An Open Issue on Applying Sharing Modeling Patterns in DEVS

Olivier Dalle
MASCOTTE project-team
I3S-CNRS/INRIA/Université de Nice-Sophia Antipolis
B.P. 93, F-06902 Sophia Antipolis Cedex, FRANCE.
E-mail: Olivier.Dalle@sophia.inria.fr

Gabriel Wainer
Carleton University Centre on Visualization and Simulation (V-Sim)
Dept. of Systems and Computer Engineering
Carleton University, 1125 Colonel By Drive
Ottawa, ON., K1S 5B6 CANADA
E-mail: gwainer@scs.carleton.ca

Keywords: Shared Components, DEVS, Component-based Modeling, Discrete Event Simulation, Systems Theory

Abstract

This paper presents a discussion on the use of the DEVS formalism with the goal of studying how DEVS (or one of its derivatives) could support sharing modeling patterns. We address this open issue and present a discussion of this field that can be addressed by the DEVS community. On one hand, we describe the benefits and possible uses of sharing sub-components for component-based modeling; on the other hand, we explain why supporting such shared sub-components conflicts with the DEVS formalism.

1 INTRODUCTION

The DEVS formalism[1] provides a hierarchical and modular modeling mechanism, which tends itself to reuse and interoperability. Here, we introduce and discuss (with respect to the DEVS formalism) a new construction for hierarchical modeling called "shared component". In order to illustrate the benefits and usefulness of such a construction, we discuss three different *modeling patterns* in which this construction is applied.

Modeling patterns are inspired from the (Software) Design Patterns of Gamma et al.[2]: a modeling pattern describes a generic modeling case for which a generic modeling recipe may be applied. Identifying modeling cases has two benefits: (i) it provides a common basis of reflexion to the community in order to find best modeling practices for particular modeling cases and (ii) it provides a set of best of modeling practices to the practitioners when they face a modeling case that matches a well known modeling pattern.

In a hierarchical component model, a shared component is a component *instance* that have more than one parent in the component hierarchy. In comparison with the existing terminologies and modeling constructions, component *sharing* must not be confused with component *reusing*. Reusing roughly correspond to the idea of making a copy: every time a component is reused, it has its own new, independent internal state. Hence, using the object oriented terminology, reused components correspond to different *instances* of a same *class*. On the contrary, sharing correspond to the idea of making an alias: every time a component is shared, it uses the same identical internal state. Using the object oriented terminology, shared components correspond to *references* to a unique *instance* of a given *class*.

This sharing feature is interesting because it allows new modeling patterns, such as the *proxy*, *shortcut* or *matriochka* modeling patterns later described in this paper. Very few component models do effectively support this sharing feature: the Fractal component model[3] does explicitly support sharing while some others, like JainSLEE[4] provide proxying techniques which is a practical way of implementing sharing. In this paper, we discuss the use of DEVS for this purpose.

The DEVS formalism is a hierarchical, component oriented formalism used for the modeling and simulation of systems, according to the principles of the Systems Theory. Most component models, including DEVS, supports reusing. However, as we first explain in this paper, formally, the standard DEVS definition does not allow for the sharing feature. Therefore, our goal is to describe some of the potential benefits of sharing components and following, to raise the open issue of finding the best way to integrate this new feature in DEVS or one of its derivatives.

In the following, we first show why the sharing construction is not permitted by the standard DEVS definition. Then we will present three new modeling patterns for which sharing appears to be useful.

2 COMPONENT SHARING IN DEVS

As previously stated, in a hierarchical component model, a shared component is a component that have more than one parent in the component hierarchy.

This definition conflicts with the definition of a DEVS hierarchical model (Coupled System) given by Zeigler et al. in [1] (p. 128):

A coupled system specification is a structure

$$N = \langle T, X_N, Y_N, D, M, I, Z \rangle$$

where

X_N is the set of inputs of the network N,

Y_N is the set of outputs of the network N,
D is the set of component references,
$M = \{M_d | d \in D\}$ are I/O systems,
$I = \{I_d | d \in D \cup \{N\}\}$ are the sets of influencers,
$Z = \{Z_d | d \in D \cup \{N\}\}$ are the interface maps.

This definition forbids sharing components because it requires that for a component $d \in D(N)$, where $D(N)$ is the set of components references of a coupled structure N, the *set of influencers* I_d of the component d is such that:

$$I_d \subseteq D(N) \cup \{N\}$$

In the case of shared components, the set $I_{d/shared}$ of influencers of d in all the system is the union of the influencers of d in all the coupled structures N_i that reference d:

$$I_{d/shared} \subseteq \bigcup_i D(N_i) \cup CS(d)$$

where

$$CS(d) = \{N_i | d \in D(N_i)\}.$$

3 SHARING MODELING PATTERNS

This section introduces three modeling patterns, which illustrate the usefulness and need of the shared component. These modeling patterns are useful for:

- modeling the real connections that may exist between components that are deeply buried into a component hierarchy (*proxy* modeling pattern);
- establishing shortcuts between components in order to reduce the overall simulation complexity of the model (the *shortcut* modeling pattern);
- enforcing layer separation and encapsulation in multi-layered architectures (the *matriochka*[1] modeling pattern).

We use the simple layered protocol stack model depicted in figure 1 as a basic example model that will be systematically referred to in the following sections.

This model represents a system made of two identical communicating `nodes`. These two `nodes` communicate with each other using a simplified OSI-like protocol stack made of the 4 upper layers of the OSI reference model (`application`, `presentation`, `session` and `transport`). These four layers have been chosen arbitrarily and their implementation needs not to be further described. At the lowest level, the two nodes communicate with each other using transport level packets. These packets are handled and delivered to each peer `node` by the central `transport network` component.

[1]Matriochka is the Russian name of the Russian dolls.

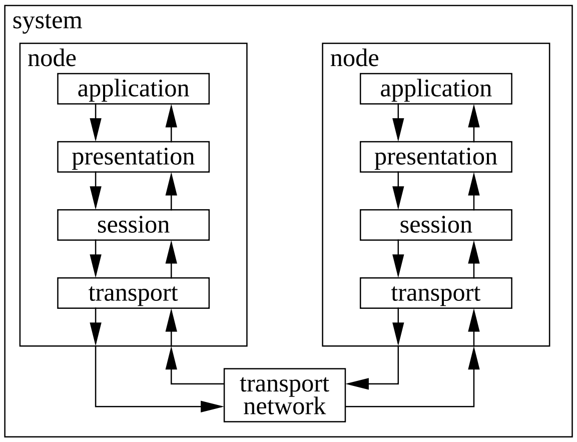

Figure 1. Two interconnected nodes communicating using an OSI-like layered protocol stack.

3.1 The *proxy* modeling pattern

Let's assume we want to model a road traffic network in which some of the vehicles, not all, are equipped with the nodes depicted on figure 1. Assume also that we want to reuse an already existing hierarchical model of the vehicle depicted on figure 2.

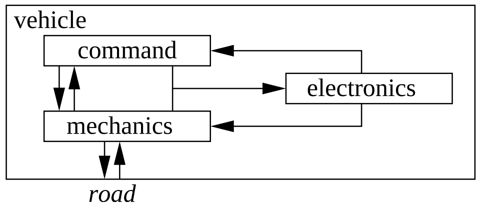

Figure 2. Decomposition of a simple vehicle model.

If we want to to plug the `node` component on this vehicle, we should plug it somewhere in the `electronics` component of the vehicle. However, as shown on figure 3, in order to plug the `node` component in the `vehicle`, the latter needs to be modified, in order to allow the `node` to reach the network (grayed area).

These modifications makes the task of reusing components more complicated. First notice that a node does not finally communicate with a network but with another node. So if we insert the same node model in two different kinds of vehicles and then we want both vehicles to communicate with each other, then both vehicles need to be modified as shown in the grayed area. In fact, the same kind of modification need to be applied to every kind of vehicle in which we want to plug the node.

Even worse, if the second node is not a vehicle but a very complex system and unfortunately the node is deeply buried in that complex system, then *all* the hierarchical levels of this

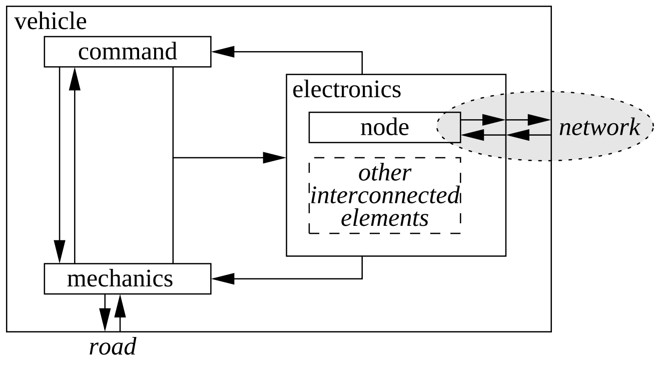

Figure 3. Model of a communicating vehicle reusing `node` and `vehicle` components.

complex system, potentially up to the root, need to be modified accordingly. However, it is worth stressing that object oriented techniques such as heritage may help to reduce (factor) significantly the amount of such modifications.

These problems can be addressed using shared components. The use of such components within the *proxy* usage pattern is illustrated on figure 4. This new construction uses a variant of the model of figure 1 in which the `transport` component is used as a shared component. This means that everywhere the shared `transport` component appears in the model, it will have the same content and state (instead of a new independent copy). Now let's consider what happens when the `node` component and its companion `transport` are inserted together multiple times, in various places (vehicles): every time, a new instance of `node` needs to be created (because it is a normal, non-shared component), while we reuse the *same* (shared) instance of the `transport`. Therefore this unique instance acts as a *proxy* between all the `node` instances. And since the `transport` and `node` are now laying close to each other, no modification is needed to the surrounding components.

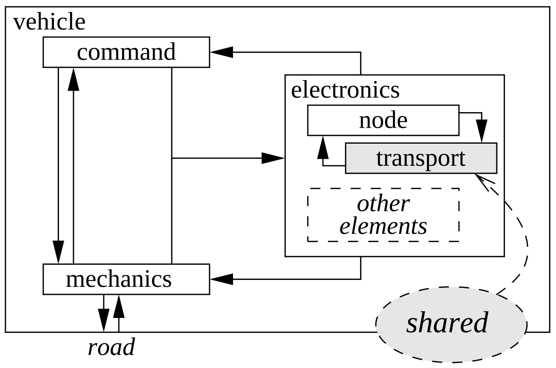

Figure 4. Communicating vehicle of figure 3, with a shared component used as a proxy.

To summarize, the *proxy* modeling pattern is useful for modeling new situations in which a given component (eg. the network) needs to be inserted in several places because it exhibits a strong ubiquitous nature. In this case the *proxy* modeling pattern allows for such an arbitrary insertion without having to modify the target component. This greatly favors the reuse of existing components, because the required modifications are local to the place where the new component is added, without any side effect on the surrounding components.

It is also worth stressing that we did not make any assumption on the dynamics of the modifications: the problem addressed thanks to shared components in this *proxy* modeling pattern is exactly the same if the insertion of a the new component need to be done once for all (the node is a fixed component of the vehicle) or dynamically during the simulation (the node is a component that may be plugged in or removed from the vehicle at any time). Hence, the benefits of sharing component are the same in the standard DEVS or in the Dynamic Structure variant such as DS-DEVS[5].

3.2 The *shortcut* modeling pattern

The *shortcut* modeling pattern consists in using a shared component to build interaction shortcuts between components. This construction may be used to shorten the interaction path between multiple components, and hence reduce the *simulation complexity* of the model (see for example [1] for a definition of the simulation complexity).

It is worth stressing that compared to the previous *proxy* pattern, the main goal of this *shortcut* is to create an interaction *that does not physically exist* in the real system: it is a new, fake interaction, that is only added in order reduce the simulation complexity. This kind of shortcut applies well to layered architectures, such as networks, in which peers at a given level need to use the services of lower layers to communicate with each other instead of directly exchanging messages.

The *shortcut* modeling pattern is illustrated by figure 5: the `application` inner component is the same as the original one described in figure 1; the `app-sc-wrapper` is a new hierarchical component that replaces the `application` component in the original model of figure 1 (both component have exactly the same interfaces); the `app-shortcut` component is a shared component that provides an alternate shorter path (hence the *shortcut* name) between every component in which it is plugged in. The decision to use this shorter path or not to use it is taken dynamically, for every packet, by the `app-switch-filter` component.

Thanks to this construction, an outgoing packet from the `application` inner component will either be directed toward the realistic path (the one with high simulation complexity) toward the `presentation` component, or toward

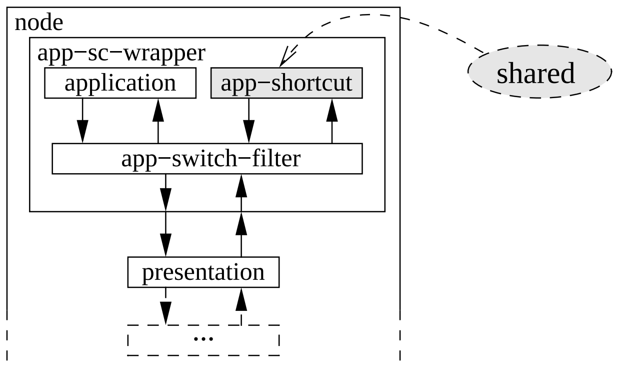

Figure 5. The *shortcut* modeling pattern applied to the `application` layer component.

the less realistic path through the `app-shortcut` component.

Compared to DS-DEVS, the dynamic structure variant of DEVS, notice that the decision to use the shortcut for a particular packet does not mean that subsequent packets will have also to use the shortcut. Since *both* path are needed at any time, the need here is not for a dynamic change of structure, but for the simultaneous availability of both structures.

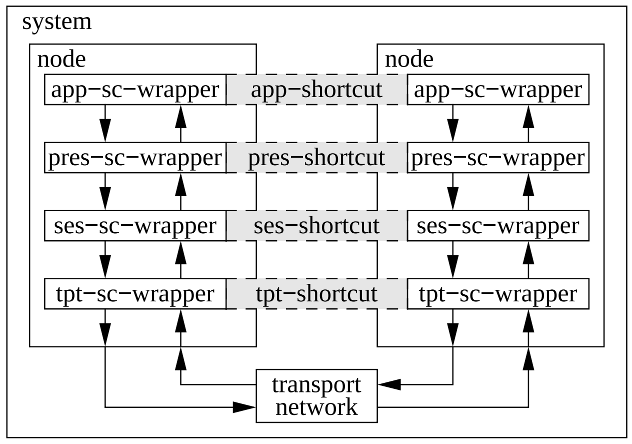

Figure 6. The *shortcut* modeling pattern may be applied (independently) to each level of a protocol stack.

This construction may be applied several times in the same model. For example, as shown on figure 6, this shortcut construction may be applied to each of the four components that model a network layer: the `application`, as already described in figure 5 but also the `presentation`, the `session` and the `transport` ones. In each case a new dedicated "switch-filter" component needs to be implemented.

Therefore, this shortcut modeling pattern provides a powerful mean for adjusting the simulation complexity of a model. However, deciding in which cases it is relevant to use the shortcut path and in which cases it is not is a difficult question because it strongly depends on the model *and* the simulation goals. This question is not further addressed in this paper.

3.3 The *matriochka* modeling pattern

The *matriochka* (or Russian doll) modeling pattern applies to models of systems that exhibit a recursively hierarchical structure. The OSI-like layered model of figure 1 is a good illustration of such a system.

However, despite the flat design of the model depicted on figure 1 reflects the usual layered representation of OSI-like models, it does not fully reflect the hierarchical philosophy of the OSI-layered reference model. Indeed, this flat design somehow violates (ignores) one of the fundamental principles of the layered approach (see for example [6] for a summarized description of the OSI layered model philosophy and its associated terminology): an entity of level (N) can only interact with entities of level $(N+1)$ and entities of level $(N-1)$. Indeed, despite no violation of this principle appears in the example of figure 1, this flat design cannot help to prevent such a violation: one could, mistakenly or on purpose, decide to connect the `application` component directly to the `transport` component (provided that these two components have compliant interfaces).

Since we are considering hierarchical modeling in this paper, a convenient way to fully enforce this fundamental isolation principle of layering is to enforce isolation by means of the component hierarchy. At this point we have two options: either we decide that the upper-most (OSI-like) layer is the outer-most component in the hierarchy or conversely (and paradoxically), we decide that the lower-most (OSI-like) layer is outer-most component in the hierarchy.

The first option is illustrated on figure 7: the left side of the figure represents the component hierarchy that implements the OSI-like layered hierarchy depicted on the right side of the figure. However, since each layer except the lowest one is implemented as a hierarchical component, new components are introduced in order to distinguish the implementation parts of the layers from their surrounding container (`application entity`, `presentation entity` and `session entity`).

Figure 7 also clearly demonstrates why this *matriochka* modeling pattern strongly appeals for shared components. First, the actual interactions are supposed to occur at the lowest (OSI-Layer) level. Therefore, we need some way of establishing connections between the inner-most `transport` components, which leads back to the *proxy* modeling pattern described in section 3.1.

Furthermore, the model depicted on figure 1 is a simplistic case of a more general interaction model in which the follow-

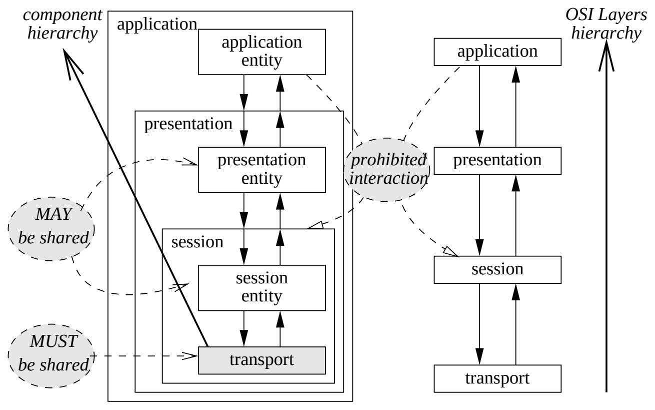

Figure 7. One of the two possible hierarchical implementation of the simple OSI-Layered model depicted in figure 1 that strictly enforces the OSI interaction policy.

ing interaction patterns could also occur:

- the services provided by an entity of level (N) may be used by several upper entities of level $(N+1)$;
- an entity of level (N) may be build on top of several services of level $(N-1)$, provided by one or possibly several entities of level $(N-1)$.

Let's consider the case of the first pattern. Applied to the upper-most layer of our reference example, this leads to the situation depicted on figure 8, where three `application` components use the same `presentation` component. Applying the *matriochka* modeling pattern means that the `presentation` component needs to be inserted at the same time in *each* of the three `application` components, as shown on figure 9. In other words, the `presentation` component needs to be shared.

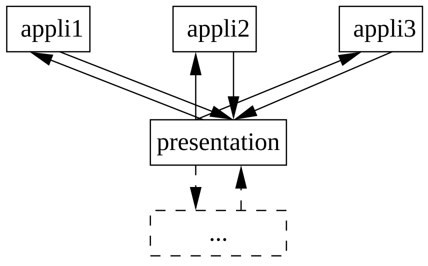

Figure 8. An example of a more complex interaction pattern: three applications interact with the same presentation component (OSI-like layered representation).

If we generalize the previous example at every possible level of our layered protocol stack, this means for any layer (N), the component entity (object instance) that implements the services of this layer (N) may be shared several times

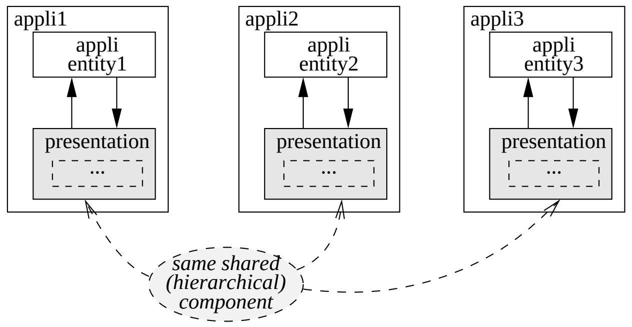

Figure 9. The *matriochka* modeling pattern applied to the example of figure 8.

amongst entities of the upper layer $(N+1)$. The number of times the same entity is shared depends on the number of times entities of level $(N+1)$ need the services of level (N). Hence, this *matriochka* pattern strongly appeals for sharing because situations are possible in which a component shared at the highest level contains one or several sub-components that are shared themselves at the next level and so own down to the bottom of the hierarchy.

4 CONCLUSION

The goal of this paper was to raise an issue about sharing components in DEVS. The issue is to decide whether DEVS should eventually support sharing and if so, to decide how and to what extent component sharing could be added in DEVS and its variants. We presented three modeling patterns, that represent three different kind of modeling problems for which we claim that the sharing feature is useful.

The *proxy* modeling pattern describes a situation in which sharing components helps to better support the ubiquitous nature of a sub-system that may be found in various places of a global system. In this situation, sharing component(s) avoids structural modification on the surrounding components. The *shortcut* modeling pattern describes a situation in which sharing components helps the simulationist to lower the simulation complexity of his models. In this situation, sharing is used to install low complexity shortcuts in addition to the regular (high complexity) interaction paths of the model. The *matriochka* modeling pattern describes a situation in which sharing components helps to recursively enforce encapsulation in order to better reflect the layered architecture of a (sub-)system. In this situation, sharing is used to decide which (where) interaction paths should be allowed.

The technical issue of implementing the sharing feature in a DEVS simulator is not addressed in this paper. For example, sharing adds a noticeable complexity to the process of switching from a sequential to a parallel and distributed implementation of the simulator. More generally, as soon as a component may have several parents in the hierarchy, which is the definition of the shared component, we may face various problems of *concurrency of (hierarchical) control*.

These technical issues are currently being partly addressed by the few component models implementations that offer the sharing feature, such as the ObjectWeb's Fractal component model[3], that fully support sharing *and* parallel and distributed execution thanks to the FractalRMI library[2].

Experiments and further studies about sharing are currently conducted in the Open Simulation Architecture (OSA) project[3] which is component-based discrete-event simulator built on top of the Fractal component model.

5 ACKNOWLEDGMENTS

This work is partly co-supported by the IST-FET "AEOLUS" project and NSERC.

REFERENCES

[1] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation*, 2nd ed. Academic Press, 2000.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[3] E. Bruneton, T. Coupaye, and J. Stefani, "The fractal component model specification," Available from http://fractal.objectweb.org/specification/, February 2004, draft version 2.0-3.

[4] S. B. Lim and D. Ferry, *Jain SLEE 1.0 Specification*, Sun Microsystems Inc. & Open Cloud Ltd., 2002, final release, availble from http://jcp.org/aboutJava/communityprocess/final/jsr022/index.html.

[5] F. Barros, "Modeling Formalisms for Dynamics Structure Systems," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 4, pp. 501–515, 1997.

[6] H. Zimmerman, "OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications*, vol. COM-28, no. 4, pp. 425–432, April 1980, (Invited paper).

BIOGRAPHY

OLIVIER DALLE is assistant professor in the C.S. dept. of Faculty of Sciences at University of Nice-Sophia Antipolis (UNSA). He received is BS from U. of Bordeaux 1 and his M.Sc. and Ph.D. from UNSA. From 1999 to 2000 he was a post-doctoral fellow at the the french space agency center in Toulouse (CNES-CST), where he started working on component-based discrete event simulation of complex telecommunication systems. In 2000, he joined the MASCOTTE project, a common team of the I3S-UNSA/CNRS Laboratory and the INRIA Research Unit, in Sophia Antipolis.

GABRIEL WAINER received the M.Sc. (1993) and Ph.D. degrees (1998, with highest honors) of the Universidad de Buenos Aires, Argentina, and Université d Aix-Marseille III, France. In July 2000, he joined the Department of Systems and Computer Engineering, Carleton University (Ottawa, ON, Canada), where he is now an Associate Professor. He has been a Professor at the Computer Sciences Department of the Universidad de Buenos Aires, Polytech de Marseille, and a Visiting Research Scholar at the ACIMS (University of Arizona) and LSIS (CNRS, France). He is Associate Editor of the Transactions of the SCS, and the International Journal of Simulation and Process Modeling. He is a chairman of the DEVS standardization study group (SISO), Director of the Ottawa Center of The McLeod Institute of Simulation Sciences and chair of the Ottawa M&SNet.

[2] `http://fractal.objectweb.org/fractalrmi/index.html`.

[3] http://osa.gforge.inria.fr/