OSA: an Open Component-based Architecture for Discrete-Event Simulation

Olivier Dalle

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

OSA: an Open Component-based Architecture for Discrete-Event Simulation

Olivier Dalle

N° 5762

Novembre 2005

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5762--FR+ENG

OSA: an Open Component-based Architecture for Discrete-Event Simulation

Olivier Dalle*

Thème COM — Systèmes communicants
Projet MASCOTTE

Rapport de recherche n° 5762 — Novembre 2005 — 23 pages

Abstract: Component-based modeling has many well-known good properties. Out of these properties is the ability to dispatch the modeling effort amongst several experts each having their own area of system expertise. Clearly, the less experts have to care about areas of expertise of others, the more efficient they are in modeling sub-systems in their own area. Furthermore, the process of studying complex systems using discrete-event computer simulations involves several areas of non-system expertise, such as discrete-event techniques or experiment planning.

This technical report serves two objectives: first it is a position paper in which we argue that ensuring a strong separation of the end-user roles is required to ensure a successful cooperation of all the experts involved in the process of simulating complex systems; second it introduces the Open Simulation Architecture (OSA) and describes how this architecture enforces such a strong separation of the roles and expertise areas.

Moreover, the OSA architecture is intended to meet the expectations of a large part of the discrete-event simulation community. This report also describes the way OSA provides an open platform intended to support simulationists in a wide set of their simulation activities, and how it allows the reuse and sharing of system models in the simulation community by means of a flexible component model (Fractal).

Key-words: Discrete-event simulation, Component-based Modeling, Open Software Platform, Separation of Concerns, Shared Component, Eclipse, Middle-ware, Fractal Component Model, Client-Server Modeling, Simulation Model Reuse, Software Reuse, Simulation Engine.

Ces travaux sont financés conjointement par l'INRIA et l'ANR (projet ANR OSERA)

This work is co-supported by INRIA and the french National Research Agency (OSERA ANR grant)

* Olivier.Dalle@sophia.inria.fr

OSA: Une architecture ouverte à base de composents pour la simulation à événements discrets

Résumé : La modélisation à base de composants a un certain nombre de qualités bien connues. La capacité à répartir le travail de modélisation entre différents experts ayant chacun leurs domaines d'expertise respectifs est l'une de ces qualités. Clairement, moins un expert doit se préoccuper d'autres domaines d'expertise que le sien, plus il est efficace. De plus, le processus d'étude de systèmes complexes à l'aide de simulations informatiques à événements discrets implique un certain nombre de domaines d'expertise non systèmes, comme les techniques portant sur les événements discrets ou les plans d'expériences.

Ce rapport de recherche vise deux objectifs : d'abord défendre l'idée qu'une forte séparation des rôles des utilisateurs est indispensable pour garantir la réussite de la coopération entre les experts impliqués dans l'étude par simulations d'un système complexe ; ensuite introduire l'*Open Simulation Architecture* (OSA), et décrire comment cette architecture permet d'assurer une telle séparation des rôles des utilisateurs finaux.

En outre, l'architecture OSA est conçue pour répondre aux attentes d'une grande partie de la communauté de la simulation à événements discrets. Ce rapport décrit de quelle façon OSA fournit une plate-forme ouverte pour assister les experts de la simulation dans une grande partie de l'ensemble de leurs activités liées à la simulation et de quelle façon il permet la réutilisation et le partage de modèles de systèmes à l'aide d'un modèle de composants flexible (Fractal).

Mots-clés : Simulation à événements discrets, Simulation à base de composants, Plate-forme logicielle ouverte, Séparation des préoccupations, Composant partagé, Eclipse, Intergiciel, Modèle de composants Fractal, Modélisation client-serveur, Réutilisation de modèles de simulation, Réutilisation de logiciel, Moteur de simulation.

1 Introduction

An interesting property of component-based approaches for modeling and simulations of discrete events systems is the ability to model complex systems by dividing the initial system, recursively, into smaller sub-systems. Out of the most stated benefits of this approach[15], one is that the process of modeling becomes easier as the complexity of the sub-systems decreases. But another less stated benefit is that it allows to dispatch the modeling effort over several system experts, each having their own area of expertise. For example, modeling a satellite communication system involve expertise in orbitography, satellite platforms, radio-frequencies, medium access protocols, antennas, transport protocols, user-level traffic, and many other areas. Modeling such a complex system necessarily implies the collaboration of several experts. Furthermore, apart the expert knowledge required to model the various subsystems of a complex system, one has also to consider the expert knowledge required in order to conduct an experimental study based on simulation, as well as the expert knowledge required in discrete-event modeling techniques.

Indeed, studying a system using discrete-event computer simulations imply several activities[2, 18, 12]. The Open Simulation Architecture aims at supporting a large number of these activities, and especially the following ones:

- **Conceptual model specification** of the system under study;
- **Architectural design of the software implementation** of conceptual model;
- **Software implementation** of the sub-systems models resulting from the architectural design;
- **Simulation scenarios configuration**;
- **Instrumentation of simulation scenarios**: configuration of probes to collect experimental data, definition of experiment plans;
- **Configuration of computational resources** to be used for the simulation run(s);
- **Control of simulation run(s);**
- **Post-processing and analysis** of simulation results.

Building a new specific software platform to support all these activities would obviously require a considerable development effort. Furthermore, this effort would be partly pointless considering the facts that (i) a large number of software tools already exist to support some of these tasks and (ii) simulationists are already accustomed to use these existing tools.

Hence the philosophy of the Open Simulation Architecture we introduce in this technical report, as its name suggests, is to offer an open architecture in which simulationists may easily plug and reuse the software pieces they need for their simulations.

Our analysis of the needs for such an architecture lead us to identify two kinds of contribution and reuse patterns: (i) contribution and reuse of software tools, and (ii) contribution and reuse of software simulation models.

The OSA architecture is intended to deal with both of these two patterns. For the first one, this is achieved by merging existing open-oriented software and platforms, such as Eclipse[7, 6] for development activities, middle-ware layers such as ProActive[3] and federating protocols such as HLA[11]. For the second one, this is achieved by using a flexible and open component architecture: the Fractal component model[5].

Nevertheless, as the software contributions will increase, the global consistency of the architecture, and therefore the reliability of its expected outputs, may become a challenging or even an intractable question. One of the key concerns of the OSA to face this challenging issue is to enforce a strong separation of end-user roles.

In section 2 we first summarize the key features of the Fractal component model used in the OSA architecture. Then, through a simple modeling case study, section 3 illustrates how the Fractal model helps to enforce the previous objectives. Eventually, we describe the overall architecture of OSA and some of its key internals in section 4.

2 The Fractal Component Model

Fractal is the ObjectWeb Consortium component reference model[4, 5]. Fractal is *neither* a software environment *nor* a runtime executive. It is a specification. In other words, it is a set of rules and features that a component-based software architecture is supposed to follow or implement in order to be compliant with this model. Fractal does not mandate the use of any specific programming language. On the contrary, it allows to combine component implementations possibly based on different programming languages.

The Fractal specification defines several levels and sub-levels of compliance. These levels allow an implementation not willing or not able to implement completely the model to indicate how much of the specification it comply with. At the lowest level, a component architecture claiming to be compliant with level 0.0 is just supposed to implement its components using the object programming paradigm. At the highest level, a component architecture claiming to be compliant with level 3.3 is supposed to fully implement all the features of the specification.

Hereafter, we summarize some of these key features (see [5] for the complete description).

Component external structure. A Fractal component is an object-oriented unit of code that has external interfaces. These interfaces may be of two kinds: either *client* or *server*. The former emits service requests, the latter receives service requests. Interfaces are named. Their name must be unique for a given component but names may be reused for naming interfaces in other components. A client interface is intended to be bound to a server interface.

Hierarchical structure Components may have a hierarchical structure (fig 1). Hierarchical components are made of a controller part (also called membrane) and a content part. The content part is composed of one or more components. Since a controller and its content recursively form a component it may have external interfaces. It may also have

internal interfaces. As external interfaces, internal interfaces may be either of type client, or of type server. Internal interfaces are only available to components of the content part. A component of the inner part may only bind its external interfaces to external interfaces of other inner components or to the inner interface of its surrounding controller. Therefore, the model strictly forbids a component to bind its external interfaces to the ones of components outside its controller or inside its neighbouring (inner part) components.

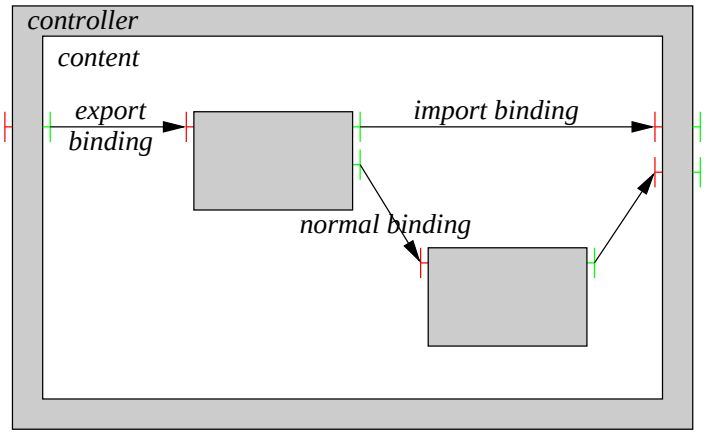

Figure 1: Example of Fractal hierarchical component.

Introspection Introspection is the ability for an object to collect useful information about other objects (possibly including itself). In the Fractal model, components have the ability to introspect their interfaces. For example, a component may retrieve its own list of available internal and external interfaces. Depending on the component type, they may also have other introspection capabilities, such as inner content architecture for composite (hierarchical) components.

Functional and controller interfaces A functional interface is an interface used to offer or obtain services to or from other components. A controller interface is a server-only interface that is offered to a component to access non-functional services, such as introspection, (re)configuration, persistence, service policy, life cycle control (ability to start/stop a component), and so on.

Factories and templates A factory component is a component that has the ability to create other components. Fractal distinguishes two kinds of factories: generic factories, that have the ability to create several kinds of components, and standard component factories, that only have the ability to create one kind of component. Templates components are a

special kind of standard factory components, that may be recursively composed of factories, and serve as a model to create normal components in a quasi isomorphic manner (isomorphic meaning the created component has the same hierarchical structure as its creator template). Since factories are components and components are created from factories, a special component is required to initiate the recursion. This special component is a generic component factory called "bootstrap".

Controllers *are* components. As suggested in figure 1, the membrane surrounding each component, the controller part, has some "thickness". Indeed, each membrane may embed arbitrarily complex non-functional services. This is the reason why the Fractal model requests these membranes to be components themselves. This reflexive property allows the controllers to benefit from all the good properties of the component model. For example, one could build a membrane component hierarchically, by embedding the services of each non-functional interface in a separate component: a persistency component, a life cycle component, and so on.

Shared components The Fractal model allows a component to appear in the content of several distinct enclosing components. Such components are called *shared components*. This property has two noticeable consequences: (i) a shared component is placed under the control of several surrounding controller components and (ii) a shared component may directly interact with components located in the inner parts of several distinct components.

3 A simple modeling case study

This case study serves two objectives: (i) provide an example to illustrate the OSA concepts explained in section 4, and (ii) demonstrate the usefulness of the concept of shared component in this context.

In section 3.1 we first give a conceptual model of the system under study. Then, in section 3.2, we present and discuss possible component implementations of this model with and without shared components.

3.1 Conceptual model

The conceptual model is the view of the system the expert has in mind. In a first step of formalizing the model, the expert may express this view through a specification, using formalisms such as the Unified Modeling Language (UML)[14], or Data Flow Diagrams (DFD)[17].

Interactions diagram Figure 2 gives a conceptual view of our example system using a very simple form of DFD in which we just show data flow interactions between system entities.

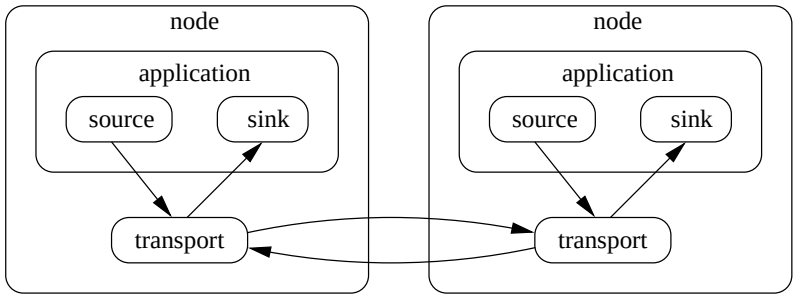

Figure 2: Simple system conceptual model.

Constitutive elements. The simple system is composed of two identical nodes that communicate with each other using the message passing paradigm. Each node of the system is decomposed in two sub-systems: the application and the transport. The application sub-system, itself, is decomposed in two sub-systems: the source and the sink.

Transport **sub-system.** The transport sub-system implements a reliable, ordered, connectionless, datagram routing and delivery service. It supports variable but limited datagram sizes. Its Application Programming Interface (API) provides the following operations and constant:

- `send(databuf,datalen,node_id,appli_id)`: send the `buflen` bytes of the `databuf` buffer to application `appli_id` of node `node_id`. It blocks the caller until the operation completes and returns a boolean indicating whether the operation was successful or not.
- `recv(databuf,datalen,from_id,from_appli)`: request reception of a maximum of `buflen` bytes into `databuf` buffer. `appli_id` and `node_id` are return values indicating the source of the message. It blocks its caller (possibly forever) until it completes. Upon completion, it returns the received packet size. It may not fail.
- `MAX_PKT_SIZE`: constant parameter of the model indicating the maximum packet size.

The implementation details of this sub-system do not need to be further described.

Application sub-system. The sink sub-system repetitively invokes the `recv` operation of the transport and silently discard any incoming datagram.

The source sub-system simulates file transfers. It repetitively do the following actions:

1. sleep for a random time chosen in the interval `[0;MAX_SLEEPING_DURATION]`;
2. pick a random file length `flen` in the interval `[0;MAX_FILE_SIZE]`

3. Call $\lceil \frac{flen}{MAX\_PKT\_SIZE} \rceil$ times the transport's `send()` service to send fragments of at most `flen` bytes.

3.2 Component implementations

In the following, we discuss two particular component implementations of our system example: a first one that does not use shared components and a second one that does use them.

3.2.1 Component implementations without shared components.

The first implementation that comes to mind is the one shown on figure 3.

It's main quality is to be structurally very close to the conceptual model. This good property is achieved by applying a very simple strategy in order to reflect interactions that need to cross sub-systems boundaries: replicate the client and server interfaces of the interacting components on the internal and external sides of any surrounding membranes that need to be crossed.

However, one shall notice the slight difference between the bindings of this implementation and the data-flow interactions of the conceptual model: in the conceptual model the sink component receives data while in this implementation it initiates the reception service call. Indeed, this is a common practice in protocol APIs that avoids buffering on the reception side: either a reception call is pending and the incoming data is delivered, or the incoming data is discarded.

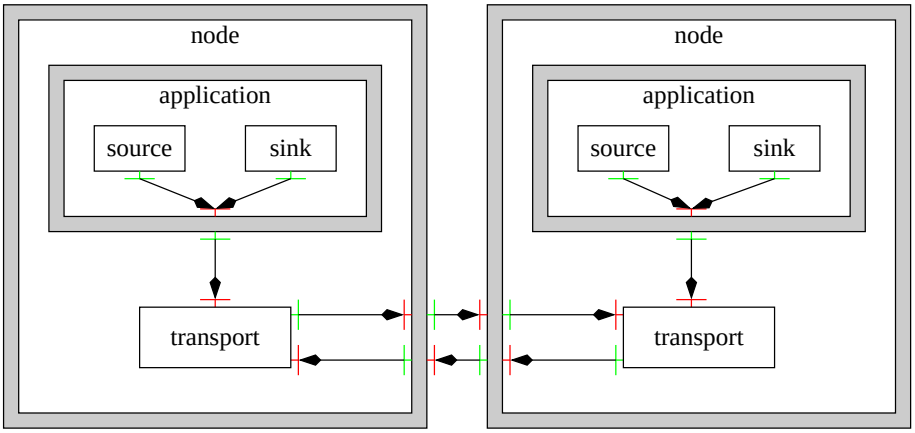

Figure 3: Component implementation without shared component

In this example, the consequence of the "replicate interfaces" strategy is that a node component have to expose an external transport interface. Therefore reusing node components

in another system model would require a minimal understanding of this transport interface which contradict our design objectives.

For example, suppose that an expert in car mechanics needs to model a new car including an communicating device for exchanging files between its car and some server through a wireless network. This is reasonable because the node model was designed in a generic way, that is without any assumption on the underlying physical network used by the transport subsystem. So, the node component could be reused by this expert to model this communicating device. Since the node is now part of a car conceptual model, the expert would obviously plug it somewhere inside its car components, which raises the following problems:

1. the expert needs to apply in turn the simple strategy and thus "pollute" its car component with a new external interface, as well as several inner components, depending how deep inside the hierarchy the new component is plugged ;

2. the expert has to find correct bindings and settings for the new external interface which may be a complex task for someone that is not supposed to be an expert in networks.

Another solution that would avoid the car expert this resulting complexity of plugging a new component *inside* its car component would be to let the node component reside *outside* the car component. But this first violates the conceptual model of the car and second this approach raises new problems such as external dependencies between components. For example, replacing the component of the car that have the electronic device with another component modeling a car that has not such a device leads to an incoherence in the system model as long as the node device is not removed.

Furthermore, it is worth stressing that the simple "replicate-interface" strategy has several other negative effects that contradict the very fundamental philosophy of component-based design. For example, let's consider the evolutivity good property of the component approach. The component-based approach is expected to ease the replacement of a component (when new improved versions of the component are released, for instance). For example, let us consider the following possible evolution of the node component: a new sub-component modeling the physical network layer is added as shown in figure 4. Since interactions between transport sub-systems have been replaced by interactions between physical layer components, the component implementation of the new conceptual model implies modifications of all the surrounding components.

3.2.2 Component implementation with shared components.

The first implementation with shared component that comes to mind is the one depicted in figure 5. In this implementation, a single shared component is used in place of the replicas of the transport component of the conceptual model. This obviously solves the problem of interactions crossing membranes of surrounding components. Since transport components are now implemented through a single instance, this unique instance can easily interact with itself without crossing membranes. Nevertheless, this implementation does

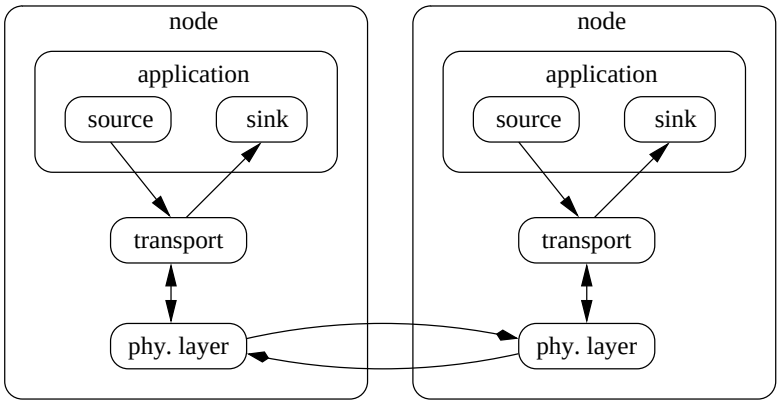

Figure 4: A possible evolution of the simple system

not strictly preserves the intent of the conceptual model. Indeed, the conceptual model implies a distributed architecture of the transport components (one instance in each node component) while the unique shared component architecture of figure 5 implies a centralized architecture. This semantic alteration of the conceptual model may have significant impact both on the implementation of the component and on its control. For example, let us assume that a transport component is supposed to consume processing time in behalf of one of its surrounding component (the node component for example). This simple use of the shared component feature makes it difficult or even impossible to decide on behalf of which surrounding component the processing time should be accounted.

Nevertheless, this issue may easily be solved by using the construct shown in figure 6 in place of the single transport shared component of figure 5. With this construct, the transport component is built hierarchically and contains two sub-components: the one named proxy is shared and the other, named local part is not. Therefore, the whole transport component is not shared anymore, but only partially by means of the proxy component. Because the proxy component is shared, it may be used by the local part component as a communication proxy, as its name suggests.

As a conclusion about shared components, philosophically, one may wonder whether using shared component is good idea since it is a way of bypassing and somehow violating the components boundaries. It is worth stressing that in the context of simulation modeling, the isolation property of components may be considered from two point of views, which may be somehow confusing. The first point of view is the software engineering one and the second the simulation one. In [5], the authors claim that "paradoxically shared component are useful to preserve component encapsulation". This is a software engineering point of view and exactly what we demonstrated with the simple model case study. Since we intend to enforce a strong separation of roles, at the early stage of translating a conceptual model into a

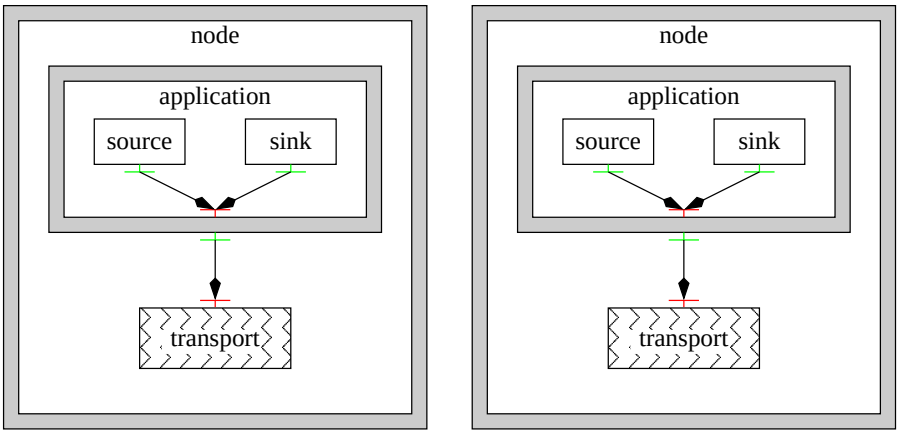

Figure 5: Component implementation with a single shared component

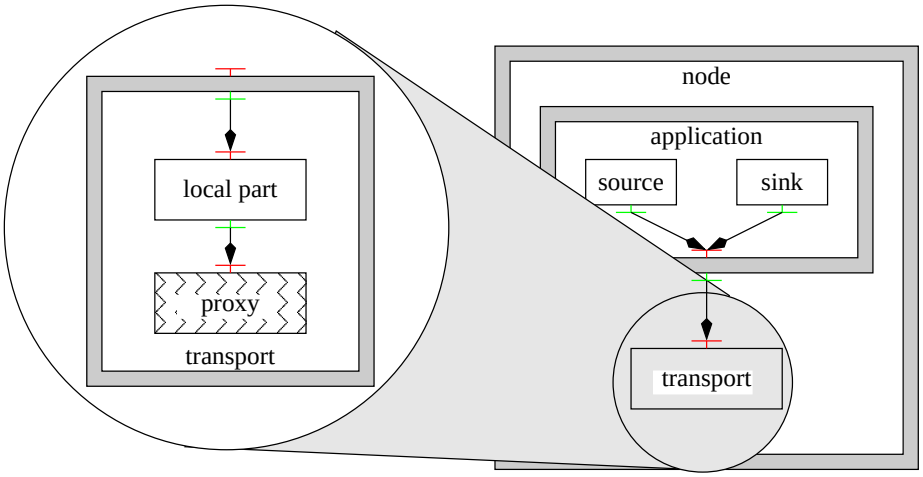

Figure 6: Alternative implementation with a single shared component

(software) component implementation, there is no reason to worry about anything related to the simulation point of view. Indeed, this early modeling task belong to the system modeler area of expertise, while the simulation of the model belongs to discrete-event simulationist area expertise.

Let us now consider the simulation point of view. Supporting the shared component modeling feature is clearly breaking the tree structure of the components. As stated in the

Fractal specification, the component structure with shared components may be a directed acyclic graph and bindings (interaction paths) may form cycles. Since interactions are eventually translated into events, shared components may imply nasty causal effects, that should be properly handled by the discrete event simulation engine.

4 The OSA architecture

The OSA architecture is intended to support users in a large number of tasks, identified in section 1. For this purpose, the OSA architecture distinguishes six classes of users:

- **model architects :** they focus on assembling components to reflect conceptual models;
- **model developers :** they focus on implementing the core part of components (functional services);
- **simulation-engine experts :** they focus on configuring and extending functionalities of the simulation-engine;
- **experimenters :** they focus on setting-up model parameters (scenario definition), data collectors (probes) and simulation runs;
- **analysts :** they focus on processing collected data (by probes) and produce an analysis of the results;
- **administrators :** they mainly focus on configuring software tools for proper utilization by other class of users (such has setting and managing model repositories for example).

This classification is intended to clarify the different roles a user (possibly the same) may have during the process of studying a system with discrete-event simulation. Based on this classification, specialized tools are designed in order to help users in the various tasks involved in each role. Since tools are specialized for a given task while the user play a given role, we achieve the strong separation of roles we are looking for.

Nevertheless, the overall architecture used to provide all these tools is implemented by means of a three layered N-tier architecture made of: a front-end user interface layer, a middle-ware executive layer and a simulation engine layer. Our intended contribution to these three layers are further described in the following sections 4.1 to 4.3.

4.1 Front-end user interface

The OSA architecture aims at providing a wide set of supporting tools to assist each class of end users. Furthermore, the architecture should enforce a strong cooperation of these tools, using an integrated and easily extensible environment. For this purpose, the Eclipse platform[7, 10, 6] perfectly match our needs.

Eclipse already provides a large amount of *plugs-in* to assist developers in the various task of software development: specification, core development in several languages (Java, C/C++, ...), unit testing and debugging, versioning control, repository management, and so on.

It is worth emphasizing that an interesting features of the Eclipse plugs-in is their ability to be extended. Indeed the Eclipse Plug-in API defines *extensions points*[9]: plugs-in that implement such extension points (which is not mandatory) may be extended in order to build new enriched or specialized versions of the base plug-in.

Our central guideline in the definition of the OSA architecture is to fully subscribe the Eclipse philosophy by offering the discrete-event simulationist community a truly open architecture that ease reuse of existing tools and encourage new valuable contributions for the community.

4.2 Middle-ware executive platform

N-tier architectures are often criticized for their potentially poor performance. Since performance is a critical issue for simulation, this architectural choice may be questionable. As a matter of fact, the middle layer is not mandatory in the OSA architecture. Indeed, in the Fractal component model, the distribution of component executions across a network through a middle-ware is an optional feature that may, or may not be activated. Since this is implemented as a non functional feature of components, the decision of activating or not this feature is totally transparent. In other words, it does not require any change in the component functional implementation. This is consistent with the idea that the selection of the centralized or distributed mode of execution is a task devoted to simulationist when it acts as an experimenter, not as a developer.

Supported patterns of distribution The OSA architecture allows the distribution of the simulation executions across the network in different manners:

- distribution of several simulation-runs, each one executing on a single computer node. In this case, the distribution support required is very limited (a "gang-scheduler" facility);
- distribution of one (or several) simulation runs across the network, simply using the Fractal model ability to distribute transparently the execution of the components, but *without any cooperation of the simulation engine.* Since the minimal requirement of the simulation engine, whether it executes in parallel or not, is to ensure consistency of event processing between components, this implements de facto the so-called *conservative* mode of parallel execution[8];
- distribution of one (or several) simulation runs across the network, using the Fractal model ability to distribute the execution of the components, and *the cooperation of the simulation engine.* Provided the components have the persistency non-functional

feature in order to regularly save their global simulation state, this lead to the so-called *optimistic* mode of parallel execution[8];

- the last form of distribution, which is somehow complementary of the previous ones, is achieved when the middle-ware is used to bridge together several simulation architectures, using the HLA standard for example[11].

The middle-ware platform selection is certainly a critical issue for performance, but this is not the only good property required. The middle-ware platform has to be well integrated with the global architecture and offer useful functionalities for a distributed mode of execution. For this purpose we foresee the ProActive platform[3] as a good candidate: it is fully compliant with the Fractal model, it has been specially designed for grid computing and thus offers advanced features such as dynamic load balancing, deployment facilities, firewall bypassing strategies and related security issues.

4.3 Back-end simulation engine

The simulation engine is not a single entity in itself. It is distributed over all component that have a surrounding membrane implementing the simulation non-functional services. These services are accessed through a dedicated `simulation-controller` interface. Component with a `simulation-controller` interface are called *active components* and those without such an interface are called *passive components*.

The various semantics of interactions between components are described in section 4.3.1. The way these semantics are implemented is further described in section 4.3.2. The queuing and scheduling policies are described respectively in sections 4.3.3 and 4.3.4. The small set of non functional services offered by the `simulation-controller` that need to be used explicitly by the components for simulation purpose are listed is section 4.3.5.

4.3.1 Semantics of interactions

The OSA simulation engine supports the three following semantics of interaction:

- Synchronous interactions: the service requested by the client is synchronous with the simulation time (no time consumption);
- Asynchronous interactions: the client is not blocked while the service is being processed by the server. This asynchronous mode of interaction implies the client shall not expect a meaningful return value from its service call (except an error).
- Blocking interactions: the client is blocked until the service is completed (or aborted) by the server. The client may expect a meaningful return value from its service call.

When one or both of the involved peers is a passive component, the only interaction mode available is the synchronous one.

4.3.2 Translation of interactions into discrete-events

As mentioned earlier in section 4.3.1, OSA components support three semantics for interactions: synchronous interactions, non blocking interactions, and blocking interactions. The first one is not translated into a discrete-event because the service execution is synchronous in the simulation time. The two others interactions are translated into discrete-events, provided that both peers are active components and thus each have a surrounding membrane with a `simulation-controller` interface.

The technique used for translating service invocations into simulation events was first introduced in [13]. It consists in using an object-oriented construction called a *functor*. A functor is the transformation of a method invocation into an object. Indeed, when a component issues a (functional) service request, through its client interface, its call is automatically intercepted by the controller part of the source component and *reified* into an event object.

This event object contains the current time in simulation, the event type, an object encapsulating the method called on the server side and its arguments (functor), and possibly other data that do not need to be further described.

In order to ensure a minimal inter-operability between components, the following rules are imposed :

- The particular event created at the time the service call is issued by the client is initialized with a type value of SOC (Start Of Call) and inserted in the event queue of the server interface;
- In case of a blocking service the particular event created at the time the service is completed by the server is initialized with a type value of EOR (End Of Reply) and inserted in the event queue of the client interface.

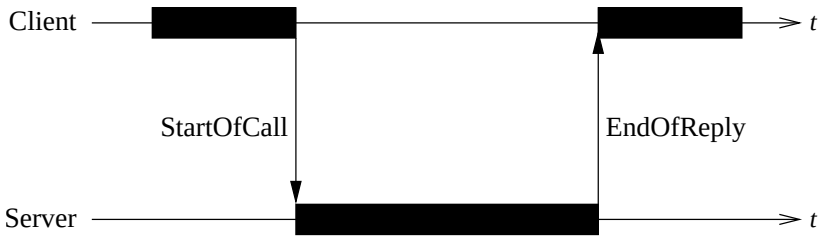

Figure 7: Temporal diagram of interactions (synchronous).

These constraints are intended to be strong enough to ensure inter-operability between components, but flexible enough to let the discrete-event simulation engine experts implement variations around the three models of interactions described in section 4.3.1.

In particular, these constraints allow additional events to be generated. For example, the server interface could be implemented so that it sends back an EOC event (End of Call)

to the client interface as soon as it decides the server has completely received the requests. This additional event may be used to simulate the time needed to issue a service call, which may have a significant meaning in the case of asynchronous interactions: instead of resuming its execution as soon as it has issued a service call, the client may be blocked as long as the service call is not complete (figure 8). In other words, using this event the service call may be split into two phases: the call phase, during which the client is blocked, and the service processing phase, during which the client is allowed to continue its execution.

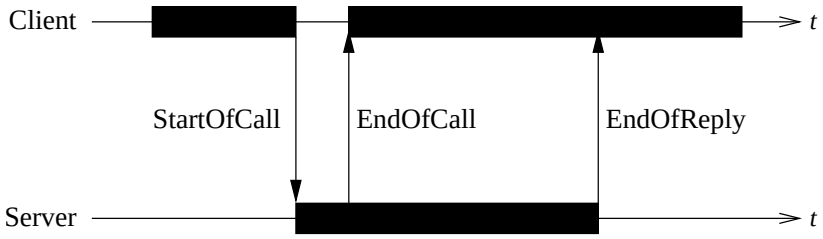

Figure 8: Temporal diagram of interactions with EOC event (asynchronous).

Despite it would solve the causal cycle issue induced by shared components raised in conclusion of section 3, the systematic generation of this event is not enforced by the simulation engine. Indeed, a causal cycle may occur when an event resulting (indirectly) from an action of given component at a given time of the simulation is received by the same component at the same time of the simulation (figure 9). This situation may only occur when (i) all the components involved along the cycle issue asynchronous calls and (ii) there is no time consumption between the time at which they serve the request of the previous component on the cycle and the time at which they issue the service call toward the next component on the cycle.

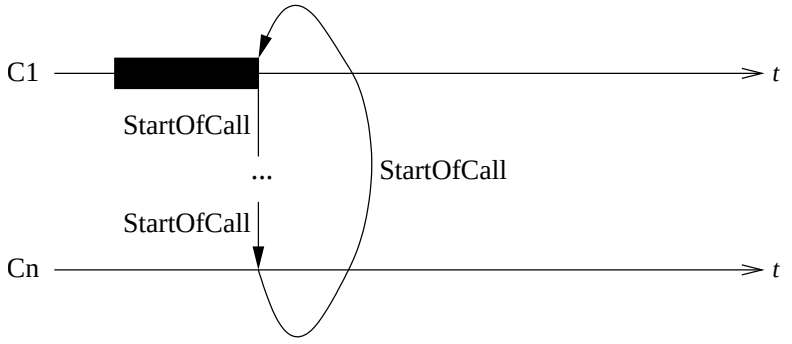

Figure 9: Temporal diagram of interactions with causal loop (asynchronous).

Enforcing a minimal delay to model the time consumption required to issue a service call is a way of properly handling causal cycle. But adding a new event (EOC) to implement this minimal delay obviously increases the number of events generated by the simulation engine. Since the duration of the simulation runs is directly impacted by the number of events generated, requiring the systematic generation of this new event would have a critical impact on the overall performance of the simulator. Thus the OSA simulation engine is designed to support such additional events, but do not enforce their systematic generation: this critical decision if left up to the simulation experts.

4.3.3 Event queueing policy

The event queueing policy is flexible. Indeed it may be replaced or reimplemented if needed, because event queues are Fractal components. Event queue components lay in the membrane component (as sub-components of the membrane component). A distinct event queue component is associated with each functional interface of a component. But since event queues are components and components may be shared, all the interfaces of a component may share the same wait queue. Or some may share a common queue and others may have their own. Sharing queues at a larger scale than a single component is even possible but the effect of sharing in this case is not specified[1].

4.3.4 Scheduling policies

The scheduling policy is flexible. As event queues, schedulers are components and lay in the membrane part of a component. Each active component of the simulation has a scheduler. But schedulers may be shared amongst several components. The scheduler manage the current execution state of the *processing resource(s)* of a component. A typical scheduler for a mono-processor type of processing resource would support the following states :

- `INIT`; the component is in its initial state and awaiting for its `start()` service to be called;
- `IDLE`: the component is sleeping and waiting for an incoming request;
- `RUNNING`: the component is consuming processing time because it is servicing a request;
- `BLOCKED`: the component has issued a blocking service call and is waiting for the EOR event (service completion).

Scheduling case study with a simple interaction model A typical blocking interaction between a client and a server component could be the one depicted on the temporal diagram of figure 10: while in the `RUNNING` state, the component on client side issues a service call. This service call is translated into an SOC event and inserted in the event

[1] This is up to the discrete simulation expert to implement a coherent behavior of event queues in this case.

queue of the server. In this particular example, the server processing resource is in the `IDLE` state at the event is received, meaning it is not processing any other service. Therefore the scheduler may immediately change the state of the processing resource to `RUNNING` and issue the service call embedded in the event (thanks to the functor construction). When the service call ends, the scheduler sends back an EOR event to the client interface. In this particular example, the scheduler on the server side has no further service request pending and thus, puts back the processing resource in `IDLE` mode. On the other side, the scheduler of the client side, may resume the client execution by delivering the result of the service and putting back the processing resource of the client in the `RUNNING` state.

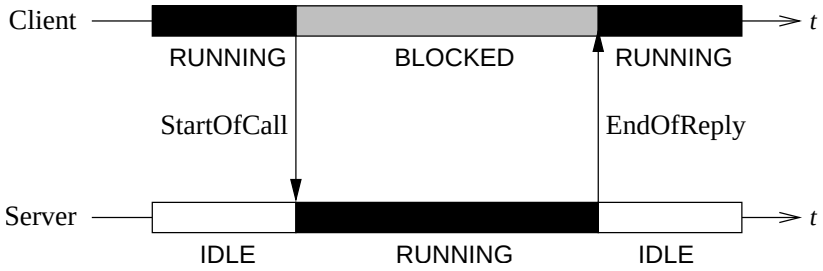

Figure 10: Temporal diagram of scheduling states during a blocking service call.

It is worth emphasizing two things about this example:

1. the number of schedulers involved in this example is unspecified. Therefore, the client and server could share the same scheduler component, meaning they only have a single shared processing resource; or they could each have their own processing resource by means of two distinct schedulers;

2. schedulers and event wait queues have to be implemented using a concurrent flow of control (typically using a threads language library). Indeed, while the processing resource *simulated* on the client side is in `BLOCKED` state, the event queue(s) on the client side should still be able to insert properly any new incoming event. The same constraint applies to the `RUNNING` state, and therefore to the server, that should still be able to handle new event insertion requests while a service call is being serviced.

Simulating parallel processing resources In order to simulate a component having N parallel processing resources, one should allow at most N requests to be serviced in parallel at any time and queue any new incoming service request. A way of implementing this parallel behavior is to implement the scheduler has a hierarchical component embedding several sequential schedulers and a *super-scheduler* in charge of dispatching processing requests (SOC events) amongst each scheduler. In this case, the resulting multi-processing facility scheduler should either provide synchronization primitives to the inner part of the

component, such a semaphores or spin-locks, to let them explicitly enforce protection of their critical code sections or simply forbid service re-entrance[16].

The special case in which the number of parallel processing resources N_p is equal to the number N_s of services provided by the component *and* service re-entrance is forbidden should also be considered: in this case, since there is no contention for the `RUNNING` state, the scheduler implementation is noticeably simplified. In case re-entrance is allowed, a similar simplified implementation may be considered as soon as number of processing resources is unlimited ($N_p = +\infty$).

Synchronizing policy As soon as several schedulers act concurrently, a synchronizing policy has to be implemented in order to enforce simulation time consistency. This synchronizing policy should be implemented using an additional synchronization component that also lays in the membrane component (typically as a sub-component of the scheduler). The protocol between the synchronization component and the scheduler component is very simple: the scheduler calls the `next_event_date()` service of the synchronizing component with the date of its next event as a parameter. The synchronizing component replies to this call with the minimum of the next events dates, as soon as all the schedulers have called the `next_event_date()` service.

In order to implement a conservative policy, it is sufficient that:

- every scheduler uses this protocol every time it has a new event to process and sets its current time to the value returned by the `next_event_date()` call.
- the synchronizing component is a unique component shared by all the active components.

A conservative policy is still possible in case several synchronizing components are used instead of a unique shared synchronizing component. But in this case, the global synchronization of the synchronization components have to be enforced, which would require an additional higher level synchronizing component. By applying this synchronizing scheme recursively, one may build a synchronizing hierarchy with an arbitrary depth (figures 11 et 12).

4.3.5 Explicit simulation services

During the simulation, the functional services (the service that implement the conceptual model of the system) may use the following services offered by the non-functional `simulation-controller` interface of their surrounding component:

- `current_time()`: returns the current simulated time;
- `abort()`: requests abortion of the simulation execution because of an abnormal execution condition;
- `terminate()`: requests normal termination of the simulation execution;

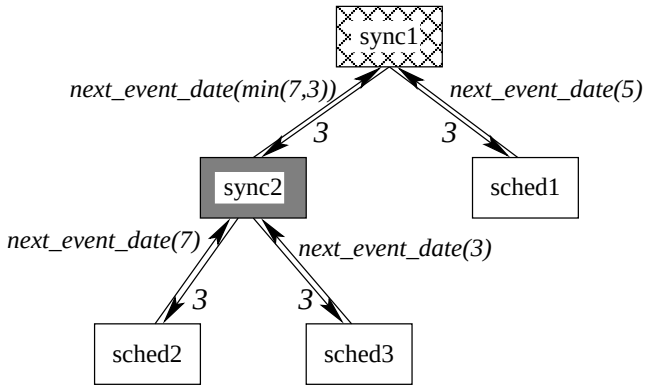

Figure 11: Synchronizing hierarchy (hierarchical tree view)

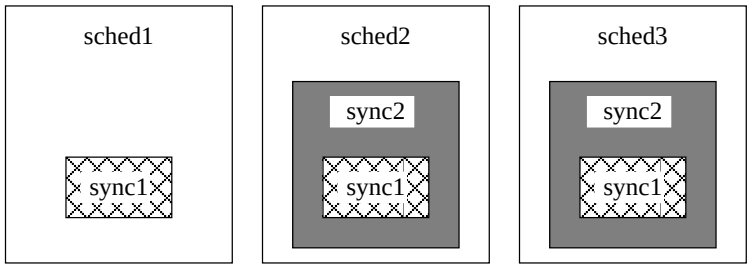

Figure 12: Synchronizing hierarchy (flat view)

- `wait(delay)`: requests the component to stay in `RUNNING` state until the simulation time reaches `current_time()+delay`;
- `wait_until(time)`: requests the component to in `RUNNING` state until the simulation time reaches `time`.

5 Project status

Since OSA is an open platform, we intend to mainly focus on the very core parts of OSA: the simulation engine, a graphical user interface to assist in model component development, assembly, and instrumentation, and a public repository of model components.

Simulation engine Several implementations of the Fractal component model are already available from the ObjectWeb Consortium[2]. The simulation engine detailed interface specifications and an early implementation provided for testing purpose, based on one of the Java implementations of the Fractal model, are planned to be released in the first quarter of 2006. A robust and optimized Java implementation and preliminary C++ implementation are expected to be released before the end of 2006.

Graphical User Interface Tools for assembly and exploration of Fractal components are already available from the ObjectWeb Consortium; Eclipse plugins to assist in Fractal component developments are also expected to be soon available from the ObjectWeb Consortium. These generic Fractal tools are already helpful for OSA model components preliminary development and assembly. Augmented versions of these tools and plugins for OSA are planned to be released in the second quarter of 2006. Nevertheless, some parts of the Graphical User Interface that are very specific to OSA, such as simulation scenarios settings and instrumentation, need more developments and may not be released until the end of 2006.

Model repository In order to ease reuse and exchange of OSA model components, a public repository has to be set up. Nevertheless the definitive design and architecture of this repository still need further work. The first step of this design effort is to define a packaging specification for model components. Indeed, reusing and exchanging software components raises common critical issues, such as those related to software documentation, versioning, and dependency solving. Similar issues have found dedicated solutions in several communities: in the Linux community, for example, in which "distribution" vendors each provide packaging support and repositories; others examples may be found in various specific software communities such as the Perl language and LaTeX text processing users communities with their respective package and repository systems (CPAN and LTAN). In the particular case of Fractal component based software architectures, these issues are currently under

[2] http://fractal.objectweb.org/

study (see [1] for example). Therefore, we expect practical solutions to emerge soon and serve as a basis to the design of the future OSA repository and packaging architecture.

6 Conclusion

In this technical report we introduced the Open Simulation Architecture and demonstrated how this architecture could meet the expectations of a large part of the discrete-event simulation community. Indeed, the OSA architecture aims at (i) providing an open platform that support simulationists in a wide set of their simulation activities, and (ii) allow the reuse and sharing of system models by means of a flexible component model.

We also demonstrated how this architecture helps in solving the critical issue of integrating numerous software contributions into a single open platform. The issue is addressed by enforcing a strong separation of the roles of the OSA end-users, a well-known object-oriented software design pattern called *separation of concerns*.

In OSA, this design pattern is implemented by means of the Fractal component model. This component model relies on a client-server model of interactions. However, it is worth emphasizing that this particular model of interactions is not restrictive. Indeed, popular modeling formalisms, such as Petri nets or DEVS, mays easily be constructed using an underlying client-server approach.

We also exhibited and discussed some of its interesting features, such as the concept shared components. Nevertheless, further investigations need to be conducted around this shared component concept. On one hand, its combination with usual features of component models, such as the hierarchical structure, and the ability to easily replace one component by another offers a wide area of experimentation to the discrete-event simulation experts. Indeed, this allow to easily implement and explore new simulation policies and architectures, in which key components, such as the scheduler, may be totally or partially shared amongst component. On the other hand, it raises new issues that need to be properly handled, such as concurrency of control and potential causal loops.

References

[1] M. Alia, R. Lenglet, T. Coupaye, and A. Lefebvre. Querying reflexive component-based architectures. In *30th EUROMICRO Conference*, pages 127–134. IEEE, 2004.

[2] J. Banks, editor. *Handbook of Simulation Principles, Methodology, Advances, Applications, and Practice.* Wiley-EMP, 1998.

[3] F. Baude, D. Caromel, L. Mestre, F. Huet, and J. Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proc. of the 11th IEEE Intl. Symp. on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[4] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. In *Seventh Intl. Workshop on Component-Oriented Programming (WCOP02)*, ECOOP2002, Malaga, Spain, June 2002.

[5] 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.

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

[7] Eclipse Platform: Technical Overview, 2003. http://www.eclipse.org/whitepapers/-eclipse-overview.pdf.

[8] R. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.

[9] E. Gamma and K. Beck. *Contributing to Eclipse: principles, patterns, and plugs-in.* The Eclipse series. Addison-Wesley, 2004.

[10] S. Holzner. *Eclipse.* O'Reilly, May 2004.

[11] IEEE-SA. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA), Federate Interface Specification.* Std 1516.1-2000.

[12] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley, 1991.

[13] P. Mussi and G. Siegel. The PROSIT sequential simulator: a test-bed for object oriented discrete event simulation. In *Proc. of 7th European Simulation Symposium*, Erlangen, Germany, October 1995.

[14] Object Management Group, Framingham, Massachusetts. *UML 2.0 Superstructure Specification*, Oct. 2004.

[15] N. Oses, M. Pidd, and R. J. Brook. Critical issues in the development of component-based discrete simulation. *Simulation Modelling Practice and Theory*, 12:495–514, 2004.

[16] A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 2nd edition, 2001.

[17] E. Yourdon. *Modern Structured Analysis.* Prentice-Hall, 1989.

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

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399