Observations in DEVS framework

G. Quesnel[1], É. Ramat[2] and R. Duboz[3]

[1]National Institute for Agricultural Research, Biometrics and Artificial Intelligence unit, France

[2]Université du littoral côte d'opale, Computer Science Lab. of Littoral, France

[3]Agricultural Research for Developing Countries, Research Unit 22, France

Abstract

To observe a model is of key importance in the modeling and simulation activity. In this paper, we distinguish two types of observation in dynamical models: observations of model behaviors for analysis and observations of model performed by another model. Even in the first usual case, the observation mechanism is generally not explicit nor specified. This is generally not a problem unless we want to clearly specify the algorithmic in experimental frames. Secondly, a model can need the state of another one to compute its own state. In this case, the model "observes" another one to get its state. This is not quite the same compare to the first type. Indeed, in this case, the observation participates in the global dynamics. In this work, we specify these two types of observation in the Discrete Event System Specification (DEVS) formalism. To achieve that, we extend the formalism with two functions, the request function and the observation function. It responds to a pragmatic need for the specification and implementation of complex experimental frames and optimisation techniques and to simplify the development of models where numerous entities interact. We give the abstract simulator algorithms we have implemented in our DEVS simulators, the Virtual Laboratory Environment (VLE).

keywords: : Computer, Methodology, Simulation, System dynamics

1 Introduction

In the modeling and simulation activity, the observation of model's behavior play a very important role. Nevertheless, mechanisms to achieve such observations are generally not explicit nor specified. To capture the dynamics of models, the modeler generally connect simulation software to output streams like files, database or visualisation software for instance. This is generally not useful to specify it unless we want the observation to be modeled as a part of an experimental frame [Traoré and Zeigler, 2003], where simulation results have to be capture following a precise experimental plan (for example when the plan defines the time step to perform observation). Furthermore, for dynamical experimental frames, where observations dynamics has to be specified (in the field of optimisation by simulation [Gosavi, 2003], simulation results serve as an input to compute

new parameters and run a new simulation. Observations here do not participate of the model dynamics itself. Another type of observation can be identified when a model needs to know the state of another one to compute its own state. This is often the case in spatial explicit model [Wainer and Giambiasi, 2001, Quesnel et al., 2005, Versmisse and Ramat, 2005] Individual Based Model (IBM) or Agent Based Model (ABM) [Duboz et al., 2006] and in coupled equation systems [Kofman and Junco, 2001]. In these models, observations participate in the global dynamics. We think it is important to provide a clear semantic for these two types of observation.

In this paper, we consider the Discrete Event Specification System (DEVS) [Zeigler, 1976, Zeigler et al., 2000]. This formalism is well situated to clearly specified both system dynamics and experimental frame. Nevertheless, even if it is possible to specify observations using DEVS without any modification, it appears that the addition of two particular functions dedicated to observation process can be very useful and computationally more efficient than a classic DEVS implementation. We propose these two functions, the "request function" and the "observation function" in addition to the classic DEVS specification. We provide the abstract simulators to implement them in the DEVS framework. Furthermore, we illustrate how the C++ language is well situated to rigorously implement the specification.

2 The request function

2.1 Problematic

In very communicative systems, like IBM, it is often necessary for a model to know the states of the surrounding models to change their own state. This model realize an observation on the neighbourhood. This kind of observation, which we call *request*, is easily achievable in DEVS framework and several solutions exists:

- Using the DEVS algorithms and instantaneous state, i.e. when $ta(s) = 0$, to build output value. Two ways are possible:
  - on external transition: if the observation event does not require an internal transition, it may be generated by the external function or output function. In this case, only one instantaneous state is necessary:
    1. $s1 = \delta_{ext}(s)$ receives the request and keeps the remaining time in state s: $x = ta(s) - e$ and computes an output
    2. $ta(s1) = 0$ instantaneous state to process λ
    3. $\lambda(s1)$ builds and sends an output value
    4. $s = \delta_{int}(s1)$ restores to the state before request
    5. $ta(s) = x$ restores the duration time before request, x was computed in the first step
  - on internal transition: if the observation event requires an internal transition to process a correct output, two instantaneous states are required:
    1. $s1 = \delta_{ext}(s)$ receives the request and keep the remaining time in state s: $x = ta(s) - e$
    2. $ta(s1) = 0$ instantaneous state to process δ_{int}
    3. $\lambda(s1)$ sends nothing
    4. $s2 = \delta_{int}(s1)$ computes an output
    5. $ta(s2) = 0$ instantaneous state to process λ
    6. $\lambda(s2)$ sends the correct output

7. $s3 = \delta_{int}(s2)$ restores to the state before request, where $s3 = s$
8. $ta(s3) = x$ restores the duration of state before request, x was computed in the first step

- Models share their states when a modification arrived in internal or external transitions. This technique was developed in CellDEVS [Wainer and Giambiasi, 2001].
- In his book [Zeigler et al., 2000], B. P. Zeigler defines the memoryless systems like a static system where the output is unambiguously determined by the current input and call it *function specified systems*.

$$Fnss = \langle X, Y, \lambda \rangle$$

Where:

$$\begin{cases} X \text{ is the set of inputs} \\ Y \text{ is the set of outputs} \\ \lambda : X \rightarrow Y \text{ is the output function} \end{cases}$$

In the request cases presented above, we can notice that models that receive requests must manage theirs states using internal and external transition functions. The instantaneous states, although clearly defined in DEVS are hard tasks to modeler who implements models, mainly in state models, where each state should expect to receive request events.

In this paper, we propose a formal approach to develop request interactions between models. The aim of this formalization is to simplify the development of model for the modeler by simplifying state graphs of its models.

In the next section, we propose an approach based on the $Fnss$ and a slight modification of the DEVS algorithms for the introduction of an event named *request*.

2.2 Formal specification

Starting from Fnss, we want to specify a particular DEVS output function providing the same behavior as λ for Fnss (i.e. to compute instantaneously an output without changing the current state of the model). Considering a model M in the state s, a request of the current state of M from another model is an external transition on M as follow:

$$\delta_{ext}(X, Q) = s' \mid \{Q \equiv s' \equiv (s, e)\}$$

Considering s' as an instantaneous state ($ta(s') = 0$) and $\delta_{int}(s') = s$ then we can write:

$$\delta_{int}(\delta_{ext}(X, Q)) = s$$

and

$$\lambda(\delta_{ext}(X, Q)) = \alpha \mid \{\alpha \subseteq s,\ \alpha \in Y\}$$

Renaming this particular output function we give the formal specification of the request function as:

$$\lambda_{req} : X \times Q \rightarrow Y$$

Where:

$$\begin{cases} Q = (s, e) \mid s \in S \\ X \text{ request port and associated values} \\ Y \text{ response port and associated values} \end{cases}$$

In classic DEVS, the confluence between internal and request function could be managed by the $Select$ function. In Parallel DEVS, the δ_{con} function could be used. In our implementation, we assume that request events are managed after all imminent transitions in a model. Then we do not provide δ_{con} the ability to manage requests. We ensure in the coordinator implementation that all imminent bags are fired before calling the request function. We give the algorithms in the next section.

2.3 Algorithms

In this section, we present our changes in the DEVS algorithms for coordinator and simulator. The root-coordinator was not changed.

2.3.1 Coordinator

In coordinator, we add the management of the request events *r-message* (the request messages):

```
1  Devs-Coordinator
2  variables:
3   DEVN = (X,Y,D,{M_d},{I_d},{Z_{i,d}},Select,Select_req)
4   parent // parent coordinator
5   tl // time of last event
6   tn // time of next event
7   d* // selected imminent child
8   eventlist // list of element (d,tn_d) sorted by tn_d
9           // and Select
10
11  [...] // The same algorithms than DEVS classic
12        // coordinator.
13
14  when receive r-message (x,t) at t with input x
15   if not (tl ≤ t ≤ tn) then
16    error: bad synchronization
17   receivers = {r|r ∈ D,N ∈ I_r,Z_{N,r} (x) ≠ ∅}
18   for-each r in receivers
19    send r-message (x_r,t) to r
20   sort eventlist according to tn_d and Select_req
21   tl = t
22   tn = min{tn_d|d ∈ D}
23
24  when receive r-message (y_{d*},t) at t with output y_{d*}
25   if d* ∈ I_N & Z_{d*,N}(y_{d*}) ≠ ∅ then
26    send r-message (y_N,t) to parent
27   put y_{d*} of d*
28   receivers = {r|r ∈ D, d* ∈ I_r,Z_{d*,r} (y_{d*}) ≠ ∅}
29   for-each r in receivers
30    send r-message (x_r,t) to r
31 end Devs-Coordinator
```

- l. 14: condition is activated on receipt of a request event from the parent and to broadcast into the network.
- l. 24: condition is activated on receipt of a request event from the network and to broadcast into the network.

This algorithm is the same as the *x-message* and *y-message* proposed by B. P. Ziegler. The main difference is the $Select_{req}$ function that sorts the events in order to put the request events in a buffer to consume all events produced at the same date. This function ensures the sorting of events and avoids causality problem.

2.3.2 Simulator

We define the DEVS atomic model by the tuple:

$$M = \langle X, Y, \delta_{int}, \delta_{ext}, \delta_{req}, ta, \lambda \rangle$$

```
1  Devs-Simulator
2  variables:
3   parent // parent coordinator
4   tl // time of last event
5   tn // time of next event
6   DEVS // associated model with total state (s,e)
7   y // current output value of the associated model
8
9   [...] // The same algorithms than DEVS classic
10        // simulator.
11
12  when receive *-message (*, t) at t
13   if t != tn ten
14    error: bad synchronization
15   y = λ(s)
16   send y-message or r-message (y,t) to parent
17   s = δint(s)
18   tl = t
19   tn = tl + ta(s)
20
21  when receive r-message (x,t) at t with input x
22   if not (tl ≤ t ≤ tn) then
23    error: bad synchronization
24   y = δreq(s,t,x)
25   send y-message(y,t) to parent
26 end Devs-Simulator
```

- l. 12: the classic internal event process. The change is the output function λ may returns an external or a request event to the coordinator or root-coordinator.
- l. 21: condition is activated on receipt of a request message. In this case, the δ_{req} is called and the result is broadcast to the network.

In Parallel DEVS, the *select* function is replaced by a well defined management of events, using *bags* or *mails*. To produce the same behaviour for the request transition than in Classic DEVS, we update the Parallel DEVS coordinator by adding a management of *r-message*.

3 Observation transition

3.1 Problematic

The construction of a DEVS platform for modeling and simulation (presented in section 4) required to observe models and their evolution during the simulation. Observation of DEVS models involves watching or capturing theirs states. These captures can take place during a change of a model (in a δ_{int} or δ_{ext} transition functions) or at a specific date.

In DEVS, the commonly solution, presented in figure 1, used is to connect models to observe, both input and output, to an observation model. This model have in charge to send observation messages and treatment of the models responses. For instance, to build a discrete observation, the observation model uses its ta function with a constant to send, at each time step, an observation event.

[Figure 1 about here.]

These solutions impose to the modelers to mix the state graphs between observation and behavior of theirs models. In addition, by merging the state graphs, the modeler may make the results of its models dependent to the observation he makes. For example, if a model is observed asynchronously with its behaviour, the computation of the e (in abstract simulator, $e = t - tl$ when process an external transition $\delta_{ext} : Q \times X \rightarrow S$ with $Q = (s, e)$, e the time elapsed since the last transition) or internal state variables, with the quality of real in computer engineering, can give different results with or without observations.

Finally, in DEVS, a model can have several states at the same date (when at least one call to the $ta(s)$ function returns 0), known as instantaneous states. If an observation model sends an event at a specific time to capture the state of a model to observe, this one can provide an instantaneous state and not the last state if an another model send, at the same date an external event, or in a next bag in DEVS parallel.

In this section, we propose a formalization of the observation models for the management of the state models before a change of date. We propose this change by adding an additional function in the atomic model in order to avoid mix graphs state observation and behavior.

3.2 Formal specification

The observation is specified following the same reasoning that for the request function as follow:

$$\lambda_{obs} : X \times Q \rightarrow Y$$

Where:

$$\begin{cases} Q = (s, e) \mid s \in S \\ X \text{ observation port and values} \\ Y \text{ response port and values} \end{cases}$$

The observation upon a model does not modify its current state. The difference between request and observation function relies on the algorithmic of the coordinator. Observations functions must return the state of models at time t, after all transitions

were fired at this date. If it exists one or several instantaneous internal transitions at t, the observation function must be treated after all these transitions to be sure we observe the final state at t. In the following, we give the algorithms of coordinators and simulators in DEVS.

3.3 Algorithms

3.3.1 Root coordinator

The root-coordinator implements the overall simulation loop. It sends messages to its direct subordinate (simulator or coordinator). The root-coordinator first sends an initialize message (*i-message*), and loop on internal transition (**-message*) from its child to perform the simulation cycles until some termination conditions.

```
1  Devs-Root-Coordinator
2   variables:
3    t // current simulation time
4    tl // the last simulation time
5    child // direct subordinate devs-simulator
6             // or devs-coordinator
7
8   t = t0
9   send initialization message (i,t) to child
10  t = tn of its child
11  tl = t
12  loop
13     send (*,t) message to child
14     t = tn of its child
15     if tl < t then
16        send flush-observation message (t) to child
17        tl = t
18  until end of simulation
19 end Devs-Root-Coordinator
```

In the root-coordinator algorithm, we add an additional variable tl which indicates the last date of the simulation. When a date change occurs (l. 15), the root-coordinator sends a message (*o-message*) to its direct subordinate to flush the observation event (l. 16).

3.3.2 Coordinator

The treatment of the order of observation events and the state of the simulation is required to guarantee that observation are send to all child before a change in the date of the simulation.

```
1  Devs-Coordinator
2  variables:
3   DEVN = (X, Y, D, {Md}, {Id}, {Zi,d}, Select)
```

```
4  parent // parent coordinator
5  tl // time of last event
6  tn // time of next event
7  d* // selected imminent child
8  eventlist // list of element (d, tn_d) sorted by tn_d
9          // and Select
10
11 [...] // The same algorithms than DEVS classic
12       // coordinator.
13
14 when receive o-message (x,t) at t with input x
15  if not (tl ≤ t ≤ tn) then
16   error: bad synchronization
17  receivers = {r | r ∈ D, N ∈ I_r, Z_{N,r} (x) ≠ ∅}
18  for-each r in receivers
19   send o-message (x_r) to r
20
21 when receive o-message (y_{d*},t) output y_{d*} from d*
22  if d* ∈ I_N and Z_{d*,N}(y_{d*}) ≠ ∅ then
23   send o-message (y_n,t) value y_N = Z_{d*,N}(y_{d*}) to parent
24  receivers = {r | r ∈ D, d* ∈ I_r, Z_{d*,r} (x) ≠ ∅}
25  for-each r in receivers
26   send o-message (x_r) to r
27
28 when receive fo-message (t) at t
29  for-each d in D
30   send flush-observation message (t) to child
```

- l. 14: the *o-message (x,t)* are input messages (x, y) from the coupled model. His goal is to route the observation event to the children and store observation.
- l. 21: the *o-message* (y_{d*},*t)* are output messages from children of the coupled model. His goal is to route the observation event to the children and store observation or to route observation to the output of the coupled model.
- l. 28: the *fo-message (t)* are messages from root-coordinator when the simulation date is change. The message is send to all child.

The previous algorithm shows in l. 14-30 the distribution of observation events from or to the simulators.

3.3.3 Simulator

We define the DEVS atomic model by the tuple:

$$M = \langle X, Y, \delta_{int}, \delta_{ext}, \delta_{obs}, ta, \lambda \rangle$$

We propose this algorithm for the simulator of the Atomic DEVS with the observation transition based on the Classic DEVS of [Zeigler et al., 2000]:

```
1  Devs-Simulator
2  variables:
3   parent // parent coordinator
4   tl // time of last event
5   tn // time of next event
6   DEVS // associated model with total state (s,e)
7   y // current output value of the associated model
8   obslist // list of element (x)
9
10  [...] // The same algorithms than DEVS classic
11        // simulator.
12
13  when receive o-message (x,t) at t with input x
14   if not (tl ≤ t ≤ tn) then
15    error: bad synchronization
16   add (x) to obslist
17
18  when receive fo-message (t) at t
19   for-each x in obslist
20    y = δobs(s,t,x)
21    send y-message(y,t) to parent
22   obslist = ∅
23 end Devs-Simulator
```

- l. 13: when receives an *o-message* from the parent, we just add the event into the observation list.
- l. 18: when receives an *fo-message* from the parent (originally send by the root-coordinator), we compute the observation transition and send the output to the parent and remove it to the observation list.

To develop the observation transition into classic DEVS, we add the observation event, *o-message* in algorithm (l. 13-16) to store the observation message. In l. 20, the transition δ_{obs}, activated when the simulator receipts a *fo-message* from the coordinator or root-coordinator, computes an output value y from the current state, the time and the observation event. This function guarantees that the state of the model is not modified.

The algorithms of coordinators and simulators presented above are exactly the same in the Parallel DEVS specification using a buffer of observations events in the simulator and *fo-message* from the root-coordinator when the date of simulation is changed.

4 Implementation in VLE

In this part, we describe the implementation of the observation and request transition into an operational DEVS environment: VLE.

4.1 VLE

The VLE, *Virtual Laboratory Environment*, is an environment for modeling, simulation and analysis [Quesnel et al., 2007] of complex systems. VLE is an open-source software and API (Application Programming Interface), developed in C++, oriented object, available on Unix, Linux and Windows. VLE allows to develop DEVS models, build experimental frames and run-it on grid computing. This environment proposes a DSDE abstract simulators from [Barros, 2003] and some classic DEVS extensions like Cell-DEVS, QSS 1 & 2, Cell-Qss, Petri net, Finite State Automata, etc.

The VLE DSDE abstract simulator defines an Atomic Model like a class:

```
1  class Dynamics
2  {
3  public:
4    // classic Parallel DEVS functions
5    Time timeAdvance() const;
6    void internalTransition(
7            const Time& time);
8    void externalTransition(
9            const Time& time,
10           const ExternalEventList& lst);
11   void output(
12           const Time& time,
13           ExternalEventList& out) const;
14   void confluentTransition(
15           const Time& time,
16           const ExternalEventList& lst) const;
17
18   // request transition
19   void request(
20           const RequestEvent& event,
21           ExternalEventList& out) const;
22
23   // observation transition
24   Value observation(
25           const ObservationEvent& event) const;
26 };
```

- l. 4-16: the classic Parallel DEVS functions: ta, δ_{int}, δ^b_{ext}, λ and δ_{conf}.

- l. 18-21: the request method, called when at least on *request event* arrived on input X^b. δ_{req} is a *constant* function, to prevent user to modify the state of its model. The *out* parameter is an external event list used to put external events output ports.
- l. 23-25: the observation method, called when an *observation event* arrived on a input X. δ_{obs} is also a *constant* function to prevent user to modify the state of its model. This function returns a Value (simple type as integer, real, boolean, string, and complex type as set, dictionary, matrix etc.).

With the VLE simulator, we propose a strictly implementation of the DEVS specification by using the C++ *const* keyword. This feature specifies constant variables and functions. If a function is constant then, the C++ compiler forbid the developer to modify the state of its object (a model in VLE simulator). This feature ables the VLE modeler to develop strictly DEVS models and also allows a better learning of DEVS for the new user.

4.2 Experimental frames in VLE

The study of models is very important in the modeling cycle. The experimental frames as sensitivity analysis, replicas generation, etc. are used to study the possibilities of models. In the VLE environment, we propose tools for managing the experimental frames. These tools provides:

- Define a graph of the original models, and each model atomic defines several experimental conditions and how they would be observed (We use the observation defined in section 3).
- Build instances of the experimental design by performing combinations between different experimental conditions and applying a number of replicas with particular seed.
- Execute these instances of the experimental design on the grid computation composed of workstations and/or clusters (distributed and parallelized).

Figure 2 shows the three steps of the previous point. The experimental frame is defined in an XML file. This file describes the experimentation: what type of combination ? Number of replicas ? Seed for the random number generator ?

The XML input file can be directly edited with an XML editor but it can also be generated with the tools provided by the environment. It allows to script the experimental frames process and theirs simulations on grid computing.

[Figure 2 about here.]

The VLE environment is based on a set of libraries called VFL (*VLE Foundation Libraries*). The development of programs over these interfaces is easily achievable. Thus, in order to collaborate with users of statistical tools, we provide an interface to the program R[R Development Core Team, 2006]: a tool and language for statistical computing. This package, called RVLE, has the same capabilities as the VFL and provided an easy-to-use tool to exploit and to explore model output based on VLE simulations.

5 Conclusion and perspectives

In this paper, we present the formalization of the observation in the DEVS framework. These observations are classified into two types. First, observation between models, secondly, observation for interact with experimental frames. For both types, we propose the formalisation in DEVS and the changes in the DEVS abstract algorithms.

The separation between the dynamics of the system and its observation is a fundamental issue. Similarly, the simplification of interactions between models is important. These considerations are set in a context of complex systems modeling and based on a significant number of interact entities. Moreover, the simulations are usually in the core of the process of analysis which are based on the observation of models. In this paper, we extend the DEVS formalism to ensure those two objectives.

The next step of this work is to continue the integration of this extended DEVS framework in experimental frame. The observation mechanism is the first element. The construction of an DEVS extension for IBM is as well dependent on these developments. The concept of request will help us to propose a elementary brick for the construction of models based on interactive entities.

References

F. J. Barros. Dynamic Structure Multiparadigm Modeling and Simulation. *ACM Transactions on Modeling and Computer Simulation*, 13(3):259–275, 2003.

R. Duboz, D. Versmisse, G. Quesnel, A. Muzzy, and É. Ramat. Specification of Dynamic Structure Discrete Event Multiagent Systems. In *In proceedings of Agent Directed Simulation (Spring Simulation Multiconference)*, Huntsville, Alabama, USA, April 2006.

A. Gosavi. *Simulation-Based Optimisation – Parametric Optimization Techniques and Reinforcement Learning*. Kluwer Academic Publishers, 2003.

E. Kofman and S. Junco. Quantized State Systems. a DEVS Approach for Continuous Systems Simulation. In *Transactions of SCS.*, volume 18, pages 123–132, 2001.

G. Quesnel, R. Duboz, D. Versmisse, and É. Ramat. DEVS Coupling of Spatial and Ordinary Differential Equations: VLE framework. In *Proceedings of OICMS 2005 conference*, Clermont Ferrand, France, June 2005.

G. Quesnel, R. Duboz, É. Ramat, and M.K. Traore. VLE - A Multi-Modeling and Simulation Environment. In *Moving Towards the Unified Simulation Approach, Proceedings of the Summer Simulation 2007 conference*, pages 367–374, San Diego, USA, July 2007. SCS - ACM.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. URL http://www.R-project.org. ISBN 3-900051-07-0.

M. K. Traoré and B. P. Zeigler. Experimental Frames Methodology. In *NSF Workshop on Modeling and Simulation for Design of Large Software-Intensive Systems: Challenges and New Research Directions. DLS03*, Tucson, AZ, USA, December 2003.

D. Versmisse and É. Ramat. Management of perturbations within a spatialized differential equations system. In *European Simulation and Modelling Conference*, pages 520–524, Porto, Portugal, october 2005. SCS.

G. A. Wainer and N. Giambiasi. Application of the Cell-DEVS paradigm for cell spaces modelling and simulation. In *Simulation*, volume 76, pages 22–39, 2001.

B. P. Zeigler. *Theory Of Modeling and Simulation*. Wiley Interscience, 1976.

B. P. Zeigler, D. Kim, and H. Praehofer. *Theory of modeling and simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2000.

List of Figures

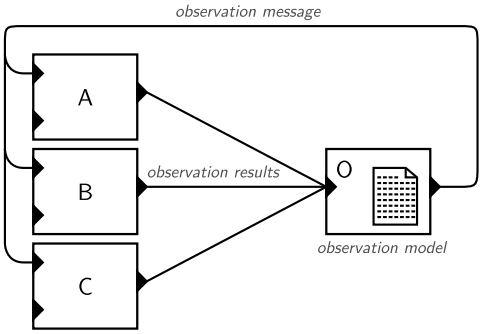

Figure 1: This picture show the classic solution to observe model. Three atomic models A, B, C, the system, are observed by the observation models O which sends observation event and wait result to store it in output file, database etc.

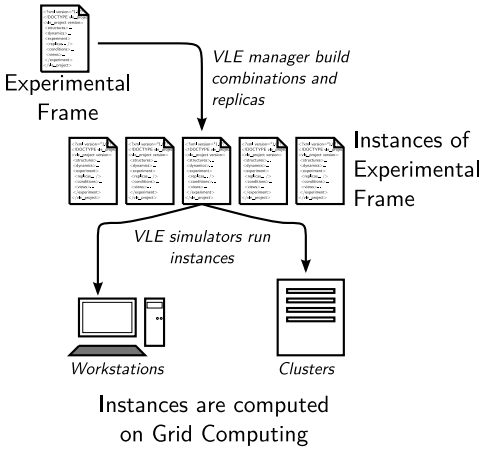

Figure 2: This picture shows the workflow of the achievement of en experimental frames in VLE.