect the messages that come from the *Indirect Links* to the corresponding model inside the *Node*. The implementation of this model and its behavior is explained in section 4, figure 7. Likewise, in Appendix II we show the formal definition of a DEVS coupled model: *Indirect Links* (figure 6).

All the models have been defined using the same process. As we can see, the formal definition of the model contributes to model validation. For example, in the *Switch* (see Appendix I), we may forget to passivate the model when *messagesPassing* is empty and we can find such errors by just looking at the formal definition. We can also find varied errors in the model definition; for instance, if when there is an input in *setAnswerIn* we set the state variable *state* to SEND, instead of ANSWER, we can detect that by simply checking the formal model (prior to any implementation or simulation). In the *Indirect Links* coupled model (see Appendix II), we know that the type of message in *NodeIn* port is *DiffusionElements*. When we connect *Indirect Links* to *Node* model, we know that the type of message in the port *IndirectLinksOut* of *Node* model must be *DiffusionElements*. Otherwise, the model is not valid. We can find these errors in the model formal specifications with ease, without wasting valuable time in building a computer model that is wrong, in the early phases of the model, which will save efforts in the latter phases.

4 DAM IMPLEMENTATION USING DEVS[1]

As discussed earlier, once all the models have been formally defined and its formal behavior studied, we to translate them into computerized DEVS models using the CDBoost simulator. The parameterized DEVS atomic models, introduced in section 3, are implemented using the template provided in figure 1.

Figure 7 shows the implementation of the *switch* atomic model that is used to redirect the messages that comes from the *Indirect Links* to the corresponding model inside the *Node*. We chose this example as it allows showing some of the implementation details and the DEVS functions are simple (the rest of the atomic models are built following a similar logic: they are formally specified using DEVS, and implemented in CDBoost).

```
1 struct switch3Out_defs{   //Port definition
2 struct sendOut,answerOut,decideOut: public out_port<MSG> {};
3 struct diffusionElementIn : public in_port<MSG> {};
4 struct setAnswerIn,setDecideIn: public
5       in_port<SET_STATE_ANS> {};
6 struct setSendIn : public in_port<SET_STATE_SEND> {};
7 };
8 class switch3Out {  // DEVS atomic model definition
9  public:
10  DeviceType id; //Parameter
11 enum SwitchState{ANSWER,SEND,DECIDE};//state definition
12 state_type state;
13 struct state_type{
14   vector <MSG> outMsg;
15   SwitchState  state;  };
16..
17 void internal_transition() { state.outMsg.clear();}
18
19 void external_transition(TIME e, typename
20       make_message_bags<input_ports>::type mbs)
21  if(!get_messages<typename defs::setAnswerIn>(mbs).empty())
22   state.state = SwitchState::ANSWER;
23  if(!get_messages<typename defs::setSendIn>(mbs).empty())
24   state.state = SwitchState::SEND;
25  if(!get_messages<typename defs::setDecideIn>(mbs).empty())
26    state.state = SwitchState::DECIDE;
27  for (const auto &x : get_messages<typename
28       defs:: diffusionElementIn >(mbs)) {
29   if(x.to.type == id) state.outMsg.push_back(x); }       }
30
31 typename make_message_bags<output_ports>::type output()
32        const { // output function
33  typename make_message_bags<output_ports>::type bags;
34    switch(state.state){
35     case SwitchState::ANSWER:
36      for (int i = 0; i < (state.outMsg.size()); i++)
37       get_messages<typename defs::answerOut>(bags).
38         push_back(state.outMsg[i]);
39     case SwitchState::SEND:
40      for (int i = 0; i < (state.outMsg.size()); i++)
41       get_messages<typename defs::sendOut>(bags).
42         push_back(state.outMsg[i]);
43     case SwitchState::DECIDE:
44      for (int i = 0; i < (state.outMsg.size()); i++)
45       get_messages<typename defs::decideOut>(bags).
46          push_back(state.outMsg[i])
47      }
48  return bags;       }
49
50 TIME time_advance() const { // time_advance function
51  return (state.outMsg.empty() ? infinity() : TIME());
52  }
53 };
```

Figure 7. Computer Model of the Switch atomic model

[1] The documentation is available at:
https://github.com/SimulationEverywhere/NEP_DAM.git

The model redirects the message in the *DecideIn* port to the appropriate output port based on the model state (answer, decide, send). We use different output ports according to the model's state. The model follows the template presented in Section 2. The external transition function (lines 19-29) stores a messages received through *diffusionElementIn* port in *OutMsg* variable. It also sets the value of the *state* variable based on the inputs in the other ports *SetDecideIn, SetSendIn* and *SetAnswerIn*. When the time is consumed, we activate the output function (lines 31-48), which sends the messages stored in the *OutMs*g variable through the output port that the state variable determines. Then, we execute the internal transition function (line 17), which clears the *OutMsg* variable. Finally, the time advance function (lines 50-52) passivates the model if there is nothing to send, and triggers an instantaneous event (time advance 0) is there is something to send.

We store the parameters that define the behavior of each model in XML files. If the behavior of all nodes is the same, we will only have one XML with the parameters for that behavior (for example, if we study the diffusion of a virus over a population, where the individuals are infected with the same probability). If we need more complex behaviors, we can define it in different XML files (for instance, if different groups of individuals react differently to the virus, we might have different XML files to define the behavior of the nodes; for instance, the individuals that do not are infected because they are immune, the ones that are infected with low probability because they are vaccinated and the ones that are infected with high probability). An extreme example is where all the nodes have different behavior. In this case, we have as many XML files as nodes. This case is the one presented later on to study the information dissemination inside an organization.

We parse these XML files to instantiate the parameterized DEVS models in CDBoost and build the DAM automatically. The model parameters are dependent on the application, however, the process and the general implementation can be adapted easily. Here we show how to build the DAM using as a case study a diffusion process inside an organization.

Figure 8 presents an example of an XML file where we define the behavior of a person. It represents how the person behaves in terms of information transmission. In our model, some characteristics are attributes (i.e. they are fully specified in the diffusion model and therefore, they remain constant) and other are parameters (i.e. they are not completely defined in the model due to lack of information, or we want to change them to study their effect on the process). We use XML tags to define each of the parameters and attributes (and their values are defined as the content of the tags).

```
1 <?xml version="1.0" ?>
2 <AgentBehavior>
3   <Id>First Responder 20</Id>
4  <Location>55D6</Location>
5  <ReactionTime>00:00:10:000</ReactionTime>
6  <AnswerPriorityType>DEVICE_PRIORITY </AnswerPriorityType>
7  <SendPriorityType> PRIORITY_LIST </SendPriorityType>
8  <MyDevices>
9   <PriorityDevice priority="1" device="MOBILEPHONE"
10     send2Multiple="false" sendSeparateFromReceive="false"/>
11     ...
12 </MyDevices>
13 <SortedTasks>
14     <PriorityTask priority="1" task="ANSWER"/>
15     <PriorityTask priority="2" task="SEND"/>
16  ...
17 </SortedTasks>
18 <AnswerDevicePriority>
19     <PriorityDevice priority="1" device="RADIO" />
20  ...
21 </AnswerDevicePriority>
22 <AnswerPersonPriority>
23     <PriorityPerson priority="1" id="1"/>
24 ...
25     <PriorityPerson priority="3" id="97"/>
26 </AnswerPersonPriority>
27 <SendCommandPriority>
28     <PriorityCommandTo priority="1" to="1" msg="
29      Tell population to stay at home" />
30        ...
31     <PriorityCommandTo priority="3" to="97" msg="
32        Tell population to stay at home" />
33 </SendCommandPriority>
34 <ActionExecutionPriority>
35 <PriorityAction priority="1" id="Tell population to stay
36     at home"/>
37 </ActionExecutionPriority>
38 <CommunicationRelations>
39 <RelationPerson id="1">
40     <Device device="RADIO"/>
41     <Device device="MOBILEPHONE"/>
42 </RelationPerson>
43 <RelationPerson id="5">
44     <Device device="BEEPER"/>
45 </RelationPerson >
46     ...
47 </CommunicationRelations>
48 <MessageBehavior>
49 <MsgReceived from="1" content="Tell people to stay at
50     home">
51     <Msg2Send to="5" content="Tell people to stay at home
52  acknowledgement" cmpulsory="true" send2Multiple="false"/>
53     <Action2Do id=" Tell population to stay at home "/>
54 </MsgReceived>
55 </MessageBehavior >
56 <ActionBehavior>
57 <Action id="Tell population to stay at home">
58     <AverageExecutionTime time="00:10:00:000"/>
59     <Location>55D6</Location>
60     <Msg2Send to="1" content=" Tell population to stay at
61       home completed" compulsory="true" 57bsend2Multiple =
62        "false"> <Device device="BEEPER"/></Msg2Send>
63 </Action>
64 </ActionBehavior>
65</AgentBehavior>
```

Figure 8. XML definition a person's behavior

The behavior of node (i.e. person) is defined between the tags <*AgentBehavior*>. *Id* is an attribute that identifies the person, based on the organizational structure. *Location* is a dynamic attribute that represents the location of the person. *Reaction Time* is a parameter that indicates how long it takes to react to a stimulus. *Answer Priority Type* is a parameter that identifies the priority of the person to receive a specific command (i.e. the ele-

ments that are diffused in the model), and it can be based on *who* is sending the command, on the *device* that is receiving the message, or at random. *Send Priority Type* identifies how the person chooses the commands s/they will send. Their priority can be based on a *priority list*, on *arrival time* or at random. The value inside the tags represents the value of the attribute. Here, we have First Responder 20, located in position 55D6. Their reaction time is 10s, and they prioritize the reception of commands based on the device they came from. For sending commands, they have a priority list. *MyDevices* includes all the devices the agent can use; and it has as many elements as devices. Each device is represented as a tag (*PriorityDevice*) with four attributes (*priority*, *device*, *send2Multiple*, *sendSeparateFromReceive*). *SortedTasks* represents how the agent sorts the tasks they should conduct under an emergency scenario. *AnswerDevicePriority* can have as many entries as devices. *AnswerPersonPriority* has as many elements as individuals the agent has relation with, with two attributes: *priority* and *id*. In this example, receiving a message from person 1 has the highest priority. Person 97 has priority 3. *SendCommandPriority* classifies the set of messages the person may send during an emergency. Every element has three attributes: *priority*, receiver (*to*) and content of a message (*msg*). In this example, transmitting "Tell population to stay home" to person 1 has a high priority. Transmitting "Tell population to stay home" to person 97 has priority 3. *ActionExecutionPriority* has two elements: *priority* and *id*. In this case, "Tell people to stay at home" has the highest priority. *CommunicationRelations* identifies the relations with different individuals. It has one element per individual the agent is connected to. *MessageBehavior* represents how the agent behaves when they receive messages. Finally, *ActionBehavior* identifies how the person should behave when doing an action.

As we mentioned, in order to implement the coupled models, we first need to instantiate the atomic models inside them. To do so, we use one function for each type of coupled. An example of this function is shown in Figure 9.

```
1 /***Instantiate atomics inside the coupled***/
2 pair<vector<string>,vector<string>> AtomicsCoupled;
3 create_atomics_text_msg_device(DeviceType, Id, delay,
4         outOfOrderAcknow){
5 create_atomic_inbox(DeviceType,Id, delay,outOfOrderAcknow);
6  string inbox = "inbox"+DeviceType+Id;
7 create_atomic_outbox(DeviceType,Id,delay, outOfOrderAcknow);
8  string outbox = "outbox"+DeviceType+Id;
9  create_atomic_msgClassifierNewReadCon(DeviceType, Id));
10
11  /****Define coupled: first the I/O ports ****/
12  "using iports_"+DeviceType+Id+"=<inp_setOutOfOrder,
13 inp_network, inp_fromKeyboard>;";   // input ports
14 "using oports_"+DeviceType+Id+
15          "=<outp_toScreen,outp_network>;";
16
17  "using submodels_"+DeviceType+Id+"= models_tuple<"+inbox+
18 ","+outbox+","+msgClassifierNewReadCom+">;"; // SUBMODELS
19
20  //External Input Couplings - eics
21  "using eics_"+DeviceType+Id+" =tuple<"EIC<
22   inp_setOutOfOrder," +inbox+",inbox_defs<SetDeviceState>
23        ::setStateIn>,";
24  "EIC<inp_setOutOfOrder,"+outbox+",outbox_defs
25       <SetDeviceState>::setStateIn>,";
26  "EIC<inp_network,"+inbox+", inbox_defs<SetDeviceState>
27       ::newIn>,";
28  "EIC<inp_fromKeyboard,")+msgClassifierNewReadCom+",
29      msgClassifierNewRead_defs<Communication>::in>>;";
30
31  //External Input Couplings - eocs
32  "using eocs_"+DeviceType+Id+" =tuple<";
33  "EOC<"+inbox+",inbox_defs<SetDeviceState>::displayOut,
34         outp_toScreen>,";
35  "EOC<"+outbox+", outbox_defs<SetDeviceState>::displayOut,
36     outp_toScreen>,";
37   "EOC<"+outbox+", outbox_defs<SetDeviceState>::networkOut,
38        outp_network>";
39  >;"
40
41  //Internal Couplings - ics
42  "using ics_"+DeviceType+Id+" =tuple<";
43  "IC<"+msgClassifierNewReadCom+",msgClassifierNewRead_defs
44     <Communication>::newOut,"+outbox+",outbox_defs
45    <SetDeviceState>::newIn>,";
46  "IC<"+msgClassifierNewReadCom+", msgClassifierNewRead_defs
47     <Communication>::readout,"+inbox+" ,inbox_defs
48     <SetDeviceState>::readIn>>;";
49 }
```

Figure 9 Generating the DEVS computerized model of the coupled models e-mail, beeper, and fax

These functions use the XML file in figure 8, and we convert it into the syntax needed by CDBoost. The connections inside the coupled are defined using the logics explained in figure 2 for the coupled models implementation. The rules are written in a way that the output of the function (shown in figure 10) contains all the code needed. Figure 9 shows the implementation of the function used to instantiate a coupled model representing devices that send/receive text (i.e. email). Figure 10 shows the output of this function: the atomics inside the coupled are instantiated and the coupled model is defined following CDBoost definitions, so it can be simulated. We have chosen a simple example to explain the logic behind it. The rest of the functions are implemented following a similar logic taking into account more parameters of the XML file.

```
1 //Atomic models inside the instantiated coupled model
2 template<typename TIME>
3 class msgClassifierNewReadCom : public
4    msgClassifierNewRead<Communication, TIME> {
5 public:
6 msgClassifierNewReadCom(): msgClassifierNewRead<
7    Communication, TIME>(TIME("00:00:500")) {};
8  };
9 template<typename TIME>
10 class inboxFAX1 : public inbox<SetDeviceState, TIME> {
11 public:
12 inboxFAX1():inbox<SetDeviceState,TIME>(DeviceId (DeviceType
13   ::FAX,"1"),TIME("00:00:500"), TIME("00:01:000")){};
14 };
15 template<typename TIME>
16 class outboxFAX1 : public outbox<SetDeviceState, TIME> {
17 public:
18 outboxFAX1(): outbox<SetDeviceState, TIME>(DeviceId
19  (DeviceType::FAX,"1"),TIME("00:00:500"),TIME("00:01:000"))
20    {};      };
21 // instantiated coupled model
```

```
22 using iports_FAX1 = tuple<inp_setOutOfOrder,inp_network,
23         inp_fromKeyboard>;
24 using oports_FAX1 = tuple<outp_toScreen,outp_network>;
25 using submodels_FAX1=models_tuple<inboxFAX1,outboxFAX1,
26    msgClassifierNewReadCom>;
27 using eics_FAX1 =std::tuple<
28   EIC<inp_setOutOfOrder,inboxFAX1, inbox_defs
29        <SetDeviceState>::setStateIn>,
30   EIC<inp_setOutOfOrder,outboxFAX1, outbox_defs
31        <SetDeviceState>::setStateIn>,
32   EIC<inp_network, inboxFAX1, inbox_defs
33        <SetDeviceState>::newIn>,
34   EIC<inp_fromKeyboard,msgClassifierNewReadCom,
35        msgClassifierNewRead_defs<Communication>::in> >;
36
37 using eocs_FAX1 =tuple<
38   EOC<inboxFAX1,inbox_defs<SetDeviceState>::displayOut,
39     outp_toScreen>,
40   EOC<outboxFAX1, outbox_defs<SetDeviceState>::displayOut,
41     outp_toScreen>,
42   EOC<outboxFAX1, outbox_defs<SetDeviceState>::networkOut,
43        outp_network>       >;
44
45 using ics_FAX1 =std::tuple<
46  IC<msgClassifierNewReadCom, msgClassifierNewRead_defs
47      <Communication>::newOut, outboxFAX1,outbox_defs
48       <SetDeviceState>::newIn>,
49  IC<msgClassifierNewReadCom, msgClassifierNewRead_defs
50     <Communication>::readOut,inboxFAX1, inbox_defs
51      <SetDeviceState>::readIn>             >;
```

Figure 10. Output of the function explained in Figure 9

In Figure 9 (lines 1-9), we instantiate the atomic models used inside the couple as we show in Figure 10 (lines 1-20). We call a function that takes as inputs the atomic model parameters and returns the model instantiated in a format that CDBoost understands. The function takes as inputs the type of text message device (i.e. e-mail, fax or beeper), the id of the person that owns the device (i.e. the Id in the agent XML file), and two characteristics of the devices: the delay introduced in the communication and the time it takes to acknowledge that it is out of order.

The rest of the figure defines the coupled model instantiation. Lines 11-19 (Figure 9) returns the coupled model input and output ports and the submodels inside the coupled implemented as a tuple as shown in Figure 10 (lines 21-26). Lines 20-30 (Figure 9) generates the External Input Couplings (EIC) as a tuple of tuples of 3 elements: the name of the input port, the name of submodel connected to the input port, and the input port name of the submodel (lines 27-36 figure10) External Output Couplings (EOC) are defined as the EIC but with a different order: submodel name, output port name of the sub model and output port name of the coupled (see lines 31-39 in Figure 9 for the function definition and lines 37-43 in Figure 10 for the output). Finally, Internal Couplings (IC) are defined as a tuple of tuples of four elements: name of the outcoming subcomponent, sub model output port name, the name of the incoming sub, sub model input port name. In Figure 9 (lines 41-49), we show the code that generates the implementation. The output of the code is shown in Figure 10 (lines 45-51).

The top-level model is built using a program that takes the XML files where the agents are defined, it reads each XML file and transforms them into a structure to generate the parameters of all the functions explained earlier in this section. The output is a file with thousands of lines of code that CDBoost understands. This file includes all the atomic and coupled models' instantiated, which, once compiled, generates the Diffusion Computer model ready to generate results.

```
1 int main(int argc, char ** argv) {
2     int numberOfPersons = stoi(argv[1]);
3     string folder = argv[2];
4     string mainModel = string("../TOPMODEL/MainTop.cpp");
5     string content, tSUBMODELS, tIC, tEIC, tEOC, tIPORTS;
6     string tOPORTS = "outp_taskDeviceFinished,
7         outp_taskActionFinished";
8
9      myModelfile.open(mainModel);
10      TOP = open_coupled(string("TOP"));
11
12    ifstream infile("NEP_Cadmium_Headers");
13   //Define Headers and I/O ports inside MainTop.cpp
14    for(int i=0; infile.eof()!=true ; i++)
15   // get content of infile
16        content += infile.get();
17    myModelfile << content << endl;
18
19    for(int i = 1; i <= numberOfPersons; i++){
20     // DEVICES
21     in = folder+string("P")+to_string(i)+string(".xml");
22     person.load(in);
23    DEVICES = DevicesCoupledModel(person);
24      for(int j = 0; j<DEVICES.first.size(); j++)
25        myModelfile << DEVICES.first[j] << endl;
26      for(int j = 0; j<DEVICES.second.size(); j++)
27        myModelfile << DEVICES.second[j] << endl;
28    }
29 ...
```

Figure 11. Code snippet of the top model generator

Figure 11 shows a part of the program that generates the top model (i.e. an instance of the DAM to study an information diffusion process), which can be seen in Figure 12. The rest of the program is developed following the same logic. We use the number of agents (i.e. the number of XML files to be loaded) and their directory path. The number of agents is used to define the number of instances of *Devices* and *Person* models inside the coupled model, as shown in lines 2 and 19. In lines 5-7 we define all the variables needed to define the top coupled model. Then, we define our coupled model. First, we parse a file where the headers of CDBoost and of the parameterized DEVS atomic models are defined (lines 12-17). The top model ports are also defined in that file. The output of this part of the program is shown in figure 18, lines 1-8. Then, we call the functions explained earlier to generate the DEVS component in the top model. In lines 19-28 (Figure 11), we show the definition of all the *Devices* coupled models. For each agent, we define a *Devices* model by loading the proper XML and calling the function that generates the coupled model (line 23). We then generate all the atomic instantiated models and the coupled model in *MainTop.cpp* (lines 24-27). Figure 12 shows a code snippet of the output of this.

```
1 struct inp_generator : public in_port<Command>{};
2// SET INPUT PORTS FOR COUPLED
3 struct inp_network : public in_port<Communication>{};
4 ...
5 outp_myLocation : public out_port<PeopleLocation>{};
6   // SET OUTPUT PORTS FOR COUPLED
7 outp_network : public out_port<Communication>{};
8 ...
9 template<typename TIME>
10   // Define atomic and coupled unit devices
11 class filterDevicesNetwork1: public
12      filterDevicesNetwork<TIME> {
13 public: filterDevicesNetwork1():
14       filterDevicesNetwork<TIME>("1") {};            };
15
16 template<typename TIME>
17 class filterDevicesSetOutOrder1: public
18        filterDevicesSetOutOrder<TIME> {
19 public: filterDevicesSetOutOrder1():
20       filterDevicesSetOutOrder<TIME>("1") {};        };
21
22 template<typename TIME>
23 class phoneMOBILEPHONE1 : public phone<SetDeviceState,
24            TIME> {
25 public: phoneMOBILEPHONE1(): phone<SetDeviceState,TIME>
26  (DeviceId(DeviceType::MOBILEPHONE, "1"),TIME("00:00:500"),
27   TIME("00:01:000")) {};   };
28
29 template<typename TIME>
30 class phoneLANDLINEPHONE1 : public phone<SetDeviceState,
31      TIME> {
32 public: phoneLANDLINEPHONE1(): phone<SetDeviceState,
33    TIME>(DeviceId(DeviceType::LANDLINEPHONE,"1"),
34    TIME("00:00:500"),TIME("00:01:000")) {};          };
35//DEFINE COUPLED DEVICE
36 using iports_DEVICES1 = tuple<inp_setOutOfOrder,
37   inp_in_com,inp_network>;
38 using oports_DEVICES1 = tuple<outp_out_com, outp_network>;
39 using submodels_DEVICES1 = models_tuple<
40   filterDevicesSetOutOrder1, filterDevicesNetwork1,
41    filterDevicesMicroKeyboard, sinkDevices_atomic,
42   phoneMOBILEPHONE1, phoneLANDLINEPHONE1,>
43 using eics_DEVICES1 = tuple<
44   EIC<inp_setOutOfOrder,filterDevicesSetOutOrder1,
45     filterDevicesSetOutOrder_defs::in>,
46   EIC<inp_in_com,filterDevicesMicroKeyboard,
47     filterDevicesMicroKeyboard_defs::in>,
48   EIC<inp_network,filterDevicesNetwork1,
49     filterDevicesNetwork_defs::in>         >;
50 ...
```

Figure 12. Output of the program defined in Figure 11

5 CASE STUDY: INFORMATION DIFFUSION

In this section, we present a case study where the DAM is used to simulate an information diffusion process in an organization. Inside the organization, the people have different communication mechanism to transmit the information such as Landline Phones, Radio, E-mail, Mobile Phones, Face-to-Face communications, etc.

We used the original data from an existing Nuclear Emergency Plan (NEP) in Spain. All the requirements related to the problem are defined in a requirements document [33]–[35] that contains a comprehensive definition of the NEP organization, the communications means to transmit the information and the rules each person follows to transmit the information. Using this data, we defined the XML files presented in Figure 8 for each person involved in the diffusion process.

To study the information diffusion process we instantiate the DAM into a NEP DAM as follows:

- Each *Node* is instantiated as *Person* (i.e. as a person working in the emergency), whose behavior is defined using a parameterized DEVS coupled model.
- Each *Indirect Link* model is instantiated as a *Devices* model. *Devices* is a coupled model that contains all the devices the specific person can use. It is instantiated using the attribute <*MyDevices*> in the XML file.
- The *Links Connector* is mapped into the *Networks* (i.e. a coupled model that contains all the networks connecting specific devices: Radio, Internet, satellite, etc.).
- The *Direct Link* is mapped to a *Face-to-Face Connector* because in our model the direct links represent people talking face to face.
- *Indirect Links Updater* and *Links Connectors Updater* are instantiated to *Devices* and *Networks Updaters* respectively since they introduce change on the state of *Devices* and *Networks* models. They model if the devices or networks break or recover dynamically.
- *Node Updater* is mapped to *People in Location*, since the only attribute of the *Person* model we want to track is the people who share location and therefore they can communicate face-to-face.
- *Direct Link Updater* is not included in this specific instantiation since we are not interested in modifying the properties that may affect face-to-face communications such as environmental noise.
- *Diffusion Element Generator* is converted into the *Command Generator* since the diffusion elements in this specific process are commands that give information to people to solve the emergency.

Based on this definition, we executed a version of the NEP DAM including 149 persons with their Devices, including the Head of the Nuclear Emergency Plan and the Radiological Group. The Computer Model is generated automatically using the XML files that represent the individual behavior of each person, as explained in section 4. If the data presented in the requirements document is stored in a structured way (e.g. tables), it can be automatically parsed to create these XML files. The data in the requirements document can also be directly stored in XML format. If the data is stored in natural language as a set of text specifications, it should be manually translated to XML format or an intermediated structured format that allows creating the XML files automatically.

We have focused on studying what happens when the command "Establish Emergency Level 0" is decreed by the NEP Director and the specific communication device inside the Radiological Group fails with different probabilities. The failure may represent that the device runs out of battery, it does not receive

a signal, it breaks, etc. We have simulated different scenarios where this device fails with different probabilities (i.e. 10%, 20%, etc.).The simulations represent a 95% Confidence Interval for the mean of people that receive the command "Establish Emergency Level 0". The confidence interval is represented as notches in the plot (figure 13). This analysis provides some information to decision makers and it is useful to validate our model.

Based on the NEP specifications, we know that 63 people should receive a command from the head of the radiological group. We also know that the Radiological group only uses a *radiological group device* (RGD, a specific device with mixed radiophone communication).

Figure 13 shows the number of people that receive the command "Establish Emergency Level 0" when we simulate different probabilities of failure of the *RGD*. We use a box plot in which the triangle represents the mean, the horizontal line the median and the circles the outliers.

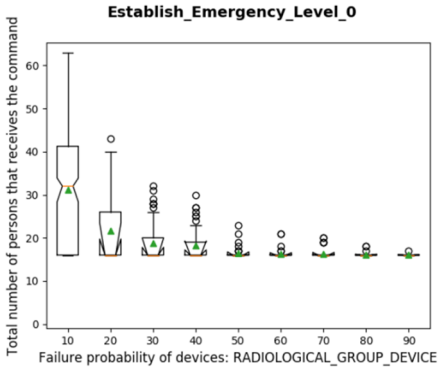


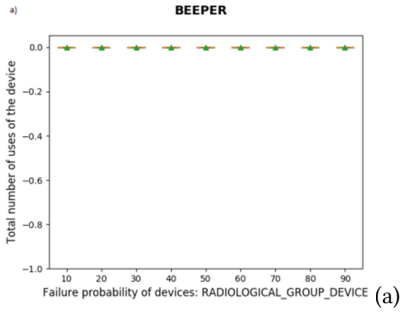
Figure 13 *RGD* failures

We can see that, regardless the failure probability, some people always receive the command. This number remains constant since they are part of the NEP leadership and they do not use the RGD to communicate. However, even with a 10% failure probability, in 75% of the cases, less than 40 people receive the command and the median is around 30. If the failure probability increases to 20%, the median is drastically reduced to less than 20 people. This value remains constant when the failure probability increases over 20%. Based on this analysis and taking into account the definitions of the NEP, we can conclude that we cannot afford a failure rate of only 10% in the RGDs because in more than 75% of the cases less than 40 people out of 63 receive the command.

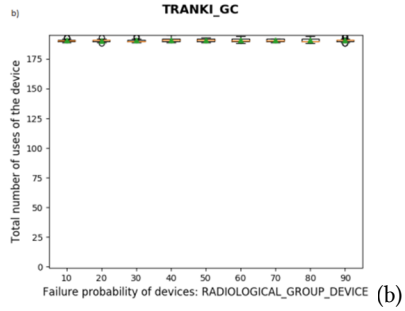
Figure 14 shows how many times each device is used based on the failure probability of the RGD. We use these results to validate our model.

In figure 14 a), we can see that the beeper is not used (the mean, medium and quartiles are all zero). Although only the beeper is shown, we obtain the same results for fax, e-mail, private landline phone, two radio channels - REMAR and REMER -, satellite phone, and TrankiE - a phone-radio used by the police -. This result is correct, as the specification document says that none of these devices should be used by the radiological group.

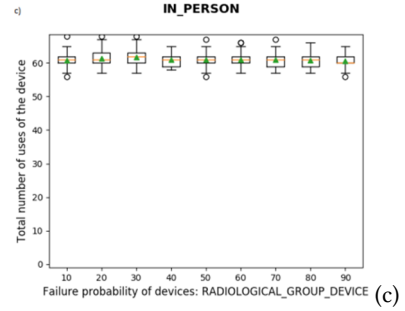
In figure 14 (b-e), we show that the data distribution is uniform when we simulate failures in the RGD. The number of attempts to establish the communication causes variability in the different simulations. These results validate the model based on the NEP specifications, which says that the Radiological Group only use the RGD. This restriction justifies why the plots in figure 14 (a-e) are uniform for the different failure probabilities.



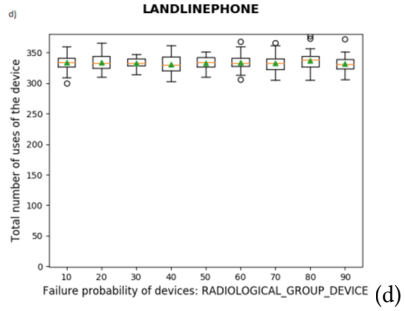
(a)



(b)



(c)



(d)

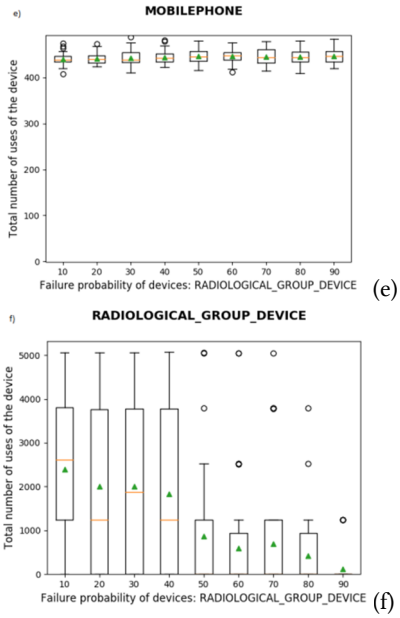


Figure 14 Number of activations of the different devices when the RGD fails with different probabilities.

Figure 14 f) shows two different trends. When the failure probability is low (less than 50%), the number of activations of the RGD is high. The variability for each failure probability is also high (i.e. wide interquartile range). When the failure probability is 50% or greater, the number of activations is significantly reduced and the variability is lower. The mean of the number of RGD activations shows a decreasing trend. When the failure probability is low, there are many devices working. If a device does not fail, the owner keeps trying to communicate (i.e. the total number of activations of the device is high). But if they see that their device is not working, they stop using it. Therefore, when a device fails and the owner has something to send the information transmission process is blocked. In those cases, the number of activations for the devices is lower. An increase in the failure probability is translated in an increase of the number of devices broken. Then, the probability to block the information transmission increases. This explains why the mean decreases. Additionally, figure 14 f) shows, that regardless the failure probability, there are cases (i.e. simulations) where the RGD is activated just one time. These results show that there is a critical person in the process, and they can block the whole information transmission if their device is broken (this has been confirmed analyzing the simulation logs and NEP specifications), which confirms that if the device of the Radiological Group head is broken, the whole process is blocked.

Based on these results, we can see we need to review the communications within the Radiological group. The simulation results allowed us to come with following questions that affect the organization: why the people within the radiological group cannot use their mobile phone or e-mail? Is there any security issue (e.g. authentication, encryption, etc.)? The discussion of these questions with decision makers will bring new scenarios to analyze to test different solutions. Then, to simulate a new scenarios they just update the model parameters in the XML files, such as the devices for each person or the communication devices that they can use with other people. Then, they run the program that automatically instantiates the DAM and generates the computerized model and runs the simulation.

6 CONCLUSIONS

We presented an Abstract Model for diffusion processes in multiplex networks, and discussed its definition and implementation using DEVS and CDBoost. To show how we get results using the DAM, we presented an information diffusion process inside an organization (a real nuclear emergency plan from Spain), and studied the effects of failures in their communication channels, focusing on the effect of different failure probabilities.

By using the DAM to simulate diffusion process in multiplex dynamic networks, we show how a formal definition provides several advantages which were detailed along the paper. A major advantage is that it allows validating the model before implementation, which improves the model quality and cost. The case study presented this the paper has shown how the DAM can generate results for different scenarios without modifying a line of code. We have also provided information to decision makers in order to improve the communications inside one group of the NEP: the radiological group.

Future research lines will focus applying the DAM to study other diffusion processes in multiplex dynamic networks.

REFERENCES

[1] M. Newman, "The structure and function of complex networks," *SIAM Rev.*, vol. 45, no. 2, pp. 167–256, 2003.
[2] S. Gómez, A. Díaz-Guilera, J. Gómez-Gardeñes, C. J. Pérez-Vicente, Y. Moreno, and A. Arenas, "Diffusion Dynamics on Multiplex Networks," *Phys. Rev. Lett.*, vol. 110, no. 2, p. 28701, 2013.
[3] H. Xiong, W. Puqing, and G. V Bobashev, "Multiple Peer Effects in the Diffusion of Innovations on Social Networks: A Simulation Study," *SSRN Electron. J.*, 2015.
[4] A. Khelil, C. Becker, J. Tian, and K. Rothermel, "An epidemic model for information diffusion in MANETs," *Proc. 5th ACM Int. Work. Model. Anal. Simul. Wirel. Mob. Syst. - MSWiM '02*, p. 54, 2002.
[5] C. Ruiz-Martin, "An architecture to simulate diffusion processes in multiplex dynamic networks," in *2017 Winter Simulation Conference (WSC)*, 2017, pp. 4630–4631.
[6] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems.* Academic press, 2000.
[7] V. Capasso and G. Serio, "A generalization of the Kermack-McKendrick deterministic epidemic model," *Math. Biosci.*, vol. 42, no. 1–2, pp. 43–61, 1978.
[8] W. Wang, Q.-H. Liu, S.-M. Cai, M. Tang, L. A. Braunstein, and H. E. Stanley, "Suppressing disease spreading by using information diffusion on multiplex networks," *Sci. Rep.*, vol. 6, no. 7600, p. 29259, 2016.
[9] C. Granell, S. Gomez, and A. Arenas, "Dynamical interplay between awareness and epidemic spreading in multiplex networks," *Phys. Rev. Lett.*, vol. 111, no. 12, pp. 1–10, 2013.
[10] A. Raj, A. Kuceyeski, and M. Weiner, "A Network Diffusion Model of Disease Progression in Dementia," *Neuron*, vol. 73, no. 6, pp. 1204–1215, 2012.
[11] O. Yağan and V. Gligor, "Analysis of complex contagions in random multiplex networks," *Phys. Rev. E - Stat. Nonlinear, Soft Matter Phys.*, vol. 86, no. 3, pp. 1–11, 2012.
[12] E. Cozzo, R. A. Baños, S. Meloni, and Y. Moreno, "Contact-based Social Contagion in Multiplex Networks," *Phys. Rev. E - Stat. Nonlinear, Soft Matter Phys.*, vol. 88, no. 5, pp. 1–5, Jul. 2013.
[13] E. Estrada and J. Gómez-Gardeñes, "Communicability reveals a transition to coordinated behavior in multiplex networks," *Phys. Rev. E - Stat. Nonlinear, Soft Matter Phys.*, vol. 89, no. 4, pp. 1–5, 2014.
[14] M. Kivela, A. Arenas, M. Barthelemy, J. P. Gleeson, Y. Moreno, and M. A.

Porter, "Multilayer networks," *J. Complex Networks*, vol. 2, no. 3, pp. 203–271, Sep. 2014.
[15] S. Boccaletti, G. Bianconi, R. Criado, C. I. del Genio, J. Gomez-Gardeñes, M. Romance, I. Sendiña-Nadal, Z. Wang, and M. Zanin, "The structure and dynamics of multilayer networks," *Phys. Rep.*, vol. 544, no. 1, pp. 1–122, 2014.
[16] Y. Jiang and J. C. Jiang, "Diffusion in Social Networks: A Multiagent Perspective," *IEEE Trans. Syst. Man, Cybern. Syst.*, vol. 45, no. 2, pp. 198–213, Feb. 2015.
[17] Y. Bouanan, G. Zacharewicz, B. Vallespir, J. Ribault, and S. Y. Diallo, "DEVS based Network : Modeling and Simulation of Propagation Processes in a Multi-Layers Network," in *Proceedings of the Modeling and Simulation of Complexity in Intelligent, Adaptive and Autonomous Systems 2016*, 2016.
[18] C. Ruiz-Martin, Y. Bouanan, G. Wainer, G. Zacharewicz, and A. Lopez-Paredes, "A hybrid approach to study communication in emergency plans," in *Proceedings of the 2016 Winter Simulation Conference*, 2016, pp. 1376–1387.
[19] F. J. Barros, "Modeling formalisms for dynamic structure systems," *ACM Trans. Model. Comput. Simul.*, vol. 7, no. 4, pp. 501–515, 1997.
[20] M. Sbayou, Y. Bouanan, G. Zacharewicz, J. Ribault, and J. François, "DEVS modelling and simulation for healthcare process application for hospital emergency department," *Simul. Ser.*, vol. 49, no. 1, 2017.
[21] J. Himmelspach and A. M. Uhrmacher, "A component-based simulation layer for JAMES," in *Proceedings of the eighteenth workshop on Parallel and distributed simulation - PADS '04*, 2004, p. 115.
[22] J.-B. Filippi and P. Bisgambiglia, "JDEVS: an implementation of a DEVS based formal framework for environmental modelling," *Environ. Model. Softw.*, vol. 19, no. 3, pp. 261–274, Mar. 2004.
[23] H. S. Sarjoughian and B. P. Zeigler, "DEVSJAVA : Basis for a DEVS-based Collaborative M & S Environment," in *Proceedings of SCS International Conference on Web-Based Modeling and Simulation*, 1998.
[24] D. Vicino, D. Niyonkuru, G. Wainer, and O. Dalle, "Sequential PDEVS Architecture," in *DEVS '15 Proceedings of the Symp on Theory of M&S: DEVS Integrative M&S Symposium*, 2015, pp. 165–172.
[25] C. Ruiz-Martin, G. Wainer, and A. Lopez-Paredes, "DISCRETE-EVENT SIMULATION OF DIFFUSION PROCESSES IN DYNAMIC MULTIPLEX NETWORKS," *SIMPAT*.
[26] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks.," *ICWSM*, vol. 8, pp. 361–362, 2009.
[27] W. De Nooy, A. Mrvar, and V. Batagelj, *Exploratory social network analysis with Pajek*. Cambridge University Press, 2005.
[28] M. De Domenico, M. A. Porter, and A. Arenas, "MuxViz: a tool for multilayer analysis and visualization of networks," *J. Complex Networks*, p. cnu038, 2014.
[29] C. Nikolai and G. Madey, "Tools of the Trade : A Survey of Various Agent Based Modeling Platforms," *J. Artif. Soc. Soc. Simul.*, vol. 12, no. 22, 2009.
[30] G. Wainer, *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. CRC Press, 2009.
[31] R. Ihaka and R. Gentleman, "R : A Language for Data Analysis and Graphics," *J. Comput. Graph. Stat.*, vol. 5, no. 3, pp. 299–314, 1996.
[32] Microsoft, "Power BI," 2015. [Online]. Available: https://powerbi.microsoft.com/es-es/.
[33] C. Ruiz-Martin, "Modelo Organizational para la Gestión de Emergencias," Universidad de Valladolid, 2013.
[34] C. Ruiz-Martin, M. Ramírez Ferrero, J. L. Gonzalez-Alvarez, and A. Lopez-Paredes, "Modelling of a Nuclear Emergency Plan: Communication Management," *Hum. Ecol. Risk Assess. An Int. J.*, vol. 21, no. 5, pp. 1152–1168, 2015.
[35] C. Ruiz-Martin, A. Lopez-Paredes, and G. Wainer, "Applying complex network theory to the assessment of organizational resilience," *IFAC-PapersOnLine*, vol. 48, no. 3, pp. 1224–1229, 2015.

APPENDIX I

The formal definition of Switch atomic model is as follows:

$$Switch(Id) = < X, Y, S, ta, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda >$$

Where

$$X = \begin{Bmatrix} (\text{"diffusionElementIn"}, diffusionElement), \\ (\text{"setAnswerIn"}, setStateAnswer), \\ (\text{"setSendIn"}, setStateSend), \\ (\text{"setDecideIn"}, setStateDecide) \end{Bmatrix}$$

$diffusionElement \in DiffusionElements$

$DiffusionElements \in \forall structure\ with\ field\ \text{"}de\text{stinatary"}$

$setStateAnswer, setStateSend, setStateDecide \in \forall structure$

$$Y = \begin{Bmatrix} (\text{"sendOut"}, diffusionElement \cup \emptyset)U \\ (\text{"answerOut"}, diffusionElement \cup \emptyset)U \\ (\text{"decideOut"}, diffusionElement \cup \emptyset) \mid \\ diffusionElement \in (DiffusionElements \mid destinatary = Id) \end{Bmatrix}$$

$$S = \begin{Bmatrix} messagesPassing, state \mid \\ messagesPassing = \emptyset \cup \\ messagesPassing\{diffusionElement \mid destinatary = Id\} \\ state \in \{ANSWER, SEND, DECIDE\} \end{Bmatrix}$$

$$ta(s) = \begin{Bmatrix} messagesPassing = \emptyset \rightarrow \infty \\ messagesPassing = \{diffusionElement \mid destinatary = Id\} \rightarrow 0ms \end{Bmatrix}$$

$$\delta_{ext}(S, e, X) = \begin{Bmatrix} (X\ in\ \text{diffusionElementIn} \mid destinatary = Id) \rightarrow messagesPassing += X \\ (X\ in\ \text{setAnswerIn}) \rightarrow state = ANSWER \\ (X\ in\ \text{setSendIn}) \rightarrow state = SEND \\ (X\ in\ \text{setDecideIn}) \rightarrow state = DECIDE \end{Bmatrix}$$

$$\delta_{int}(S) = \{messagesPassing = \emptyset\}$$

$$\delta_{con}(S, e, X) = \delta_{int}(S) + \delta_{ext}(S, e, X)$$

$$\lambda(S) = \begin{Bmatrix} if(state = ANSWER) \rightarrow send\ messagesPassing\ by\ answerOut \\ if(state = SEND) \rightarrow send\ messagesPassing\ by\ sendOut \\ if(state = DECIDE) \rightarrow send\ messagesPassing\ by\ decideOut \end{Bmatrix}$$

APPENDIX II

The formal definition of Indirect Links coupled model is as follows:

$$Indirect\ Links = < X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC >$$

Where

$$X = \begin{Bmatrix} (\text{"NodeIn"}, diffusionElement)U \\ (\text{"LinkConnectorsIn"}, diffusionElement)U \\ (\text{"UpdaterIn"}, stateUpdate) \end{Bmatrix}$$

$diffusionElement \in DiffusionElements$

$stateUpdate \in StateUpdates$

$DiffusionElements \in \forall structure\ with\ field\ \text{"}de\text{stinatary"}$

$StateUpdates \in \forall structure\ with\ field\ \text{"}LinkType\text{"}$

$$Y = \begin{Bmatrix} (\text{"}NodeOut\text{"}, diffusionElement)U \\ (\text{"}LinkConnectorsOut\text{"}, diffusionElement) \end{Bmatrix}$$

$$D = \begin{Bmatrix} FilterNode, FiterUpdater, FilterLinkConnectors, \\ LinkType_1, LinkType_2, \dots, LinkType_n, \\ LinkSink \end{Bmatrix}$$

$$M = \begin{Bmatrix} M_{FilterNode}, M_{FiterUpdater}, \dots, M_{FilterLinkConnectors}, \\ M_{LinkType1}, M_{LinkType2}, \dots, M_{LinkTypen}, \\ M_{LinkSink} \end{Bmatrix}$$

$$EIC = \begin{Bmatrix} ((Self, Self_{NodeIn}), (FilterNode, FilterNode_{In})), \\ \left((Self, Self_{UpdaterIn}), (FilterUpdater, FilterUpdater_{In})\right), \\ ((Self, Self_{LinkConnectorsIn}), (FilterLinkConnectors, FilterLinkConnectors_{In})) \end{Bmatrix}$$

$$EOC = \begin{Bmatrix} ((LinkType1, LinkType1_{NodeOut}), (Self, Self_{NodeOut})), \\ \dots \\ ((LinkTypen, LinkTypen_{NodeOut}), (Self, Self_{NodeOut})) \\ ((LinkType1, LinkType1_{LinkConnectorsOut}), (Self, Self_{LinkConnectorsOut})), \\ \dots \\ ((LinkTypen, LinkTypen_{LinkConnectorsOut}), (Self, Self_{LinkConnectorsOut})) \end{Bmatrix}$$

$$IC = \begin{Bmatrix} ((FilterNode, FilterNode_{LT1Out}), (LinkType1, LinkType1_{NodeIn})), \\ \dots \\ ((FilterNode, FilterNode_{LTnOut}), (LinkTypen, LinkTypen_{NodeIn})), \\ ((FilterNode, FilterNode_{LTn+1Out}), (LinkSink, LinkSink_{NodeIn})), \\ \dots \\ ((FilterNode, FilterNode_{LTmOut}), (LinkSink, LinkSink_{NodeIn})), \\ \left((FilterUpdater, FilterUpdater_{LT1Out}), \left(LinkType1, LinkType1_{UpdaterIn}\right)\right), \\ \dots \\ \left((FilterUpdater, FilterUpdater_{LTnOut}), \left(LinkTypen, LinkTypen_{UpdaterIn}\right)\right), \\ \left((FilterUpdater, FilterUpdater_{LTn+1Out}), \left(LinkSink, LinkSink_{UpdaterIn}\right)\right), \\ \dots \\ \left((FilterUpdater, FilterUpdater_{LTmOut}), \left(LinkSink, LinkSink_{UpdaterIn}\right)\right), \\ ((FilterLinkConnectors, FilterLinkConnectors_{LT1Out}), (LinkType1, LinkType1_{ConnectorsIn})), \\ \dots \\ ((FilterLinkConnectors, FilterLinkConnectors_{LTnOut}), (LinkTypen, LinkTypen_{ConnectorsIn})), \\ ((FilterLinkConnectors, FilterLinkConnectors_{LTn+1Out}), (LinkSink, LinkSink_{ConnectorsIn})), \\ \dots \\ ((FilterLinkConnectors, FilterLinkConnectors_{LTmOut}), (LinkSink, LinkSink_{ConnectorsIn})) \end{Bmatrix}$$

$n = \#LinkTypes\ in\ Indirect\ Link$

$m = Total\ \#LinkTypes\ in\ the\ model$