Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler eds.

EXTENDING DEVS TO SUPPORT MULTIPLE OCCURRENCE IN COMPONENT-BASED SIMULATION

Olivier Dalle

INRIA Sophia Antipolis Méditerranée
and I3S, Université de Nice-Sophia Antipolis & CNRS
B.P. 93, F-06902 Sophia Antipolis Cedex, FRANCE

Bernard P. Zeigler

University of Arizona,
Arizona Center for Integrative Modeling and Simulation
Tucson, AZ 85721, U.S.A.

Gabriel A. Wainer

Carleton University
Dept. of Systems and Computer Engineering
1125 Colonel By Dr. Ottawa, Ontario, CANADA.

ABSTRACT

This paper presents a new extension of the DEVS formalism that allows multiple occurrences of a given *instance* of a DEVS component. This paper is a follow-up to a previous short paper in which the issue of supporting a new construction called a shared component was raised, in the case of a DEVS model. In this paper, we first demonstrate, formally, that the multi-occurrence extended definition, that includes the case of shared components, is valid because any model that is built using this extended definition accepts an equivalent model built using standard DEVS. Then we recall the benefits of sharing components for modeling, and further extend this analysis to the simulation area, by investigating how shared components can help to design better simulation engines. Finally, we describe an existing implementation of a simulation software that fully supports this shared component feature, both at the modeling and simulation levels.

1 INTRODUCTION

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(Zeigler, Praehofer, and Kim 2000).

In a previous short paper (Dalle and Wainer 2007), we raised the issue of supporting a new construction in DEVS models, called a *shared component* (definition given hereafter). This paper is a follow-up to this previous short paper in which (i) we give an answer to the question raised, and (ii) we further extend our analysis of the benefits of using this new construction for building simulation engines. However, for the sake of completeness, we will also recall the benefits of using shared components for modeling.

Definition 1. *In a hierarchical component model, a* ***shared component*** *is a component* instance *that have more than one parent in the component hierarchy, with the only restriction that such component cannot appear both as an ancestor and a descendant for another component (including itself).*

In other words, a component instance S is a shared instance between two components A and B if S is both a child component of A and a child component of B. The restriction means that no cycle is allowed and consequently, the hierarchy must be a Directed Acyclic Graph (DAG). In the previous example, this restriction means that: (i) S is different from A and B; (ii) neither A nor B happen to be a descendant of S in the component hierarchy. However, A can be a descendant of B or B a descendant of A, as long as it does not create a cycle.

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 object-oriented terminology, reused components correspond to different *instances* of a same *class*. On the contrary, every time a component is shared, it uses the same identical internal state. Using object-oriented terminology, shared components correspond to *references* to a unique *instance* of a given *class*.

978-1-4244-2708-6/08/$25.00 ©2008 IEEE

Very few component models do effectively support this sharing feature: the Fractal component model (Brunneton et al. 2006) does explicitly support sharing while some others, like JainSLEE (Lim and Ferry 2002) provide proxying techniques which is a practical way of implementing sharing.

This sharing feature is closely related to another property that we introduce for the first time in this paper in the context of DEVS: multi-occurrence. An occurrence correspond to the concept of using a component instance once. Multi-occurrence corresponds to the capability of using a component instance multiple times in a model. Therefore, in order to support shared components, multi-occurrence capability is required. Notice that shared components form a subset of multi-occurring components, because a shared component is required to have more than one parent in the hierarchy (in each of which it must occur at least once), while this constraint does not apply to multi-occurrence.

In section 2, we first give a formal analysis of multi-occurrence in DEVS and prove that an extended definition of DEVS supporting this mechanism does not introduce new DEVS behaviors. This result applies in turn to shared components and proves that shared components can be used to build hierarchical models that are equivalent to standard DEVS models.

Then, in order to illustrate the benefits and usefulness of the shared component construction in DEVS, we will first recall, in section 3, some of the *modeling patterns* presented in (Dalle and Wainer 2007).

Then, in section 4, we further extend this analysis to the simulation area, by investigating the usefulness of shared components (and multiple occurrence) to design better simulation engines. Finally, in section 5 we describe an existing implementation of a simulation software that fully supports this shared component feature, both at the modeling and simulation levels.

2 FORMAL DEFINITIONS AND ANALYSIS

2.1 DEVS Coupled Model

A DEVS coupled model (or network) specification includes the external interface (input and output ports and values), the components (which must be DEVS models), and the coupling relations. This is a structure:

$$N = (X, Y, D, Md | d \in D, EIC, EOC, IC)$$

where

- $X = \{(p,v) | p \in IPorts, v \in X_p\}$ is the set of input ports and values;
- $Y = \{(p,v) | p \in OPorts, v \in Y_p\}$ is the set of output ports and values;
- D is the set of names for the components.
- $M_d, d \in D$ are DEVS models:

$$M_d = (X_d, Y_d, S, \delta_{ext}, \delta_{int}, \lambda, ta)$$

with

$$X_d = \{(p,v) | p \in IPorts_d, v \in X_p\};$$
$$Y_d = \{(p,v) | p \in OPorts_d, v \in Y_p\}.$$

- EIC, the External Input Coupling, connects external inputs to component inputs:

$$EIC \subseteq \{((N, ip_N), (d, ip_d)) | ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$$

- EOC, the External Output Coupling connects external outputs to component outputs:

$$EOC \subseteq \{((N, op_d), (N, op_N)) | op_N \in OPorts, d \in D, op_d \in OPorts_d\}$$

- IC, the Internal Coupling, connects component outputs to component inputs:

$$IC \subseteq \{((a, op_a), (b, ip_b)) | a, b \in D, op_a \in OPorts_a, ip_b \in IPorts_b\}$$

However, *no direct feedback loops are allowed*, i.e., no output port of a component may be connected to an input port of the same component i.e.,

$$((d, op_d), (e, ip_d)) \in IC \Rightarrow d \neq e.$$

with the following **interface constraints:**

$$\forall ((N, ip_N), (d, ip_d)) \in EIC : range_{ip_N}(X) \subseteq range_{ip_d}(X_d);$$

$$\forall ((d, op_d), (N, op_N)) \in EOC : range_{op_d}(Y_d) \subseteq range_{op_N}(Y);$$

$$\forall ((d, op_a), (b, ip_b)) \in IC : range_{op_a}(Y_a) \subseteq range_{ip_b}(X_b).$$

Recall that closure under coupling requires that, for any coupled model all of whose components are expressed in DEVS, there is an equivalent system (called the resultant), that is also expressible in DEVS. Thus, closure under coupling of DEVS is demonstrated by showing how to express the resultant of any coupled DEVS as a basic DEVS. This algorithmic construction provides the formal basis for the DEVS simulation protocol and provides a

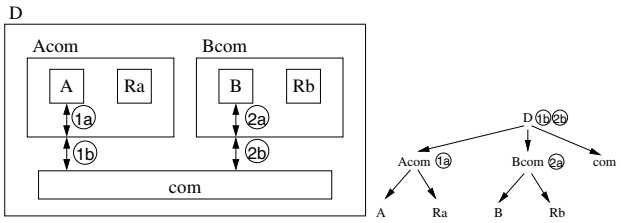

Figure 1: An example of shared component with hierarchy traversal: in order to reach the shared component com, components A and B use both the coupling (1a) and (1b) in their surrounding components Acom and Bcom, and then the couplings (2a) and (2b) in component D.

standard for verifying the correctness of implementations of DEVS simulation engines.

In the example of Figure 1-(a), a communications component named *com* is shared by some users depicted as *A* and *B*. This situation is represented with the standard DEVS hierarchical construction in which, since *A* is contained within a larger coupled model, *Acom*, the couplings must traverse the larger coupled model before reaching the communications component.

2.2 Multi-occurrence

As indicated earlier, if we allow multiple occurrences, we can provide greater visual understanding of model structure, as will illustrated shortly. In the standard definition, it was understood that distinct names index distinct DEVS models, i.e., there is a one-one mapping between the set D and the set of components. However, in order to accommodate multiple occurrences of the same component, we must relax the unique naming requirement and allow a many to one mapping. More formally, we add a set of names, D which are in one-one correspondence with the components and a mapping, f from the set of indices, D' to the set of names such that

$$f : D' \rightarrow D$$

is total on D' (every index maps to a component name) and is onto D. (every component name has at least one indexical reference to it.)

The extended specification has the following form:

$$N^{extended} = (X, Y, D', \{M_d | d \in D\}, f, EIC', EOC', IC')$$

where

D' and D are interpreted as indices and names and f is a mapping of the kind just mentioned and where the couplings are all defined using the indices rather than the names. For example, a pair, $((i,op),(j,ip))$ in the internal coupling IC' represents a coupling that is stated in terms of the indices, i and $j \in D'$, and the output and input ports of i and j, $op \in OPorts_i, ip \in IPorts_j$ respectively. The external couplings EIC' and EOC' are similarly defined.

Consider any class in the equivalence defined by f e.g.., the indices, i mapped to the same name $d, [i]_d = \{i | f(i) = d\}$. If such an equivalence class has more than one element, the indices in it represent multiple occurrences of the same component. Since such multiple occurrences represent the same component, the input and output ports they employ in the couplings must be the same. Further, for a well-defined resultant the couplings so defined must be consistent with each other. This leads us to state the following constraint.

Constraint on Coupling of Extended Specification: if there is a coupling $((i,op),(j,ip)) \in IC'$ and i', j' are f-equivalent to i, j, respectively, then there must also be a coupling $((i',op),(j',ip)) \in IC'$. Similarly, the external couplings EIC' and EOC' are so constrained.

It is now straight forward to associate a well-defined standard coupled model N_f with an extended one

$$N^{extended} = (X, Y, D', \{Md | d \in D\}, f, EIC', EOC', IC'),$$

where

$$N_f = (X, Y, D, \{M_d | d \in D\}, EIC_f, EOC_f, IC_f)$$

and

$$IC_f = \{((a,op),(b,ip)) | ((i,op),(j,ip)) \in IC', f(i) = a, f(j) = b\}.$$

The external couplings EIC_f and EOC_f are similarly defined. Note that the constraint on coupling of the extended specification ensures that the couplings defined for the associated standard model are well defined. This allows us to state the behavior of an extended DEVS model is actually the behavior of its associated standard model. Consequently we have the conclusion that:

The extended DEVS formalism allowing multiple occurrences of components, while being more flexible in expression, does not introduce new DEVS behaviors.

The utility of multi-occurrence flexibility of expression comes out in its application to hierarchical DEVS models. For such constructions we apply flattening to the hierarchical model to reduce it to a single level DEVS and then apply the interpretation just given to associate a standard DEVS behavior with it. In the example of Figure 2-(a), a communications component named *com* is shared by some users depicted as *A* and *B*. The component, indexed by subscripts 1 and 2, respectively, occurs in two places of the

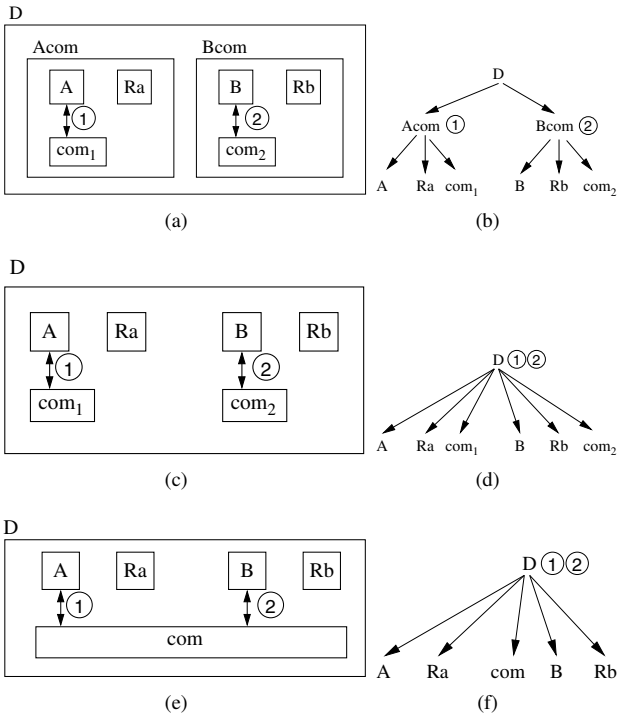

Figure 2: Examples of shared component equivalent to previous example of Figure 1 but avoiding hierarchy traversal: first using multiple occurrences and preserving the hierarchy structure in (a) and (b); then after flattening the hierarchical structure of (a) and (b) in (c) and (d); and finally after reunifying the multiple occurrence into a single one in (e) and (f). The latter is a standard DEVS.

hierarchical construction tree shown in Figure 2-(b). The reduction to a flattened version is shown in Figure 2-(c) and the interpretation as a standard DEVS in Figure 2-(e) illustrates the mapping of the indexed occurrences of the shared communication component to its unique name. In other words, the mapping f in the extended coupled model specification takes both com_1 and com_2 into com. Finally, the flattened standard DEVS, illustrated in Figure 2-(e), is well-defined since there the users A and B are distinct, falling into distinct equivalence classes, and therefore must define consistent couplings.

The operation of flattening hierarchical DEVS models is defined in the literature and is well known to reduce the hierarchy of structure while preserving the original behavior (Kim, Seong, Kim, and Park 1996, Kim, Kang, Sagong, and Seo 2000, Zacharewicz and Hamri 2007). It is an easy step to extend its application to the extended DEVS models introduced here. In such a process, a multi-step coupling that exists in the hierarchy is replaced by a direct coupling in the flattened version. This is illustrated in Figure 2-(c), where the couplings between A and com_1 labeled by the circled 1 are moved to a flattened version while the containing component *Acom* is eliminated. It is also illustrated in Figure 1 where the multi-step couplings labeled by 1a and 1b would be replaced by a direct coupling such as that labeled by 1 in Figure 2-(e).

3 MULTI-OCCURRENCE MODELING PATTERNS

In order to illustrate the usefulness of component multi-occurrence, this section recalls two of the three modeling patterns that were described, in greater detail, in (Dalle and Wainer 2007). Modeling patterns are inspired from the (Software) Design Patterns (Gamma et al. 1994) and describe 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 modeling practices to the practitioners.

The two modeling patterns we are going to describe 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).

These two modeling patterns use the same simple layered protocol stack model that is depicted in Figure 3. 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. 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.

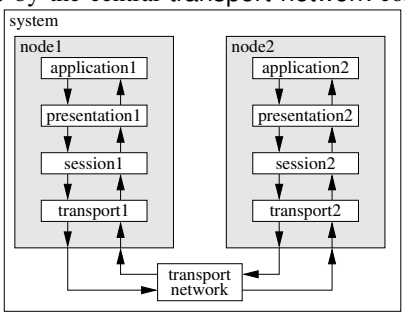

Figure 3: Two interconnected nodes communicating using an OSI-like layered protocol stack.

3.1 The *proxy* modeling pattern

Let us 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 3. Assume also that we want to reuse an already existing hierarchical model, such as a model of a vehicle.

If we want to plug one of the node components of Figure 3 in this vehicle, for example, node1, we should plug it somewhere in the electronics component of that vehicle. However, as shown on Figure 4, in order to allow the proper functioning of the node1 component, the vehicle component needs to be modified, in order to allow node1 to reach the network transport as shown in Figure 3 (grayed dashed ellipse).

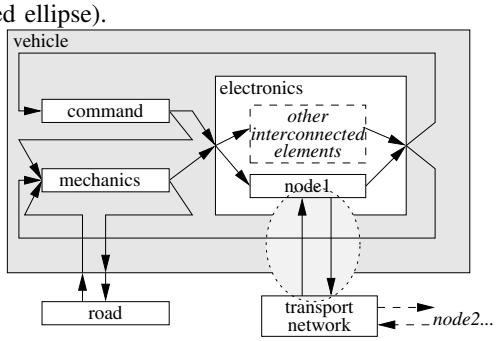

Figure 4: Model of a communicating vehicle reusing node and vehicle components. The grayed dashed ellipse points out the "hole punching" transformation needed to enable this reuse when shared component are not available.

Without using shared components, the typical modification required consists in adding a new communication port, in multiple places, in order to allow the communications of node1 to go through its surrounding components and reach their destination. Hence, these "hole punching" modifications makes the task of reusing components more complicated than a simple "drag-and-drop" operation.

This problem can be solved using the *proxy* modeling pattern, as illustrated in 5. This new construction uses a variant of the model of Figure 3 in which the transport component is used as a shared component. 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.

To summarize, the *proxy* modeling pattern is useful for modeling new situations in which a given component

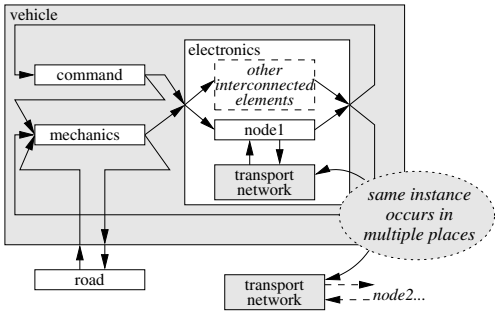

Figure 5: Communicating vehicle of Figure 4, with a shared component used as a proxy.

(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 "host" 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(Barros 1997).

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 (Zeigler, Praehofer, and Kim 2000) 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 consists in applying the transformation illustrated by Figure 6 everywhere a *shortcut end-point* is needed (the figure only shows the transformation for application1, but a similar transformation is required for application2). The application1 inner component is the same as the original one described in Figure 3; the app-sc-wrapper1 is a new wrapping hierarchical component that replaces the application1 component in the original model of Figure 3 (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-filter1 component.

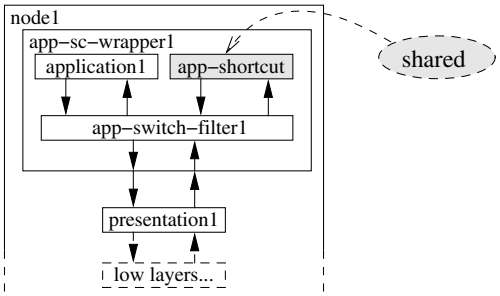

Figure 6: The *shortcut* modeling pattern applied to the application1 component. (The same modification is applied to application2, but is not shown here.)

Thanks to this construction, an outgoing packet from the application1 inner component will either be directed toward the realistic path (the one with high simulation complexity) toward the presentation1 component, or toward 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* paths are needed at any time, the need here is not for a dynamic change of structure, but for the simultaneous availability of both structures.

This construction may be applied several times in the same model. For example, as shown on Figure 7, this shortcut construction may be applied to each of the four components that model a network layer: the application, as already described in Figure 6 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).

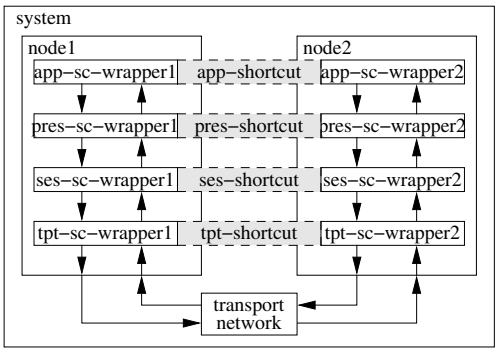

Figure 7: The *shortcut* modeling pattern may be applied (independently) to each level of a protocol stack.

4 SIMULATION WITH MULTI-OCCURRENCE

This section presents two use cases in which multi-occurrence prove to be useful when implementing a simulation:

- Future Event Set (FES) implementation: using multi-occurrence in order to provide efficient FES implementations;
- Embedding external simulators: using multi-occurrence in order to ease inter-operability between simulators;

4.1 Multi-occurrence benefits in Future Event Sets

The *scheduler* is the part of the simulation engine that is in charge of deciding which is the next event to be processed by a component. As soon as the scheduling activity happens concurrently for different models, a synchronizing policy has to be implemented in order to enforce simulation time consistency. In a canonical DEVS implementation(Zeigler, Praehofer, and Kim 2000), as shown in Figure 8, these elementary scheduling places correspond to the Atomic DEVS components, and the synchronization is implemented using additional synchronization components called "coordinators" that are placed in each Coupled DEVS.

The protocol between the coordinators and the schedulers consists in computing the global minimum, for all the Atomic DEVS, of their next event simulation time and then notifying the DEVS components of the result. This global minimum computation is distributed along the hierarchy, starting from a root coordinator, as shown with an example in Figure 9.

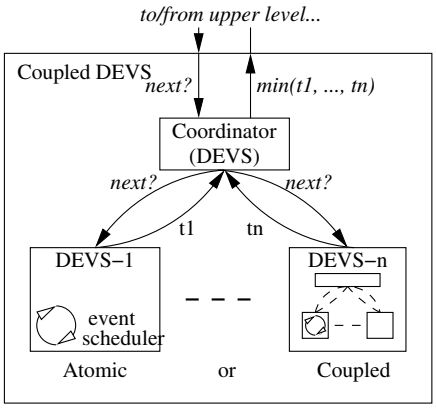

Figure 8: Scheduling hierarchy in DEVS

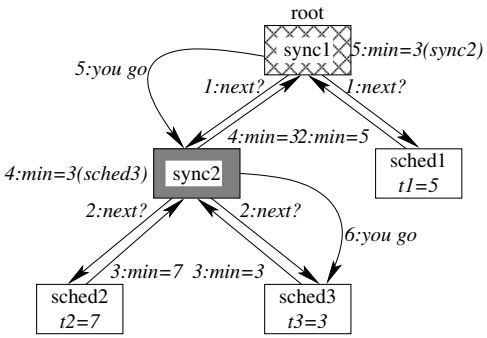

Figure 9: Applying the synchronization protocol along the hierarchy.

The example of Figure 9 also illustrates that the higher the hierarchy, the longer it takes to compute this minimum. Indeed, a synchronization step is required at each node of the hierarchy in order to wait for the sub-trees values and compute the local minimum for a given node. Therefore, in the worst case, when Atomic DEVS are deeply buried in the DEVS hierarchy, this computation is very costly.

A possible optimization to avoid this cost consists in flattening the hierarchy. Indeed, since any hierarchical coupling has a flattened equivalent, transforming a hierarchical model in its flattened equivalent prior to starting the execution is a way of reducing the number of synchronization step to only one.

Unfortunately, flattening may not be practical in some situations, when the hierarchy is still useful. This is the case when the model has to support dynamic structure changes, because these dynamic structure changes are localized in a given part of the component hierarchy.

In that case, using multi-occurrence is a way of achieving the same result as flattening, but without losing the hierarchical structure. The extreme case solution with multi-occurrence consists in replacing all the internal-nodes coordinators by the *same* instance of the root coordinator (hence occurring everywhere a coordinator is needed). Notice that with this solution, since all Atomic DEVS are now directly connected to *the* root coordinator, the need for relaying the minimal values upstream disappears. This is consistent with the fact that this root coordinator is not supposed to act as relay.

4.2 Embedding external simulators

Let us consider for example the case of coupling DEVS models with Network Simulator (NS) models, as investigated in (Kim 2006). Assume that we want to establish a communication between two users *A* and *B* through a communication component (as discussed earlier in the example of section 2), but this time using a model simulated by NS. In this case, as illustrated by Figure 10, the use of shared components allows to establish a coupling between the users component *A* and *B* and the NS wrapper component everywhere needed, while preserving a centralized implementation of the wrapper.

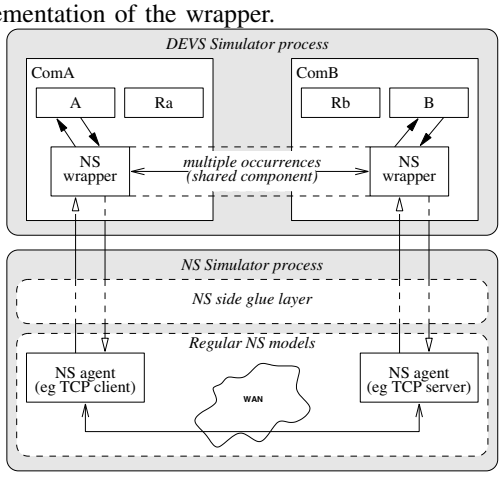

Figure 10: A transparent connexion between DEVS models and NS native models through a shared wrapper component occurring everywhere needed in the DEVS model.

5 IMPLEMENTATION IN OSA

OSA (Open Simulation Architecture) is a new collaborative platform for component-based discrete event simulations (Dalle 2007). It relies on the ObjectWeb's Fractal component model (Brunneton et al. 2006) and its Java-based implementation called AOKell (Seinturier et al. 2006). The front-end Graphical Interface is based on the Eclipse IDE

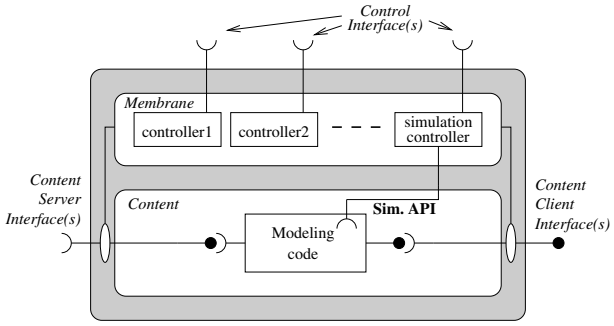

Figure 11: Architecture of an OSA/Fractal component with its simulation controller.

(des Rivières and Wiegand 2004). OSA also provides a public repository based on the Apache Foundation's *maven* building system that automatically computes the dependencies between the components used for a given simulation.

In OSA, the simulation engine is embedded directly in each component, by means of a dedicated controller. Controllers are key elements of the Fractal component model: as shown in Figure 11, in Fractal, each component is attached to, and supervised by a set of controllers, each implementing a non-functional service (such as introspection of content, couplings management, or life cycle management). The number of these controllers is not fixed, and the list of controllers associated with each component may vary from one component to another. Such a given list of controllers attached to a component is called the component *membrane*. In OSA, for the needs of simulation, we provide a list of controllers dedicated to simulation, that can be used to build various simulation-oriented membranes.

The implementation of Fractal we use, AOKell, offers another interesting feature: the ability to build component-based membranes, in which controllers are themselves implemented by means of Fractal components. Hence, this reflexive approach allows controllers to be coupled, to have a hierarchical structure, and to be shared.

These latter capabilities are plainly used in OSA. For example, despite the engine implementation is distributed in the membrane of each component, we can still implement a centralized scheduling algorithm thanks to shared components. But on the other hand, if we want to switch transparently to a fully distributed implementation running on a cluster of workstations, then it is sufficient to replace this unique shared scheduler component implementing a centralized scheduling algorithm by a set of scheduler components distributed on each execution platform and implementing a distributed synchronization algorithm.

During the simulation, the functional part of the component (the model) may use the set of services that form the Simulation API (noted Sim. API on Figure 11). Since the API is provided directly within each component, it may be changed from one component to another simply by replacing the simulation controller component. This is interesting for instance in order to *mimic* other simulators and therefore reuse their existing models.

The native OSA simulation API is process oriented and provides primitives for: getting the current simulated time, terminating the simulation, scheduling new events within each component (an event corresponds to the execution of a method by a new thread at a particular time in simulation), and pausing/resuming execution of a processing thread.

Implementing a DEVS-like simulation API on top of OSA is straight-forward: it is sufficient to create a looping thread in each DEVS using a starting event at the beginning of the simulation. This looping thread will then implement the behavior of a DEVS component using the OSA API and its client and server interfaces for sending and receiving inputs/outputs.

6 CONCLUSION

In this paper we introduced a new extension of the DEVS formalism that allows multiple occurrences of a given instance of a DEVS component. We first demonstrated formally that this multi-occurrence extension does not produce new DEVS behaviors. This result applies in turn to the particular case of the shared component, because it is based on this multi-occurrence principle. Therefore, we have the formal proof that a shared component is a valid DEVS construction that complies with the Systems Theory principles. Then, we presented and discussed various ways of exploiting this shared component construction, both for modeling and simulation. These selected examples demonstrated that a shared component is both a means of reducing the complexity of models and simulations and a means for enabling a better reuse of components.

In the last part of the paper, we presented an existing implementation of a simulator, namely the OSA architecture, that fully supports this new shared component construction. Despite the fact that this architecture is not currently implementing a DEVS API, its versatile architecture and its process-oriented API are expected to allow such an adaptation easily, which is part of our short term development program.

Finally, we address a potential drawback that has been identified with shared components concerning the breakdown of the tree structure of the component hierarchy (the tree becomes a Directed Acyclic Graph). A practical effect of this relaxed constraint on the hierarchy is that it might introduce a concurrence of control problem. Semantically, our formal analysis proved that with respect to DEVS, the behavior induced by multi-occurring components is well defined provided that the Constraint on Coupling (Section 2.2) is respected. Hence, this concurrency issue is not raised with the DEVS extension we have presented for any imple-

mentation that is equivalent to the flattened canonical form of Section 2.2.

ACKNOWLEDGMENTS

This work was done while Prof. Zeigler was visiting INRIA Sophia Antipolis. The OSA development is co-supported by the IST-FET "AEOLUS" project, the ANR SPREADS and OSERA projects and INRIA.

REFERENCES

Barros, F. 1997. Modeling Formalisms for Dynamics Structure Systems. *ACM Transactions on Modeling and Computer Simulation* 7 (4): 501–515.

Brunneton, E., T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stéfani. 2006. The FRACTAL Component Model and Its Support in Java. *Software Practice and Experience, special issue on Experiences with Auto-adaptative and Reconfigurable Systems* 36 (11-12).

Dalle, O. 2007, July. The OSA Project: an Example of Component Based Software Engineering Techniques Applied to Simulation. In *Proc. of the Summer Computer Simulation Conference (SCSC'07)*, 1155–1162. Invited paper.

Dalle, O., and G. Wainer. 2007, July 14-19. An open issue on applying sharing modeling patterns in devs. Published on the CDROM of the Summer Computer Simulation Conference (SCSC'07). Short paper.

des Rivières, J., and J. Wiegand. 2004. Eclipse: A platform for integrating development tools. *IBM Systems Journal* 43 (2): 371–383.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1994. *Design patterns – elements of reusable object-oriented software*. Addison-Wesley.

Kim, K., W. Kang, B. Sagong, and H. Seo. 2000. Efficient distributed simulation of hierarchical devs models:transforming model structure into a non-hierarchical one. In *Proc. of the 33rd Annual Simulation Symposium (SS2000)*, 227–233.

Kim, K., Y. Seong, T. Kim, and K. Park. 1996. Distributed simulation of hierarchical devs models: Hierarchical scheduling locally and time warp globally. *Simulation: Transaction of the SCS* 13 (3): 135–154.

Kim, T. 2006, Spring. DEVS-NS2 environment: An integrated tool for efficient networks modeling and simulation. Master's thesis, Electrical and Computer Engineering Dept., University of Arizona.

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

Seinturier, L., N. Pessemier, L. Ducgien, and T. Coupaye. 2006, June 29th - July 1st. A component model engineered with components and aspects. In *Proc. of the 9th Int'l Symp. on Component-Based Software Engineering*, Volume LNCS 4063, 139–153. Vasteras, Sweden: Springer.

Zacharewicz, G., and M. E.-A. Hamri. 2007, Feb 8-10. Flattening g-devs / hla structure for distributed simulation of workflows. In *Proc. of AIS-CMS Intl. Modeling and Simulation Multiconference*, 11–16. Buenos Aires.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modeling and simulation*. 2nd ed. Academic Press.

AUTHOR BIOGRAPHIES

OLIVIER DALLE is associate 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 common project-team of the I3S-UNSA/CNRS Laboratory and INRIA, in Sophia Antipolis.

BERNARD P. ZEIGLER is Professor of Electrical and Computer Engineering at the University of Arizona, Tucson and Director of the Arizona Center for Integrative Modeling and Simulation. He is internationally known for his 1976 foundational text Theory of Modeling and Simulation, recently revised for a second edition (Academic Press, 2000), He has published numerous books and research publications on the Discrete Event System Specification (DEVS) formalism. In 1995, he was named Fellow of the IEEE in recognition of his contributions to the theory of discrete event simulation. In 2000 he received the McLeod Founder's Award by the Society for Computer Simulation, its highest recognition, for his contributions to discrete event simulation. He was appointed Fellow of the Society for Modeling and Simulation, International (SCS), 2006.

GABRIEL A. WAINER received the M.Sc. (1993) and Ph.D. degrees (1998, with highest honors) of the Universidad de Buenos Aires, Argentina, and Universit 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.