On using Design Patterns for DEVS Modeling and Simulation Tools

M. Hamri and L. Baati
LSIS UMR CNRS 6168
Université Paul Cézanne

ABSTRACT
Modeling and simulation tools are more and more closed to the software engineering capabilities. Since decades modeling and simulation frameworks try to take advantage of software engineering evolvement such as functional programming, logic programming, object oriented programming. Moreover, since 1990s, design patterns as a new paradigm of object oriented programming tools, becomes an efficient solution to request some design and development issues. Each one provides a way of programming, resolving modeling and simulation issues usually related to specific domains. We are interested in discrete event modeling and simulation paradigm, especially in DEVS (Discrete Event System specification) which is a sound mathematical based framework with a hierarchical representation. DEVS based modeling and simulation frameworks integrate design patterns in designing and building models in specific domains in order to take advantage of their capabilities. This paper describes the use of design patterns in the modeling and simulation implementation tool. We describe how the design patterns can be utilized inside the DEVS abstract simulator in order to facilitate the reuse of DEVS entities.

1. INTRODUCTION

The modeling and simulation methods and tools become obviously crucial in analysing, diagnosing and representing complex dynamic systems. Several proposed formalisms depend of the models representation or especially of modelling and simulation expert interests (linear, non linear, discrete time, real time, discrete event, etc.). These last decades, discrete event modeling formalisms emerge as a practical way to represent, analyse and diagnose complex dynamic systems (Cassandra and Lafortune 1999) (Zeigler et al. 00). In this research, we are interested in DEVS (Discrete EVent System specification) formalism, a mathematically sound framework (Zeigler 1976) that was introduced in 1976 by Zeigler. It provides two systems representing ways; the basic/atomic model which presents the high resolution level of the system dynamic behavior, and the coupled/network model which describes the interaction between different submodels (Zeigler 1984).

The DEVS formalism as described in (Zeigler et al. 00) supports modelling of discrete event systems in hierarchical and modular manner. Nowadays, DEVS based frameworks utilize the object-oriented programming languages (LSIS-DME (Hamri and Zacharewicz 2007, Baati et al. 2007a), DEVSJAVA (ACIMS 2001), CD++ (Wainer 2002), JDEVS (Filippi and Bisgambiglia 2004), VLE (Ramat and Preux 2002), etc.) in order to take advantage of their capabilities (encapsulation, inheritance, modularity, reusability, maintainability, flexibility, fidelity to the real world, etc.). The Object-oriented languages evolve to more flexibility by providing to express several design patterns capabilities adapted to specific situations.

Design patterns represent a high level well-defined way of thinking that can be applied to a recurrent and specific problem in order to give a high efficiency solution. The pattern describes deeply the appropriate abstract solution, which can be applied to this problem/issue in several different contexts with the same success (Alexander 1977). Design patterns are a generic solution adapted in many domains (architecture, anthropology, art history, design, software engineer, etc.). It becomes famous through the work of Christopher Alexander on the pattern language (Alexander 1977), where he explains that each pattern represents a recurrent "problem/issue" in the environment.your own material.

The GOF (Gang Of Four) introduces the design patterns concept into the software engineering domain, especially in the object oriented modeling concept (Gamma et al. 1995). Design patterns provide an intermediate way of solving software problems, it allows a high level description of a sub-process that can be validated by different experts. They provide a common vocabulary for designers to use, communicate, document, and explore design alternatives (Gamma et al. 1995). According to the advantages of reusing high level solution of design patterns, the modeling and simulation approaches try to give more reusable and modular models. They aim to reach a high level representation of a complex system, which can help including more ideas of heterogeneous participants. According to (Ferayorni and Sarjoughian 2007), the use of design pattern provides the capability to the end models to become more flexible, and easily evolve to the environment needs/requests.

Some efforts were conducted by DEVS community to embed design patterns into DEVS modeling and simulation tools in order to enhance the domain specific models such as well depicted in (Ferayorni and Sarjoughian 2007, Jammalamadaka and Zeigler 2007). Some related efforts were conducted to propose a way of introducing design patterns into DEVS modeling and simulation process in order to enhance the end models and their interoperability (Dalle and Wainer 2007). Indeed, using design patterns in a modeling and simulation environment for a specific domain improves software development by speeding up software design specification. Hence, this strategy minimizes the software development effort, and related maintainability (Ferayorni and Sarjoughian 2007). These efforts are focused on how to enhance DEVS models in a specific domain. Our work consists of including design patterns into the DEVS modeling and simulation tool kernel. In this paper we present how to take advantage of the design patterns philosophy to increase reusability and facilitate the models and simulators implementation.

This paper presents a DEVS review in section 2. Section 3 gives an overview on design patterns and section 4 arguments their use in discrete event simulation. In sections 5 and 6 we describe our DEVS design patterns. Finally, we conclude in section 7 and expose our future work.

2. DEVS REVIEW

DEVS (Zeigler 1976) is a modular formalism for deterministic and causal systems' modeling. A DEVS atomic model has a continuous time base, inputs, states, outputs and functions (output, transition, lifetime of states). Larger models are built from atomic models connected together in a hierarchical fashion. Interactions are mediated through input and output ports. That allows for modularity. We propose a related approach that supports variable structure model, and preserves the DEVS formalism.

2.1 Formal specification of an atomic DEVS model

$$AtomicDEVS = \langle X, Y, S, \delta_{\text{int}}, \delta_{ext}, \lambda, lt \rangle$$

- The time base is continuous and not explicitly mentioned: T= IR.

- X is the set of (external) inputs of the model. They interrupt its autonomous behavior by the activation of the external transition function δ_{ext}.
- Y is the set of outputs.
- S represents the set of sequential states.
- δ_{int} is the internal transition function, allowing the system to go from one state to another autonomously.
- λ is the output function.
- lt(s) is the lifetime function.

The system reaction to an external event depends on its current state, the input value and the elapsed time.

2.2 Formal specification of a coupled DEVS model

The coupled DEVS formalism describes a discrete events system in terms of a network of coupled components.

$$CoupledDEVS = \langle X_{self}, Y_{self}, D, \{M_d / d \in D\},$$
$$EIC, EOC, IC \rangle$$

Self stands for the model itself.
- X*self* is the set of possible inputs of the coupled model.
- Y*self* is the set of possible outputs of the coupled model.
- D is a set of names associated to the model components, self is not in D.
- $\{M_d/d \in D\}$ is the set of the coupled model components, with d being in D. These components are either atomic or coupled DEVS model.
- EIC, EOC and IC define the coupling structure in the coupled system.
- EIC is the set of external input coupling, which connects the inputs of a coupled model to components inputs.
- EOC is the set of external output coupling, which connects the outputs of a coupled model to components outputs.
- IC defines the internal coupling, transforming a component output into another component's input within the coupled model.

2.3 Simulation

The interpretation of the behavior of a DEVS model is given with the DEVS conceptual simulator. It consists of processors that represent a root coordinator, coordinators that are associated to DEVS coupled models and basic simulators that simulate the behavior of DEVS atomic models.

A set of messages are exchanged between processors, grouped into rising and falling messages. These messages are:

- i-message that activates the init state and actions,
- x-message that informs the simulator about an external event arriving and allows the fire of an external transition,
- *-message is a scheduled message according to the life time of the current state that causes the execution of the output function and the corresponding internal transition change,
- y-message that specifies the output function result, and
- d-message that expresses the fact that x-message or *-message was handled by the simulator.

To enhance the simulation process, (Kim et al., 2004) and (Zacharewicz et al., 2005) proposed to flatten the simulation processor by transforming DEVS hierarchical models into non-hierarchical ones.

3. Design patterns in software engineering

3.1 Definition

A pattern is a solution to a recurrent design problem. A pattern gives us guidelines to resolve a problem in particular context. The GOF popularized this concept when they edited the first book in the field "Design Patterns: Elements of Reusable Objects-Oriented Software" (Gamma et al., 1995). Then, many scientific books and research works were born. They try to formalize software development experiences through frameworks called design patterns.

A pattern resolves a technical problem at conceptual level. Details were discussed in the solution but they do not constitute its heart. To illustrate the design patterns concept, let us consider the pattern Observer as discussed in (Gamma et al., 1995).

Name: Observer pattern.

Intent: define a one-to-many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically.

Also known: Dependants, Publish-Subscribe

Structure:

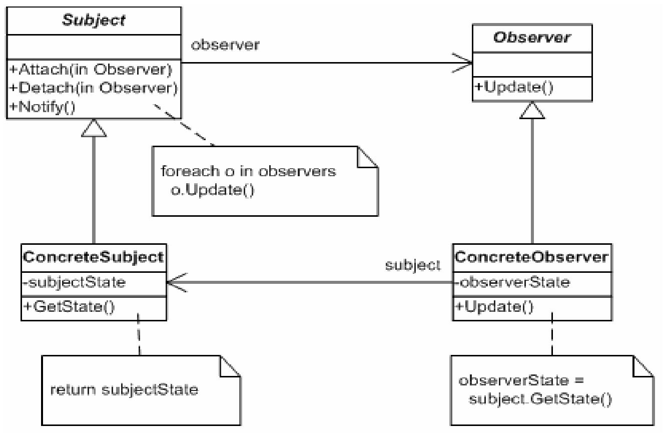

Figure 1. The observer pattern

Participants:

Subject: provides a method that notifies all observers. It should know its observers.

Observable: defines the interface update() for object that should be notified of changes in a subject.

ConcreteSubject: is a concrete class that extends the class subject to allow storing data that interest the set of concrete observers. It sends a notification to inform the concrete observers about an occurred change state. Also it holds the list of observers and it is responsible on updating this list.

ConcreteObserver: it maintains a reference on a ConcreteSubject object, stores the data of the observable object that should be consistent with its data.

We remark for this pattern, the authors use an organized structure to expose it. This structure is a form of language pattern that designers use to formalize their patterns.

4. Why we should use design patterns in discrete event simulation?

1) A summarized description

Due to many software developments in discrete event M&S, the design pattern language could be an alternative for documenting M&S applications without technical details. We know all for viewing a technical document of a software tool is a hard task and consumes an important time. The reader must look at this document (often hundreds of pages) to realize the underlying modification which is difficult to identify the concerned module and whether it is possible to implement this modification.

For instance a motivation simulationist is to conduct a simulation with an animation of the DEVS model. One of the possible solutions is to link for each state of model an icon and change the current one when a state change occurs. However the state of DEVS model in almost tools is mapped to the integer type. This fact pushes the programmer to amend the code which structured through *if-else* statement and inserts other statements to make the animation. Hence, a DEVS model coded using the state pattern makes easy the request animation. It is realized by extending the class *State* to *StateWithIcon* in which we add the attribute *Icon*. Still to redefine the method that realizes the state change to make also the change of the icon according to the current state.

2) A well-known vocabulary

Increasing communication between M&S developers by using a common vocabulary to share experiences structured through a design pattern language. For instance the simulation in DEVS is conducted using two techniques: 1) the hierarchical simulator that inherits its structure from modeling design (DEVS models) or 2) the flattened simulator which consists of only the root coordinator, one coordinator and the simulators. Therefore we can imagine the dialog below between two simulation developers in design patterns:

Simulationist 1: I want to speed up the simulation
Simulationist 2: did you use a hierarchical simulator pattern
Simulationist 1: yes!
Simulationist 2: use the flattened simulator pattern and see the result.
Simulationist 1: a good idea, I completely forgot it.

Imagine now the same discussion without skills on design pattern or simulation vocabulary.

Simulationist 1: I want to speed up the simulation process
Simulationist 2: did you optimize the simulation by reducing the number of messages sending/receiving messages between coordinators.
Simulationist 1: how I can do it?
Simulationist 2: ..... !

We can remark the difference between the two discussions above as follows: in the first discussion the simulationists communicate easily and without ambiguous. In fact the *simulationist 1* identifies the problem and gives the adequate solution in brief time. However the second discussion is ambiguous and no issue is proposed. Thus we conclude on the importance of the vocabulary.

3) A catalogue of design patterns

Providing M&S developers with a catalogue of design patterns that handle each particular M&S design problem in a particular context, should reduce the cost of development. For instance a flyweight pattern could be combined with the DEVS hierarchical simulator to optimize its structure and reduce the heap memory. The major difficulty is to have a consensus in naming the proposed patterns and making them known and available to the developers. *A task requiring enough time!*

4) Re-using formal design solutions

By extending the use of design patterns in M&S, designers and developers re-use validated design solutions instead of code in formal way. In fact there are a lot of works that show how the design patterns could be formalized due to their informal representation based on text and graphics. The works of (Taibi et al., 2003) propose a balanced approach consisting on supplying possible design patterns with a formal semantics as they showed it for the observer pattern. Therefore, developers could re-use solutions tested, verified and validated by means of formal methods.

Thus, we believe that M&S design patterns could be employed to further software developments of new formalism extensions as we will show it in later sections.

Knowing that we discover design patterns through software developments and we are able to invent them, we believe that GOF design patterns could be extended and/or generalized to satisfy M&S requirements.

5. Patterns of GOF to design DEVS models

Recall that patterns are not invented but discovered via coding experiences, we will give in this section and following one issues of design DEVS models and the simulation kernel without evoking detail of coding (thread, event queue, etc.). In fact we would obtain abstract diagrams of classes documented using pattern language. However if the proposed design pattern should be detailed to solve the problem, we should include the required details in the solution.

In the literature of design patterns related to design behavior, we found several research works and papers that could interest developers. These patterns handle different problems from a technical point of view like structuring the code, produce a maintainable code, increase confidence in code by proposing a clear mapping of behavioral concepts into object programming paradigm, etc. Henney (Henney 2003) addresses a list of patterns from the literature that implement behavior. Each one answers to a particular requirement (see the table below).

Name	**Problem**	**Solution**
Collections for states	A number of objects are managed and held in a collection, and operated on according to their common state. What is a suitable expression of the state with respect to each object?	Represent each state of interest with a separate collection that refers to all objects in that state. State transitions become transfers between collections.
Double Dispatch	How can you select a method based on the type of the target and the type or value of one other variable without hardwiring the selection as a conditional statement?	Delegate the selection of the actual method via a helper object that then calls back on the main object. The type of the helper object determines which method is selected. The helper object is normally the other variable in the interaction.
Flags for states	How can an object significantly change its behavior for only a couple of methods based on only one or two alternative internal states?	Represent the behavioral state of the object explicitly using a flag. In each of the history-sensitive methods, use a conditional to check the flag and act accordingly.
Objects for states	How can an object significantly change its behavior, depending on its internal state, without hardwired multi-part conditional code?	Separate the behavior from the main class, which holds the context, into a separate class hierarchy where each class represents the behavior in a particular state. Method calls on the context are forwarded to the mode object.

Table1. Design patterns known in software engineering for finite state machine

According to our requirements that consist on getting a structured code, easy to maintain and reusable for other simulations, we privilege the objects for states pattern.

However, the design patterns of GOF include a version of this pattern under name state pattern. So we recall this behavioral pattern in the next section.

5.1 The state pattern

Usually the state pattern is adopted to code automata. This pattern suggests coding every state of the automaton with class. This is very useful in a way that programmers use the statement if-then, switch case or matrix to code automata. With these statements, the code is centered in a piece of code that is difficult to read and maintain.

The state pattern, a well-known one to code finite state machine, suggests to code states with classes in which variables and methods are defined. However transitions are coded with methods and the code assimilates an event occurrence to the corresponding method calls. The manager (main program) has the responsibility to call a correct method of the current state when an event occurs. In (Gamma et al., 2005) the authors discuss some variant of the state pattern and gives advantages and lacks of every one. The common feature is that states are mapped into classes and transitions into methods. All made state classes inherit from an abstract state class. Furthermore methods describe transitions by returning an instance class that represents the future state.

Now we conduct some modifications to extend the state pattern to

Context: How we implement a DEVS model using the state pattern?

Problem: The output function of DEVS λ depends on the kind of state. It is defined only for active states.

Solution: The state pattern may be used in its basic form to implement a DEVS behavior. Only a slight modification should be taken into account and which consists on extending the class *state* to two classes *ActiveState* and *PassiveState*. This extension allows to code output function with a method put in the class *ActiveState* and excluded from the class *PassiveState.*

Inconvenient: The behavior is centralized on states only. However transitions are a part of the behavior.

5.2 The state/transition pattern

The state/transition pattern consists on the state one but instead of including transitions in state classes, it proposes to map transitions with independent classes. The benefit is that we get a code for DEVS behavior more clear than using the state pattern.

Let us now detail the state/transition pattern.

Name: state/transition pattern

Problem: DEVS encapsulates behaviors in atomic model thanks to the concept of port. This concept allows to two or more DEVS models to communicate via exchanged events. However the state pattern suggests to code events with methods. Considering event occurrences with method calls does not allow any handling on events. That is to say, when an event occurs we could not verify if the event belongs to the domain of a specified port before causing any firing of transition.

Solution: An object representation of events can solve the problem. This leads to code also transitions with classes. This fact has an impact in simulation performances but we get a clear code easy to read and maintain which is our first concern. In the following we discuss how DEVS elements should be implemented.

State: the class *State* is an abstract one. It possesses common attributes: *Name* to identify a state of the DEVS specification, and *Duration* to express the life time of the corresponding state *D(s)*. If the user extends the state variable s to other state variables $s= s_0x \quad xs_i \quad xs_{n-1}xs_n$, the class *State* stores those variables with their types or domain as attributes.

Event: the class *Event* implements input and output events ($x \in X, \ y \in Y$). By creating instances of the class Event we create occurrences of events. The attribute *Date* stores the occurrence time of an occurred or a planned event. Knowing that an event carries an informational message noted in the form of symbols, character, integer, etc. are possible entities to define such symbols. Thus we use an object variable to implement the event value, and the user should specify the type of event to be more precise in code; or may use particular methods to cast these events in the desired type.

Transition: the class *Transition* implements a transition of DEVS in abstract form. It centralizes common features of internal and external transition functions ($\delta_{\text{int}}, \ \delta_{\text{ext}}$):

- The variable *TargetState* identifies the future state when a transition of DEVS is fired (it may be useful to add a method that returns this state to respect the well-practices of programming).
- The method *guards* which is a Boolean, should be verified before firing the target transition.
- The method *actions* that describes the basic operations on state variables. This method is called when a valid transition is being fired. Note that actions are not output events which their occurrences depends on states (life time).

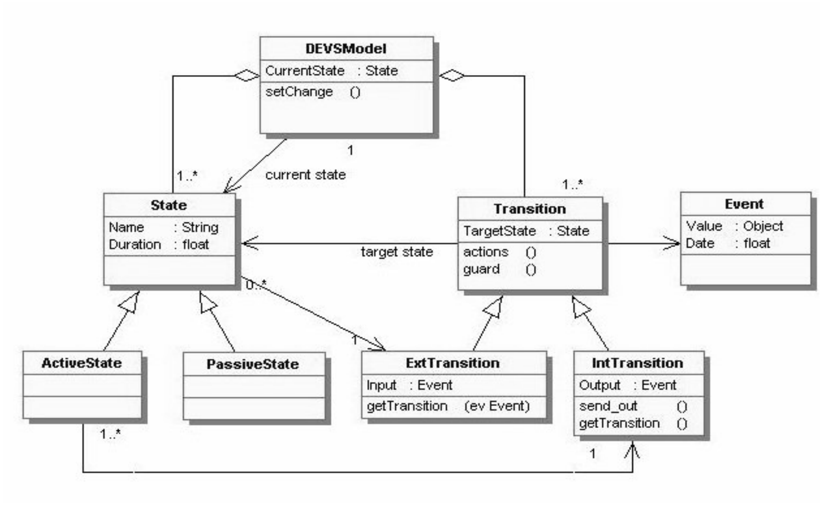

Figure 2. State/transition pattern expressed with UML

-) simulation is delayed due to the fact that transitions are not fired immediately but the simulator should identify the firing transition.

--) the size of heap memory is more important due to the instances to manage for each state and transition.

5.2.1 Applying the state/transition pattern for DEVS atomic

In DEVS, behavior is encapsulated through atomic models. The port concept allows interactions with other models. The encapsulated model receives events via input ports and sends out other events through output ports. To design these atomic models, we should keep the specified behavior through the state/transition pattern (figure 2) and insert to it the class port to encapsulate behavior and allow reuse of models.

Based on these rules of mapping and statements, we obtain the pattern shown in figure 1 for DEVS. This pattern extends the patterns discussed in the literature to implement finite state machines (mapping transition with class, event with class, etc.). These extensions are conducted to allow the implementation of DEVS due to its particular features (time, autonomous changes, actions, etc.). We remark that two relations exist in the proposed class diagram. The first one associates a state to many classes *ExtTransition* and constraints a class *ExtTransition* to be associated to only one class *State*. However the second relation is a DEVS constraint that recalls the programmer for each class *ActiveState*, he should define at least the own class *IntTransition*.

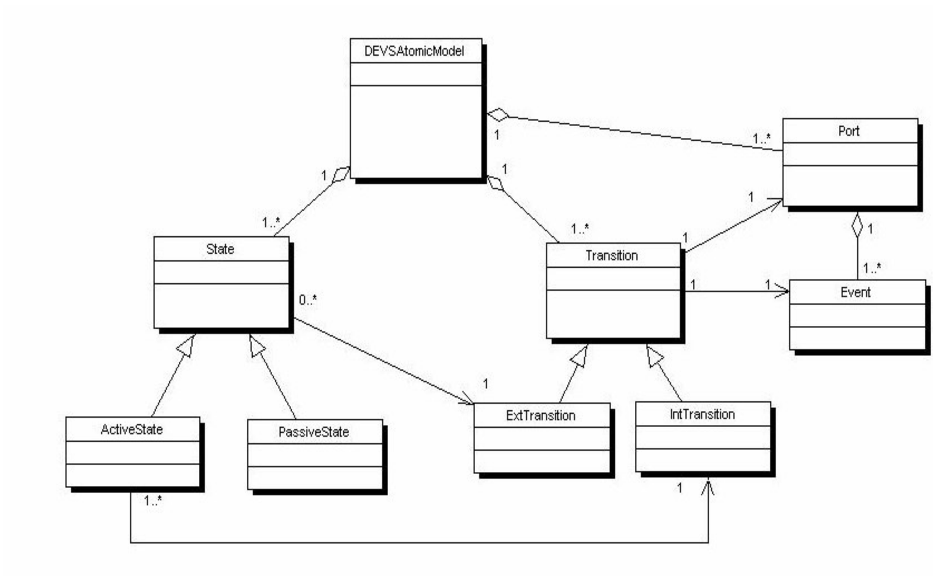

Figure 3. DEVS atomic design pattern

Advantages:

+) get a safe code easy to maintain when the modeler modifies the definition of its models by adding or deleting states and transitions in the mathematical (conceptual) definition. So modifications can be conducted in structured way.

++) modifications can be conducted during simulation to satisfy requirements of DEVS models that change structure. That means new behaviors can be defined by updating only the code at real time.

Inconvenient:

5.2.2 Design DEVS coupled models

Making a DEVS coupled model consist on reusing defined DEVS model that the user saved in top up way. Next he defines connections (oriented ones) among reused DEVS models by specifying external input, external output and internal couplings (EIC, EOC and IC respectively).

All the elements of DEVS coupled appear on the class diagram. The ports are mapped into class *Port* within references to other ports. In fact, these references define the possible couplings EIC, EOC and IC. The class DEVS coupled consists of other classes that implement DEVS coupled and atomic models. The class *AbstractModel* plays the role of intermediary allowing saving them in a unique object (vector, list, etc.) by the cast technique. Still the last element, the function select which is encapsulated like a method in the class DEVS coupled and should define the priority between DEVS components at the same level with the same parent.

Name: DEVS coupled model

Context: How to design a set of DEVS components inter-related via port

Problem: first get only the structure of the DEVS coupled model, the dynamics will given by the simulator. So only the data and functions (select) are designed at this level. Secondly get the recursive form of the structure in which a DEVS component consists of other ones.

Solution: the composite pattern perfectly fit to the problem of DEVS coupled. We have to supply this pattern with some amendments to get the entire definition of a DEVS coupled model.

AbstractModel: is an abstract class to refractory the common attributes of DEVS atomic and coupled that consists of ports.

DEVSCoupled: a class that gets the subcomponents which are atomic and new coupled.

DEVSAtomic: defined above.

Port: an abstract class that should be extended to create input and output ports. The diagram shows how the different couplings IC, EIC and EOC are conducted on port classes.

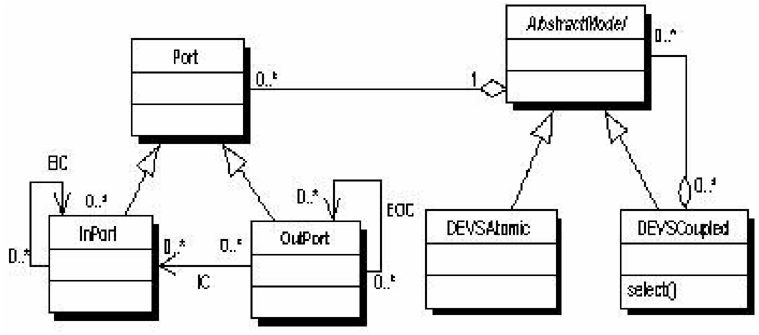

Figure 4. DEVS coupled design pattern

6. Patterns of GOF to design DEVS simulator

The observer pattern of GOF is a behavioral one that consists of a subject (observable) and observers. The subject is an observable entity by a list of observers. If observers register to get information from a subject, he must notify this information once its state change is confirmed. Otherwise, the observers ignore the sent information. In the state of the art of design patterns, we find several applications that adopted this pattern. A well-known application is the MVC (Model-View-Control) that provides a distribute architecture to get independent components safe and easy to maintain. Therefore, we believe that the observer pattern fit to the classic DEVS simulator in which the interaction between the root, sub-coordinators and simulators consists on sending and receiving events. Another advantage is enhancing and making easy the design of further DEVS extension tools.

Designing the DEVS simulator of Zeigler in its first version using the observer pattern, leads to an interesting design in which the notification of events is conducted in inverse-V way. In addition the components in DEVS simulator are at once subject who should notify events and observer who listens notified events from subjects. Consequently the number of events to manage is important due to the tree structure of the simulation. So we should take care when we design the DEVS simulator to avoid synchronization errors that could be fatal on the simulation and causing behavior errors. These problems will be noted and discussed in our solution design.

To design the abstract DEVS simulator of Zeigler we should combine the observer and composite patterns. The composite pattern keeps the hierarchical structure of the simulator which is a tree. At high level represented with the root will correspond to the root coordinator, then the sub-coordinators fit to the nodes of the tree. At the low level, the leafs correspond to atomic simulators. In conclusion, the composite pattern well fit to hierarchical structure of the abstract simulator. It remains to specify the messages exchange through the different nodes. This is possible, by making each node as a subject observed by the parent to get the done- and y- events in case of an upcoordinator or only the done-event in the case of root coordinator; and by child (sub-coordinators and/or simulators) to get the x-, i- and *- events. Thus, we remark that each coordinator has two observers: the parent and sub-coordinators that should listen and react to each event arriving from the subject. However, the abstract simulator classifies the notified events into two classes: up and down events. The first class consists of x-, i- and *- events and the second one consists of done- and y-events.

This statement allows the observers to handle correctly the received events by the cast technique according to the event classes noted above. Therefore, we design the following pattern for the abstract DEVS simulator of Zeigler.

Using this architecture, the elements of the simulator root: sub-coordinators and simulators are independent components. Each component communicates with the others by exchanging events instead of calling their services (methods). Since a component is notified it should react by executing the corresponding (private) method which is implemented inside and called through the method update(). Therefore, the implementation of each component is encapsulated in its own class. Consequently we obtain a safe code that guaranties the absence of method call conflicts. We note also that this architecture could be easily extended to allow distribute simulations. By regarding the web services patterns, the author proposes a version of the observer pattern for web services (Monday 2003). Therefore, the proposed DEVS simulator could be dispatched through the web with little modifications.

A description of this design pattern using the pattern language leads to the following specification:

Name: Hierarchical DEVS simulator

Problem: How to implement a hierarchical DEVS simulator with respect to the abstract of Zeigler in which the structure is organized in form of tree and its elements (root, coordinators and simulators) communicate through sending/receiving messages.

Solution: Combine the composite pattern with the observer one to get the structure of the simulator and insure the synchronization mechanism respectively.

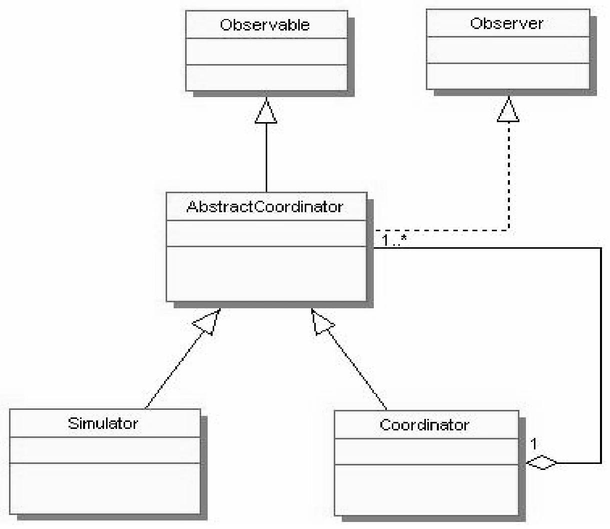

Figure 5. The abstract simulator using observer pattern

Advantages: The structure of the abstract simulator is completely distributed through different classes that code the simulator elements. Each class manages the received events, execute the related instructions and sends the appropriate output events. Moreover this structure could be updated during the execution of simulation (except the root class) in case of Dynamic DEVS in which the DEVS model in which the DEVS model changes by deleting and/or adding nodes on the simulator structure. This pattern could constitute a basis for extension of DEVS in which we should consider specific requirements (PDEVS, GDEVS, etc.).

The wide well-documented and used design patterns, is well known by developers and provide easier communication between utilized objects. The observer design pattern became a standard and it is well known by developers. The proposed DEVS based design pattern provides a generic way of thinking, that can be adopted by DEVS community to design and perform their specific models. Otherwise, this proposed DEVS based design pattern is flexible and could be adapted to the specific requirements of the wide and diverse DEVS community (dynamic hierarchical DEVS structure (Baati et al. 2006), Cell-DEVS (Wainer 2002), PDEVS (Chow and Zeigler 1994), DSDE (Barros 1998), ρDEVS (Uhrmacher et al. 2006), dynDEVS (Uhrmacher 2001), etc.).

7. Conclusion

In this paper we proposed an approach to conduct DEVS models and simulators based on design patterns paradigm from software engineering. In fact we chose the observer an composite patterns to design the DEVS conceptual simulator of Zeigler and we propose generalized patterns to design DEVS behavior. Thanks to its event-oriented architecture, the sending/receiving process of messages is clearly and well designed according to simulator requirements. The exchanged messages are mapped into object messages and not with methods calls which is a limited solution to resolve complex and growing requests. By using the Observer pattern, the components of the designed simulator are less coupled due to the fact that the coupling is realized at run-time through the exchange of messages except the first coupling to define the set of observers (call the method *addObservers()*).

Therefore, two main advantages induce from this pattern for the design of DEVS simulator. Firstly, the implementation of simulator components is encapsulated through classes that communicate through the exchange of messages. Secondly, the structure of the simulator could be updated at run-time. This point allows adding and deleting sub-coordinators and/or simulators. This facility is useful to design simulators for DEVS models with variable structure. The characteristic of these models is that they modify their own structure dynamically once particular events occur. These events lead the model to change its structure. Consequently, the corresponding simulator should be updated according to the current structure of the target model instead of re-constructing this simulator from the beginning. This should be shown in the near future.

Our future direction is to propose a DEVS toolkit (platform) easy to re-use by designers developing further simulators of

DEVS extensions. In fact we imagine a toolkit in form of a DEVS kernel to which the DEVS community turns. Still to develop the solution by combining design patterns or discover new ones. That is the way we specify to reduce the DEVS-based and extended DEVS-based tools gap.

REFERENCES

(Alexander 1977) Alexander, C. 1977. A Pattern Language. Oxford University Press.

(Baati et al. 2006) Baati, L., C. Frydman, and N. Giambiasi. 2006. Simulation Semantics for Dynamic Hierarchical Structure DEVS Model DEVS06. In Proceedings of DEVS'06-SpringSim'06. Huntsville Alabama.

(Barros 1998) Barros, F. 1998. Multimodels and Dynamic Structure Models: An Integration of DSDE/DEVS and OOPM. In Proceedings of the 1998 Winter Simulation Conference: pp. 413-419.

(Cassandra and Lafortune 1999) Cassandra, C.G., and S. Lafortune. 1999. Introduction to discrete event systems. Kluwer Academic Publishers.

(Chow and Zeigler 1994) Chow, A. C., B. P. Zeigler. 1994. Parallel DEVS : A parallel, hierarchical, modular modeling formalism. In Winter Simulation conference Proceedings. Orlando. Florida.

(Dalle and Wainer 2007) Dalle, O., G. Wainer. 2007. An Open Issue on Applying Sharing Modeling Patterns in DEVS. In Proceedings of the 2007 summer computer simulation conference. Article No.7.

(Ferayorni and Sarjoughian 2007) Ferayorni, A. E., H. S. Sarjoughian. 2007. Domain driven simulation modeling for software design. SCSC 2007: 297-304.

(Filippi and Bisgambiglia 2004) Filippi, J. B., P. Bisgambiglia. 2004. JDEVS: an implementation of a DEVS based formal framework for environmental modelling. Environmental Modelling and Software 19(3): 261-274.

(Gamma et al. 1995) Gamma, E., R. Helm, R. Johnson, J. Vlissides, 1995, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

(Hamri and Zacharewicz 2007) Hamri, M., G. Zacharewicz. 2007. LSIS_DME: An Environment for Modeling and Simulation of DEVS Specifications. AIS-CMS International modeling and simulation multiconference. Buenos Aires - Argentina.

(Henney 2003) K. Henney March 2003. Methods for states: A Pattern for Realizing Object Lifecycles. In Viking plop conference.

(Jammalamadaka and Zeigler 2007) Jammalamadaka, R., B. P. Zeigler. 2007. A Generic Pattern for Modifying Traditional PDE Solvers to Exploit Heterogeneity in Asynchronous Behavior. PADS 2007:45-52

(Monday 2003) Monday, P. B. 2003. Web services patterns: java edition. APress.

(Quesnel et al. 2008) Quesnel, G., R. Duboz and É. Ramat. 2008. The Virtual Laboratory Environment - An Operational Framework for Multi-Modeling, Simulation and Analysis of Complex Systems. Simulation Modeling Practice and Theory.

(Ramat and Preux 2002) Ramat, E., P. Preux. 2002. Virtual laboratory environment (VLE): a software environment oriented agent and object for modeling and simulation of complex systems. Simulation Modeling Practice and Theory.

(Sarjoughian and Zeigler 2000) Sarjoughian, H. S., and B. P. Zeigler. 2000. DEVS and HLA: Complementary paradigms for M&S. Transactions of the Society for Computer Simulation. 17 (4): 187-97.

(Sarjoughian and Zeigler 1998) Sarjoughian, H., Zeigler, B. 1998. Devsjava : Basis for a DEVSbased collaborative ms environment. In Proceedings of 1998 SCS International Conference on Web-Based Modeling and Simulation. volume 5: p 29.36. San Diego, CA

(Uhrmacher 2001) Uhrmacher, A.M. 2001. Dynamic Structures in Modeling and Simulation – A Reflective Approach. ACM Transactions on Modeling and Simulation. Vol.11. No.2: 206-232.

(Uhrmacher et al. 2006) Uhrmacher, A. M., J. Himmelspach, M. Röhl, and R. Ewald. 2006. Introducing variable ports and multi-couplings for Cell biological modeling in DEVS. in Proceedings of the 2006 Winter Simulation Conference. IEEE.

(Taibi et al., 2003) Toufik Taibi, David Ngo Chek Ling: Formal specification of design pattern combination using BPSL. Information & Software Technology 45(3): 157-170 (2003).

(Wainer 2002) Wainer. G. 2002. CD++: a toolkit to develop DEVS models. in Software, Practice and Experience. Wiley. vol. 32 No.3: pp.1261-1307.

(Zeigler 1976) Zeigler B. P. 1976. Theory of Modelling and Simulation. Wiley & Sons. New York. NY.

(Zeigler 1984) Zeigler, B. P. 1984. Multifacetted Modelling and Discrete Event Simulation. Academic Press. London.

(Zeigler et al. 2000) Zeigler P. B., T. G. Kim, and H. Preahofer. 2000. Theory of Modeling and Simulation, New York, NY, Academic Press.