Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

DISTRIBUTED SIMULATION WITH COTS SIMULATION PACKAGES

Csaba Attila Boer

Erasmus University of Rotterdam
Faculty of Economics
Dept. of Computer Science
P.O. Box 1738
3000 DR Rotterdam, THE NETHERLANDS

Alexander Verbraeck

Delft University of Technology
Faculty of Technology, Policy and Management
System Engineering Department
P.O. Box 5105
2600 GA Delft, THE NETHERLANDS

ABSTRACT

Connecting COTS (Commercial-off-the-Self) simulation packages entails various difficulties. First of all, commercially available simulation packages hide the access to some internal data that is needed to connect to other simulation models in the distributed simulation study. Next, the data sharing between simulation models is complicated. In order to carry out distributed simulation studies applying COTS simulation packages, we have to exactly define the interfacing and data transfer mechanisms. In this paper we present a theoretical description of different solutions for interfacing and transferring data between various simulation models. These mechanisms have been implemented and tested, and applied in a project in order to prove their practical usage.

1 INTRODUCTION

Large-scale complex simulation studies often involve different organizations; consequently these simulation studies can be considered as inter-organizational studies. Usually each organization develops its own simulation model of its system using a COTS simulation package that fits its activity (e.g. manufacturing, transportation, supply-chain, etc.). When carrying out a study that looks at joint or related activities of more than one organization, and we would like to reuse the already developed models, the individual simulation models of their systems will also need interconnections (e.g. in a supply chain simulation study a factory simulation model has a relation with a warehouse or distributor simulation model, namely the product and cash flow). In order to improve the performance of the whole system we would like to analyze the overall behavior of the inter-organizational system. To achieve this we must analyze the inter-organizational system as a whole. In this way the whole simulation model of a large-scale complex system with the several involved organizations is considered as a collection of coupled simulation models, where each simulation model represents a system of one organization. This leads to a distributed simulation study. In this paper we aim to analyze the various possibilities of interoperation between the simulation models that have been developed in different COTS simulation packages.

The interoperability in distributed simulation involves at least the *data transfer* and the *time synchronization* between the simulation models. The data transfer itself entails actions related to:

- the exchange of an occurrence (e.g. signal) or an event with other simulation models,
- simulated entity transfer from one simulation model to another.

In the HLA (High Level Architecture) (Defense Modeling and Simulation Office 1996) terminology the data that is transferred for the first action is called an *interaction* (that can carry a set of parameters), while the data transferred for the second action is called an *object* (that carries attributes of the entity objects). A detailed description of different time synchronization mechanisms can be found in (Fujimoto 2000).

In this paper we would like to concentrate mainly on the data transfer. Before being transferred to other models, the data must be accessed. The way of accessing data in COTS simulation packages is usually restricted. In section 2 we investigate the possible ways of accessing data in simulation models developed in COTS simulation packages and introduce the concepts of interface functions. Section 3 describes the concept of the interoperability function, which together with the interface function forms a wrapper that makes possible to connect the COTS simulation model to a distributed simulation architecture. Section 4 focuses on the problems that might occur when we try to connect two COTS simulation packages, specially focus on the situation when the type of the entity attributes differ. Based on the theoretical description given in Section 2, 3 and 4, Section 5 includes the practical usage of these concepts, and finally Section 6 concludes.

2 INTERFACING COTS SIMULATION MODELS

The concept of discrete event simulation modeling is well defined by the DEVS formalism (Zeigler et al., 2000). This is an impressive result and it provides an accepted framework and theoretical foundation in the area of simulation. Most of the COTS discrete event simulation packages are based on this concept.

Although the already existing concepts give us a good starting point to specify the interfacing of COTS simulation packages with other COTS simulation packages, we need to define further concepts as well. First let us introduce the following notation:

$M \triangleright P$, means a simulation model *M* is designed and developed in package *P*.

A simulation model M at a certain time t can have a collection of simulated entities $\widetilde{E}_t$, which are instances of the abstract simulation entities. $\widetilde{E}_t = \{E_{i,t}\}_{i=0}^{n}$, where $E_{i,t}$ represents the ith entity instance in the model M at time t.

$A_{i,t} = \{a_{i,t}^{j}\}_{j=0}^{m_i}$ is the *collection of attribute values* of entity instance $E_{i,t}$ in the simulation model *M* at time *t*, where, $i \in [0,n]$, and $n \geq 0$, $m_i \geq 0$.

Definition 1. Let $\widetilde{A}_t = \bigcup_{i=0}^{n} A_{i,t} = \{a_{i,t}^{j}\}_{i\in[0,n],\, j\in[0,m_i]}$ be the *collection of all attribute values* that belongs to the simulation model *M* at time *t*, where, $n \geq 0$, $m_i \geq 0$.

In order to achieve interoperability these attributes (and their entities) must be accessible outside the simulation package. Therefore we define the *access function* that specifies if an attribute is accessible or not. This function is defined as a characteristic function.

Definition 2. An *access function* is defined as

$$F : N \times N \rightarrow \{0,1\}$$

$$F(k,l) = \begin{cases} 0, & \text{if } a_{k,t}^{l} \text{ is not accessible at time } t \\ 1, & \text{if } a_{k,t}^{l} \text{ is accessible at time } t \end{cases}$$

This function is defined if $k \in [0,n]$ and $l \in [0,m_k]$.

Definition 3. Let $\hat{A}_t = \{a_{k,t}^{l} \in \widetilde{A}_t \mid \exists F : F(k,l) = 1\}$ be the *collection of all accessible attributes* at time t.

A simulation package can be *fully open*, *partly open* or *fully closed*. Let us define these concepts using the previous definitions.

Definition 4. A simulation package *P* is called *fully open* in any simulation time *t* if $\forall M \triangleright P$, $\hat{A}_t = \widetilde{A}_t$.

Definition 5. A simulation package *P* is called *fully closed* in any simulation time *t* if $\forall M \triangleright P$, $\hat{A}_t = \emptyset$.

Definition 6. A simulation package *P* is called *partly open* in any simulation time t if $\forall M \triangleright P$, $\hat{A}_t \neq \emptyset$, $\widetilde{A}_t \neq \hat{A}_t$.

Most of the currently available COTS simulation packages are partly open. They do not offer a direct access to the attributes, but offer interfaces through which some of the model attributes are accessible. These interfaces can be defined as a function for each simulation package.

Definition 7. An *interface function* for accessible attributes is defined as $G : \delta \rightarrow \hat{A}_t$, where δ is a set of references. Based on the set of references using the G function we can access an attribute at time *t*.

If we define $\delta = N \times N$, then $G : N \times N \rightarrow \hat{A}_t$, $G(k,l) = a_{k,t}^{l}$, $\forall$ simulation time *t*. This function is defined if $k \in [0,n]$, $l \in [0,m_k]$.

There is a coherence between the *access function* and *interface function*. That is, if the attribute of an entity instance is not accessible then it can not have an interface function. While the access function provides a possibility to setup the attributes accessible, the interface function gives the possibility to access them.

In a simulation model we can distinguish *fixed* and *moving* entities. During a simulation run the fixed entities are bound to a certain location, while the moving entities can move through the model. Usually these moving entities (e.g. a product entity) are created and are destroyed by bounded entities (e.g. a manufacture or destroy entity).

In order to illustrate the previous concepts we take a simple example. To start with, we design and develop a model *M* in package *P*. Suppose we use three abstract entities: truck, generator and workstation. The model M at simulation time *t* has four entity instances (E_0, E_1, E_2 and E_3): two fixed entities (E_2 and E_3) and two moving entities (E_0 and E_1). The fixed entities are the stations, in our case a generator (E_2) and a workstation (E_3). The moving entities are the trucks that after generation process move to the workstation. Table 1 gives the attribute values of the entities at time *t*.

Based on this information the collection of entity instances are: $\widetilde{E}_t = \{E_{0,t}, E_{1,t}, E_{2,t}, E_{3,t}\}$. The attribute values for each entity are:

$$A_{0,t} = \{a_{0,t}^{0}, a_{0,t}^{1}, a_{0,t}^{2}\}$$
$$A_{1,t} = \{a_{1,t}^{0}, a_{1,t}^{1}, a_{1,t}^{2}\},$$
$$A_2 = \{a_{2,t}^{0}\},$$
$$A_3 = \{a_{3,t}^{0}\}.$$

Table 1: Attribute Values at Simulation Time *t*

Entity Instances	Attribute
$E_{0,t}$ (truck)	$a^0_{0,t}$ (velocity of the truck)
	$a^1_{0,t}$ (type of the truck)
	$a^2_{0,t}$ (info about the shipment)
$E_{1,t}$ (truck)	$a^0_{1,t}$ (velocity of the truck)
	$a^1_{1,t}$ (type of the truck)
	$a^2_{1,t}$ (info about the shipment)
$E_{2,t}$ (create)	$a^0_{2,t}$ (generator type)
$E_{3,t}$ (workstation)	$a^0_{3,t}$ (delay type)

The collection of attributes are:

$$\tilde{A}_t = \left\{a^0_{0,t}, a^1_{0,t}, a^2_{0,t}, a^0_{1,t}, a^1_{1,t}, a^2_{1,t}, a^0_{2,t}, a^0_{3,t}\right\}$$

Suppose that models can reach only the attributes of the moving entities from model M. This can be described by specifying the access functions as:

$$F(0,0) = F(0,1) = F(0,2) = 1,$$
$$F(1,0) = F(1,1) = F(1,2) = 1,$$
$$F(1,0) = 0 \text{ and } F(2,0) = 0.$$

Applying these characteristic function F we can specify the

$$\hat{A}_t = \left\{a^0_{0,t}, a^1_{0,t}, a^2_{0,t}, a^0_{1,t}, a^1_{1,t}, a^2_{1,t}\right\}.$$

As we can see $\hat{A}_t \neq \emptyset$ and $\tilde{A}_t \neq \hat{A}_t$ which implies that the simulation package *P* is partly open. The accessible attributes are:

$$a^0_{0,t}, a^1_{0,t}, a^2_{0,t}, a^0_{1,t}, a^1_{1,t} \text{ and } a^2_{1,t}.$$

In order to achieve the accessible attributes we can define the interface functions when $\delta = N \times N$:

$$G(0,0) = a^0_{0,t},\ G(0,1) = a^1_{0,t},\ G(0,2) = a^2_{0,t},$$
$$G(1,0) = a^0_{1,t},\ G(1,1) = a^1_{1,t},\ G(1,2) = a^2_{1,t}.$$

The implementation of the *G* function can be done in the following way:

$$G(k,l) = getEntity(k)\,Attribute(l)$$

For example, for all the truck entities we can specify the following functions:

$$getTruck(k)Velocity = G(k,0) = a^0_{k,t}$$
$$getTruck(k)Type = G(k,1) = a^1_{k,t}$$
$$getTruck(k)InfoShipment = G(k,2) = a^2_{k,t}$$

By creating an entity, a unique identifier (the *k* number) is assigned to it. Using this *k* number and the interface functions we can easily access the attribute values of the entity. Other models that want to use these entities are confronted with the problem to find out these unique identifiers. In a distributed simulation study a distributed simulation architecture can provide a mechanism (e.g. publish / subscribe) for informing the other models about the updating some of the relevant information (Kuhl et al., 1999).

In order to make the COTS simulation packages suitable for distributed simulation the vendors should make them as open as possible by enlarging the *collection of all accessible attributes* set. Furthermore, for all these accessible attributes a large variety of interface functions should be provided.

3 INTEROPERABILITY BETWEEN COTS SIMULATION MODELS

The simulation models designed and developed in COTS simulation packages can not achieve a *direct* interoperation with other COTS simulation models. Therefore for those packages that are open or partly open a so-called *wrapper* must be developed that takes care of the interoperability with the other model. The wrapper provides a two way interaction:

1. Interactions towards the COTS simulation models using the interface functions for accessing the internal data (e.g. entity instances and their attribute values, simulation time, next event in the event calendar, etc.)
2. Interactions with other models or wrappers of the models through a distributed simulation architecture using the interoperability functions.

Most of the COTS simulation packages have possibilities for the modeler to define such a kind of wrapper around the simulation model. Figure 1 depicts an architecture where two models are connected through their wrappers to a distributed simulation architecture. The interoperability itself between the simulation models is achieved by applying a distributed simulation architecture, like HLA (Defense Modeling and Simulation Office 1996) or the FAMAS Backbone (FAMAS MV2 Backbone Project 2001). The distributed simulation architecture provides the interoperability functions for the simulation wrappers.

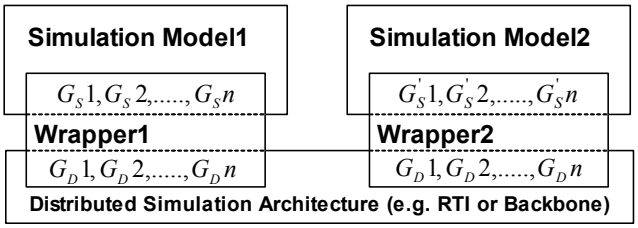

Figure 1: Connection of Two COTS Simulation Models

We can formalize the wrapper W as a set of interface function $\{G_S i\}_{i\in[0,n]}$ offered by the COTS simulation package and a collection of interoperability functions $\{G_D i\}_{i\in[0,n]}$ offered by the distributed simulation architecture. Since $\{G_S i\}_{i\in[0,n]}$ is provided by the COTS simulation package, therefore is simulation package dependent. In the same way $\{G_D i\}_{i\in[0,n]}$ interoperability functions are dependent on the distributed simulation architecture.

Definition 8. An *interoperability function* is defined as $G_D : \delta_D \rightarrow \delta$, where δ_D defines the set of references for the distributed simulation architecture and δ defines the set of references for the accessible elements in the COTS simulation package.

In order to present the concept of the interoperability function let us take a very simple example, when a model transfers an entity instance to another model (e.g. a truck instance is transferred from one model to another one, as shown in Figure 2).

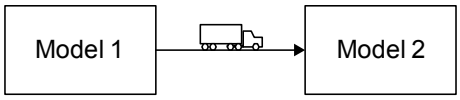

Figure 2: Model 1 Transfers a Truck Entity to Model 2

The first model registers the truck entity for other models through the distributed simulation architecture. The other model can subscribe for the entity object if it is interested. After the entity instance has achieved a certain point (when must be transferred to the second model) the first model publishes the entity to the second one, which means that basically sends the entity to the second model. This phase is called updating. Then the second model tries to discover the entity instance (tries to pick up the sent message).

Using the attribute values of the previous example we can define $\delta_D = N \times N \times N$, where the first dimension defines the action (e.g. register, subscribe, publish, etc.), the second one the entity instance and the third one the attribute. Some of the implementation of the G_D function can be done as follows:

$$G_D(0,k,0) = registerTruck(k)Velocity$$
$$G_D(0,k,1) = registerTruck(k)Type$$
$$G_D(1,k,0) = subscribeTruck(k)Velocity$$
$$G_D(2,k,0) = publishTruck(k)Velocity$$
$$G_D(3,k,0) = updateTruck(k)InfoShipment$$

4 INCONSISTENCY PROBLEMS DURING THE INTEROPERABILITY PROCESS

As we stated before, when two simulation models interact they might need to transfer the simulation entity from one model to the other one. If an entity instance is created in a model and is transferred to another one, the receiver model must support the instantiation of the type of the transferred entity. Figure 2 depicts two models, the first model (sender) generates truck entities and transfers them to the second model (receiver). Let us represent the abstract truck entities as $\widetilde{E}_{Truck}$. At simulation time t the sender creates a truck entity instance $E_i \in \widetilde{E}_{Truck}$ that is transferred to the second model. Due to the fact that the truck entity is transferred to the second model, both models should provide the possibility to instantiate the $\widetilde{E}_{Truck}$ abstract entity. Unfortunately, this solution is not supported in most of the cases.

If the simulation packages work with similar set of entities, that means that both the sender and receiver can instantiate the same type of entities (e.g. a truck entity). However, if the set of entities are different the instantiation is very difficult if not impossible. In some of the cases this problem can be solved using syntactical analyzers by checking the definitions of the entities. For example, in the sender model the abstract entity of the transferred entity is a truck entity $\widetilde{E}_{Truck}$. The receiver model might miss the truck entity, but might contain an abstract lorry entity $\widetilde{E}_{Lorry}$. The truck and lorry entities are basically the same (described by the same attributes), but they are defined by a different name.

The aim of the syntactical analyzers is to find the syntactical errors and to discover the possible matching between different types of entities (e.g. $\widetilde{E}_{Truck}$ and $\widetilde{E}_{Lorry}$). When the receiver model can not instantiate any transportation type entity (e.g. $\widetilde{E}_{Truck}$ or $\widetilde{E}_{Lorry}$) then the transfer of this kind of entities can not occur. Basically in this situation the receiver is not allowed to subscribe for any transportation entity transfer.

Further, in some of the cases the abstract entity names are the same but they define different attribute sets (second row in Table 2). For example, both the first and second models can instantiate an $\widetilde{E}_{truck}$ abstract entity, but in the second model the $\widetilde{E}_{truck}$ does not include the definition of the information about its shipment. Semantic analyzers can be applied in order to tackle this problem.

Table 2: Entity and Attributes Relations

Set of Entity	same	different	missing
Set of Attributes	same	different	missing
Type of Attributes	same	different	-

Moreover, if the set of entities and the set of attributes are the same we might still be confronted with the problem that the type of the attributes differs (third row in Table 2). We would like to give an approach for solving the situation when the attribute types are different.

The architecture of Figure 3 depicts two simulation models that are connected to each other through a distributed simulation architecture. The first simulation model transfers an entity to the second simulation model. The second model is able to instantiate the same abstract entity but some of the attribute types of the instantiated transferred entity differ. Therefore the wrapper of the second model creates a temporal entity instance table and maps the original attributes of the transferred entity with those that the second model supports. Let us take a simple example in order to show this mechanism. The example is depicted in Figure 4, namely the transfer of a truck entity from the first model (*M1*) to the second model (*M2*) and then back to the first model (*M1*). Suppose that model $M1 \triangleright P1$ (simulation model *M1* is designed and developed in package *P1*), $M2 \triangleright P2$. Simulation package *P1* supports all the attribute types, such as string, real, integer, etc. (e.g. eM-Plant), but *P2* supports only the type real (e.g. Arena). Both the *P1* and *P2* package can instantiate the truck entity. The set of the attributes of the truck entity are the same (velocity, type, shipment), and some of them differ only in their types.

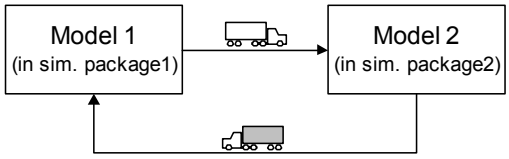

Figure 4: Circular Entity Transfer

Suppose that the truck entity is created in the first simulation model with the attribute values Velocity =80, Type = 'carrier' and Info Shipment = empty (Figure 5). At a certain point in time the entity leaves the first model and is transferred to the second simulation model. The second model is limited compared to the first one in the sense that it can only interoperate with numbers. In spite of this limitation we are able to represent also the type and info shipment using numbers. The wrapper of the receiver model creates a temporal entity instance table, where it maps the original attribute values with numbers (e.g. for the second model the "carrier" as an attribute value for the truck entity means number 1 and if the truck is "empty" for the info shipment attribute means 0).

After the truck entity is instantiated in the second model, it will go through some modifications, in the sense that the shipment attribute of the truck is changed. The truck is not empty anymore, it will carry a container. This action updates the information about the truck shipment in the entity instance table. For the second model if the truck is filled with a container then the value of shipment is updated from 0 to 12. After some time the filled truck is transferred back to the first simulation model. In this case the wrapper uses again the temporal entity instance table to convert the numbers to their original types. Figure 5 gives a clear picture of this mechanism.

For some of the COTS simulation packages the $\{G_S i\}_{i\in[0,n]}$ interface functions are restricted to return only a certain type (e.g. integer) but on the other hand inside the simulation model more then a single type can be handled. For example Arena (Kelton et al., 2001) can easily expose integer numbers (through the EVENT block) but inside the package we can use other types as well. When connecting to other packages, and using temporal entity instance tables we can overcome this restriction.

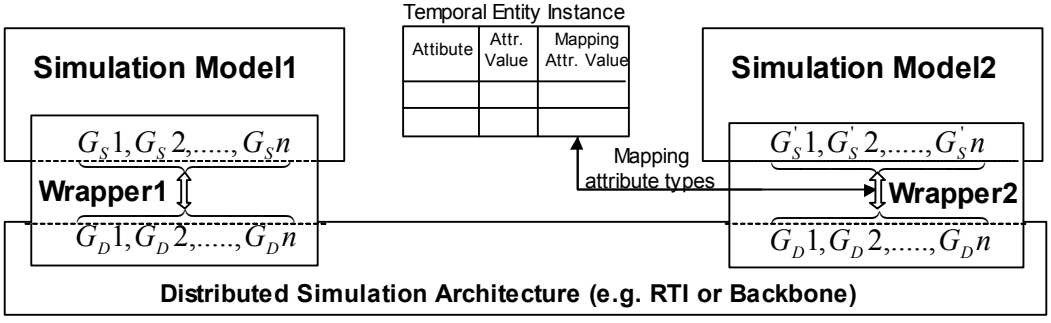

Figure 3: Architecture for Attribute Type Inconsistency.

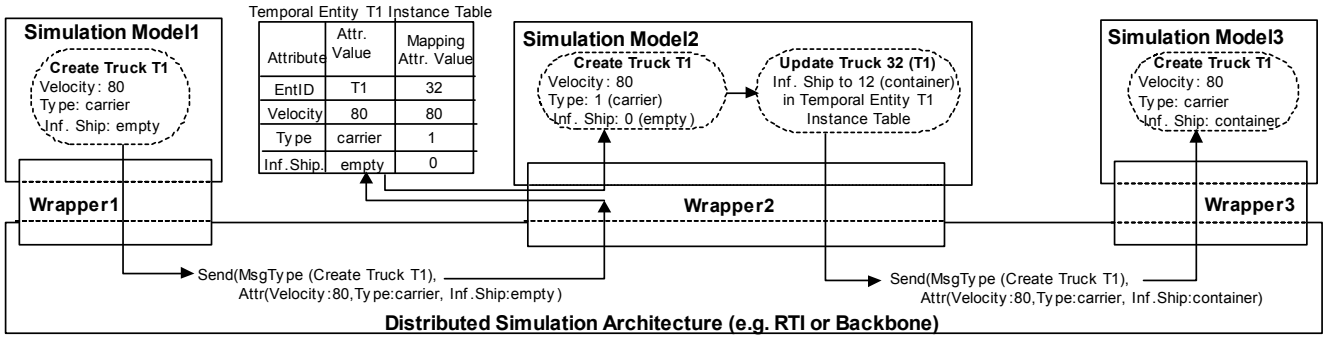

Figure 5: Entity Transfer with Inconsistent Attribute Types

5 IMPLEMENTATION

In this section we would like to present a simple implementation of the previously discussed concepts. For this reason we describe how to connect two simple simulation models, which are developed in different simulation packages, namely in Arena (Kelton et al., 2001) and in eM-Plant (eM-Plant official website 2003). These two COTS simulation packages are based on different concepts: while Arena is a process oriented simulation package, the eM-Plant is an object oriented simulation package. Both packages are partly open and they do not offer a direct access to the internal data, but they offer possibilities to interface them.

5.1 The Simulation Models

The simulation models that we use as examples are very simple as we concentrate fully on the interoperability aspects.

Figure 6 depicts the Arena model, which creates an instance of a truck entity. This movable entity instance goes through two stations, then arrives to a point when is transferred to the eM-Plant model.

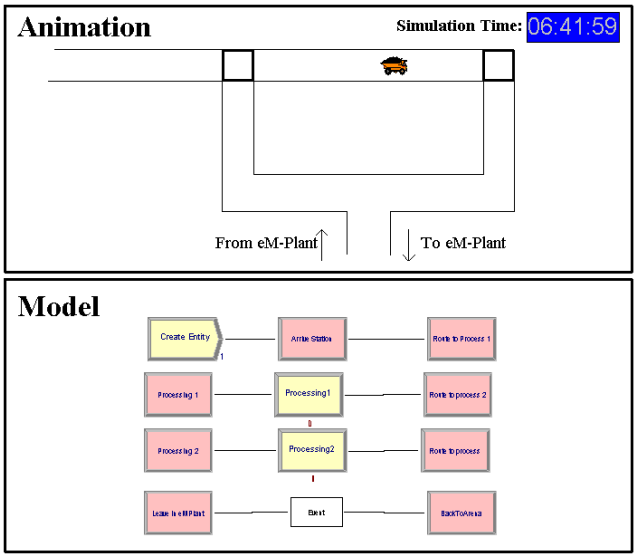

Figure 6: The Arena Model

In eM-Plant model (Figure 7) the truck entity is then created, this follows a certain route, picks up a container and finally is transferred back to the Arena model.

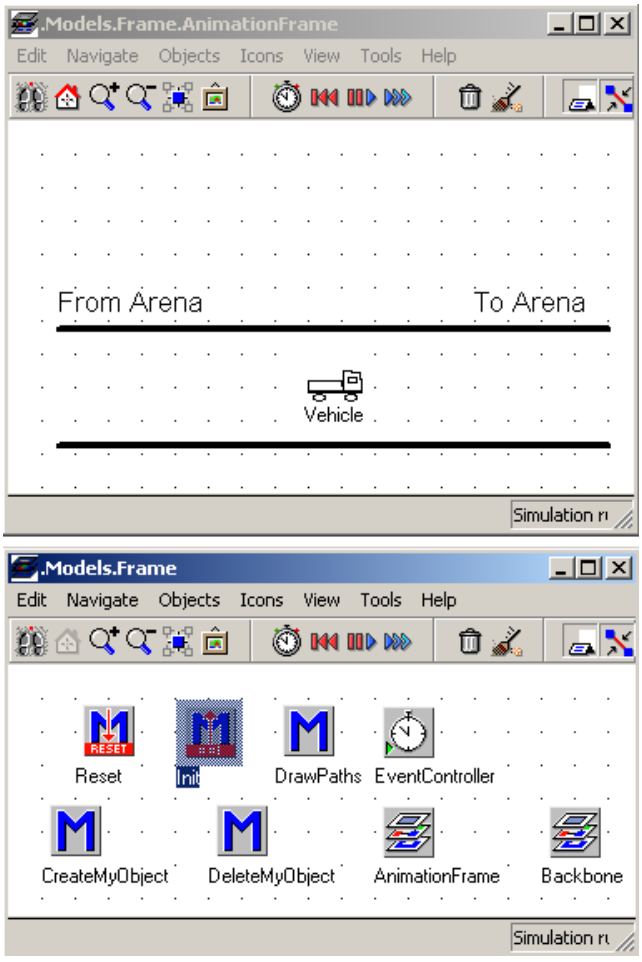

Figure 7: The eM-Plant Model

5.2 The Distributed Simulation Architecture

In order to achieve the interoperability between the simulation models we need a distributed simulation architecture that provides the run control of the overall system, the data

sharing, the time synchronization, etc. The currently existing standard for the distributed simulation architecture is the HLA (Defense Modeling and Simulation Office 1996). The HLA uses an *Object Model Template (OMT),* that provides a common method for recording information and establishes the format of key models. Furthermore the HLA provides an *HLA Interface Specification,* which defines Run-Time Infrastructure (RTI) services and identifies "callback" functions that each simulation model must provide. All the facilities needed for the interoperation are included in RTI.

Another possibility is to use a lightweight interoperability framework such as the FAMAS architecture (FAMAS, 2001). The FAMAS Simulation Backbone Architecture is represented by technical and functional components. Whereas the functional components represent the simulation models themselves, the technical components provide common tasks used by the functional components. There are five well-defined subsystems, namely the Run Control Subsystem, the Backbone Time Manager Subsystem, the Logging Subsystem and the Visualization Subsystem (Boer et al. 2002).

Due to its simplicity and the demand of minimal functionalities for this example we have chosen the FAMAS Backbone as a distributed simulation architecture.

5.3 The Wrappers

The Arena and eM-Plant are partly open COTS simulation packages. There is no direct access to the internal data. Theoretically the wrapper consists of two sets of functions: a set of interface functions $\{G_S i\}_{i\in[0,n]}$ offered by the COTS simulation package and a collection of interoperability functions $\{G_D i\}_{i\in[0,n]}$ offered by the distributed simulation architecture (Figure 1).

Due to the fact that Arena and eM-Plant both support added functionality through DLLs (Dynamic Link Library) and their interface functions can be implemented in a DLL, the wrapper is regarded and implemented as a DLL. Unfortunately, we can not consider a common DLL because Arena and eM-Plant implement the interface functions in a completely different way. However, there are similarities in the DLLs because they use the same interoperation functions for each COTS simulation package (if they use the same distributed simulation architecture).

Compared to the standalone COTS simulation models the distributed COTS simulation models need some additions in their modeling design. The Arena model (Figure 6) can be run as a standalone simulation model, but in order to run it in a distributed way we need an additional modeling part (Figure 8).

The additional part communicates through the wrapper with the distributed simulation architecture (in this case the FAMAS backbone). It is responsible to connect to the Run

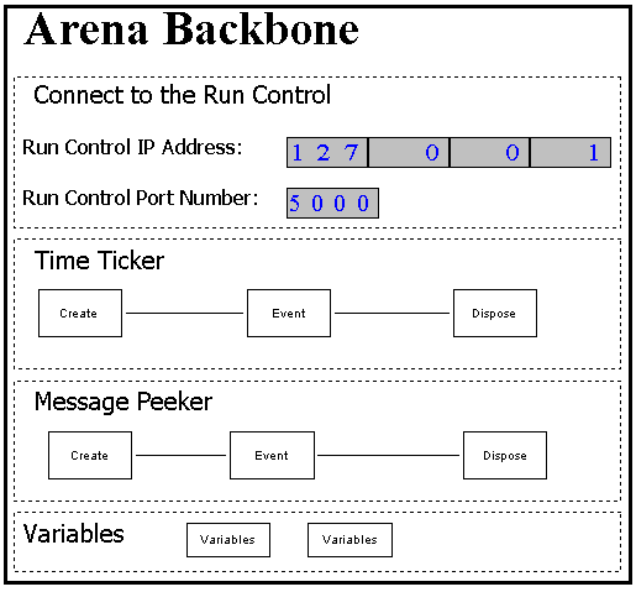

Figure 8: Arena Backbone

Control subsystem, to send and to peek for messages using a certain time resolution. This is necessary for time synchronization and data transfer. Peeking for messages is done because the single-threaded Arena simulation engine crashes when data is 'pushed' into Arena using a separate thread in the DLL. Sending messages is done as follows. When an entity reaches an EVENT block this triggers the `cevent` function in the DLL. The `cevent` is a kind of interface between the model and the outside world. Due to the fact that through `cevent` we can communicate only with integers we need a mapping table for the shared data. A piece of implementation of the interface function `cevent` is depicted here:

```
extern "C" void cdecl cevent (SMINT l, SMINT n)
{
 AFX_MANAGE_STATE(AfxGetStaticModuleState());
 switch (n)
  {
 case 1:     //eventAdvance
     {
      SMREAL currentTime = gettnw();
      theApp.ArenaAdvance(currentTime);
      break;
     }
 case 2:     //gotoEMPlant
     {
      //send create entity to eMPlant
     theApp.pFamasMessageHandler->
     sendFamasMessage(…);
     }
 case 3:     //eventPeek:
     {
     MessageType* pMessage =
         theApp.PeekFamasMessage();
     if (pMessage!=NULL)
```

```
        {
            if ((pMessage->sender ==
                "eMPlantModel") &&
                    (pMessage->type== "NEWTRUCK"))
            {
            … //process the message (create
              //truck)
            }
    }
}
```

The eM-Plant model needs an extension as well in the modeling design if it would like it to participate in distributed simulation. We can run the model (Figure 7) as a standalone simulation model, but in order to make it distributed we need the additional model component depicted in Figure 9.

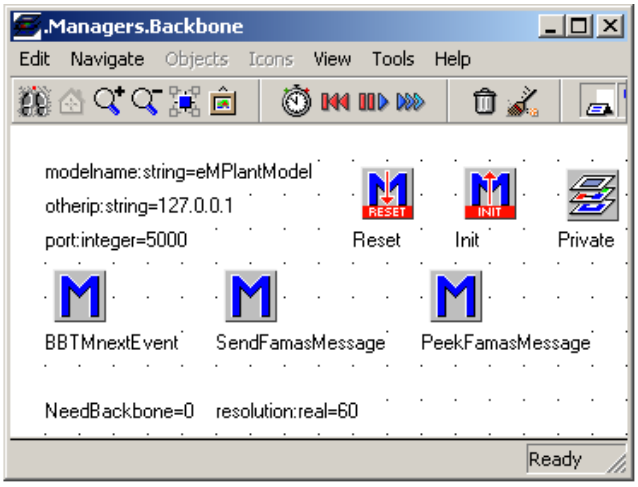

Figure 9: eM-Plant Backbone Module

Connecting eM-Plant models to the backbone is more flexible than connecting Arena models because from eM-Plant environment we can trigger any DLL functions directly (there is no need for something like the EVENT module) at any time. Furthermore, in contrast with Arena eM-Plant can transfer not only integers but also other types (e.g. string). The following DLL function can be directly called from the eM-Plant environment.

```
//sending a message from eM-Plant
extern "C" __declspec(dllexport) void Send-
FamasMessage(UF_Value *ret, UF_Value *arg)
{
…
theApp.pFamasMessageHandler->
    SendFamasMessage(parameters);
theApp.pFamasMessageHandler->checkMessages();
…
}
```

Both the Arena and eM-Plant use the general interoperability functions $\{G_D i\}_{i\in[0,n]}$, such as PeekMessage, SendMessage, checkMessage, nextEvent, etc., which are distributed simulation architecture dependent.

6 DISCUSSION AND CONCLUSIONS

Coupling two very simple COTS simulation models in a distributed way does not make sense because the problem they solve can be designed and developed in a couple of minutes in one COTS simulation package. Distributed simulation plays an important role in the case of complex simulation models, where for example different organizations are involved that on the one hand try to hide their business logic from the other parties, but, on the other hand, their system are dependent on each other. The different organizations are responsible for designing and developing their simulation models and offering the interfaces (for data transfer) for other parties.

Coupling standalone COTS simulation models without any modification is impossible due to the basic design of the model and package limitations. If a standalone simulation model wants to participate in a distributed simulation study slight modifications (dependent on the complexity of the model and the distributed simulation architecture) are needed. Furthermore various consistency checking are needed with other simulation models (e.g. syntactic, semantic, pragmatic analyzes). The modeler is also responsible on developing the wrappers (e.g. as DLLs) for the simulation models. If the interoperability functions offered by the distributed simulation architectures are transparent and flexible that will help a lot for the modeler. In the same way the interface functions offered by the COTS simulation models must be as transparent and flexible as possible.

REFERENCES

Boer, C. A., A. Verbraeck, and H.P.M. Veeke. 2002. Distributed Simulation of Complex Systems: Application in Container Handling. *Proceedings of SISO European Simulation Interoperability Workshop*, Harrow, Middlesex, UK, June 24-27.

Defense Modeling and Simulation Office. 1996. HLA specification.

Kelton, W. D., R. P. Sadowski, and D. A. Sadowski. 2001. *Simulation with Arena*, McGraw-Hill.

Kuhl, F., R. Weatherly, and J. Dahmann 1999. *Creating Computer Simulation Systems. An Introduction to the High Level Architecture*, Prentice Hall.

eM-Plant official website. Technomatics Technology Ltd. <www.emplant.de/simulation.html> [accessed March 28, 2003]

FAMAS MV2 Backbone Project. 2001. Research Program FAMAS Maasvlakte II Project 0.2 - Simulation Back-

bone., Delft, The Netherlands. Available online via <www.famas.tudelft.nl> [accessed March 30, 2003].

Fujimoto, R. M. 2000. *Parallel and Distributed Simulation Systems.* John Wiley & Sons,Inc., New York.

Law, A. M. and W. D. Kelton, 2000. *Simulation Modeling and Analysis*, McGraw-Hill.

Zeigler, B.P., H. Praehofer, and T.G. Kim. 2000. *Theory of Modeling and Simulation.* Academic Press, San Diego.

AUTHOR BIOGRAPHIES

CSABA ATTILA BOER is a Ph.D. student at the Department of Computer Science of the Faculty of Economics at Erasmus University Rotterdam, The Netherlands. He received his M.Sc. degree in Computer Science at the Babes Bolyai University, Cluj Napoca, Romania. Since April 2001 he has been involved in FAMAS MV2 Simulation Backbone project. His research focuses on Multi-Level Distributed Simulation of Complex System. His email address is <acboer@few.eur.nl>.

ALEXANDER VERBRAECK is an Associate Professor in the Systems Engineering Group of the Faculty of Technology, Policy and Management of Delft University of Technology, and part-time research professor in supply chain management at the R.H. Smith School of Business of the University of Maryland. He is a specialist in discrete event simulation both real-time analysis and control of complex transportation systems and for modeling business systems. His current research focus is on the development of generic libraries of distributed object oriented simulation building blocks and the development of Java-based simulation languages for Web-services . His email address is <a.verbraeck@tbm.tudelft.nl>.