Incorparating the Actor Model into SCIVE on an Abstract Semantic Level

Christian Fröhlich[*]
AI and VR Group, Bielefeld University

Marc E. Latoschik[†]
AI and VR Group, Bielefeld University.

Abstract

This paper illustrates the temporal synchronization and process flow facilities of a real-time simulation system which is based on the actor model. The requirements of such systems are compared against the actor model's basic features and structures. The paper describes how a modular architecture benefits from the actor model on the module level and points out how the actor model enhances parallelism and concurrency even down to the entity level. The actor idea is incorporated into a novel simulation core for intelligent virtual environments (SCIVE). SCIVE supports well-known and established Virtual Reality paradigms like the scene graph metaphor, field routing concepts etc. In addition, SCIVE provides an explicit semantic representation of its internals as well as of the specific virtual environments' contents. SCIVE uses a knowledge representation layer (KRL) to tie together the participating modules of a simulation system and reflects this information between the modules and processes. As a consequence, the actor model based temporal relations are lifted to the KRL which in turn is implemented by a real-time tailored semantic net base formalism. The modules' process flow is henceforth described on the KRL. This high-level description is extended down to the level of detailed function calls between the modules. Functions, their parameters, and their return values are reflected on the KRL. This provides an integrative semantic description and interconnection layer uniformly accessible a) for the incorporated technical modules and processes as well as b) for the human designers and developers.

Keywords: Intelligent Virtual Environment, Virtual Reality, Simulation Core, Ubiquitous Computing, Application Framework.

Index Terms: D.2.11 [Software Engineering]: Software Architectures— [I.6.7]: Simulation and Modeling—Simulation Support SystemsEnvironments H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Artificial, Augmented, and Virtual Realities I.2.4 [Artificial Intelligence]: Knowledge Representation Formalisms and Methods—

1 Introduction

The development of sophisticated Virtual Reality applications comprises complex software engineering tasks. Modeling believable immersive Virtual Reality environments nowadays requires a variety of modules, such as graphics, physics or haptics, which contribute to the overall simulation within their own task-specific boundaries. This is particularly true in the area of intelligent virtual environments (IVEs) which, e.g., encompasses work on computer games, ubiquitous computing, or smart graphics. IVEs require diverse Artificial Intelligence (AI) methods. AI provides useful–if not necessary–methods for a variety of aspects in such fields like path planning, application logic, semantic environment descriptions, or the simulation of cognitive processes for artificial agents or NPCs (non-player characters).

[*]e-mail: cfroehli@techfak.uni-bielefeld.de
[†]e-mail:marcl@techfak.uni-bielefeld.de

Since visual perception is very important for generating immersive environments, most VR applications focus on the graphical simulation output. Tools like OpenGL Performer, Open Inventor [12], Open Scene Graph, OpenSG [11] or the VRML successor X3D [8] are based upon a hierarchical scene graph structure, which allows for complex rendering tasks. Extension mechanisms like field routing or rapid prototyping via scipting interfaces allow for new customized node types and interconnected application graphs.

Purpose-built VR application frameworks adopt and extend these mechanisms, by adding in- and output customization, see e.g. Lightning [4], Avango [13], VR Juggler [3] or the CAVELib^{TM}. Also network distribution paradigms are often integrated to enable distributed rendering on a cluster architecture, as can be seen in Avango [13] or DIVE [5].

Lots of VR applications are designed based upon the useful scene graph data representation. The applications are built around the graph structure utilizing it's benefits like field routing or node inheritance. This leads to a close coupling between the application and the utilized framework itself, which inhibits the application's portability. As already mentioned, todays VR applications require the integration of additional simulation modules, like physics or haptics. Such modules are often integrated on a case by case base or they are closely coupled to a holistic architecture often found in game engines like the Doom 3 Engine or the Unreal Engine 3.

Real-time simulation systems benefit from a more modular approach, at least when it comes to the extensibility and portability of the framework itself, as well as the developed applications, as seen in FlowVR [1] or SCIVE [10]. The latter introduces the mechanism of semantic reflection [9], which is inspired by the reflection paradigm in Object Oriented Programing (OOP). The goal is the decoupling of simulation content and the simulation system itself. Semantic reflection enables the mapping of different simulation aspects onto a semantic representation–in this case a semantic network. This mapping covers the whole world state of a simulation, including scene entity represenations, functional definitions, as well as definitions of participating modules and representations of temporal processes. This results in a complete simulation graph. A schematic illustration of the semantic reflection concept is shown in Figure 1.

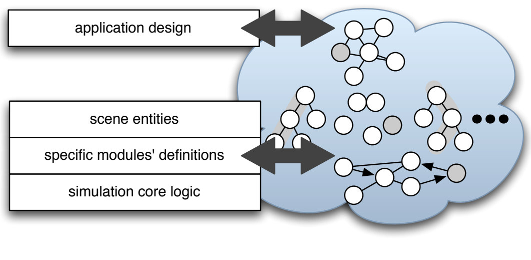

Figure 1: Mapping of data and object representations from various simulation applications' layers to a unified semantic knowledge representation, resulting in a simulation graph (graphic taken from [9]).

Semantic modelling in the context of software development and simulation systems is still in it's early stages, but has to offer some promising approaches to recent issues.

Inside a modular architecture the various simulation modules can be regarded as *actors*. According to the actor model by Hewitt, Bishop and Steiger [7] actors are the universal primitives for concurrent digital computation. The actor model was inspired by prior concepts like the lambda calculus by Church or the programming language *Smalltalk*. It has been implemented in a number of programming languages (actor languages) since, such as *Erlang*, *Io* or *E*, and it is the base for more modern languages like *Stackless Python*, which is used for example in computer games programming, e.g. EVE Online. Also games using the Unreal Engine 3 use actors as objects in virtual worlds, even though there is no detailed information available on this aspect. These previous language implementations of the actor model are not very common in todays Virtual Reality programming. This is because no explicit binding between these proprietary languages and the Virtual Reality domain has been established so far. Therefore we propose a different approach. We describe the actor model's principles on a higher level semantic structure, to be more independent from a specific language implementation, which does not have the required capabilities for efficient Virtual Reality development.

The following chapter will give more insights into the actor model and it's relevance for modular simulation frameworks.

2 The Actor Model and its Parallels to Modular Simulation Systems

In a first step we reflect the actor model's structure on the level of the participating simulation modules and their temporal synchronization. It is possible to implement the model on the more detailed level of the different world entities as well. To implement this, every reflected entity, such as data object or functional definitions, has to be modelled as an actor. This is an approach we will explore in the near future.

As mentioned above, actors can be seen as the primitives of digital computation. An actor can reveive messages and respond to them by making local decisions, creating new actors, send new messages or determine how to react to the next message it receives.

Looking at modular simulation systems a similar strucutre can be observed. Every simulation module completes it's own task inside it's local boundaries. A physics engine for example computes physical based object behaviour or general forces like gravity or joints between different objects, while a graphics component only cares about graphical rendering of a running simulation. So each actor/module runs it's own simulation without knowledge of what the others do. Nontheless every module receives messages and responds to them in a proper way.

To compare a simulation module to the actor-primitive we have to take a look at the general features an actor must have according to the actor model. A module has to be able to:

1. Receive messages from other modules
2. Make local decisions or computations
3. Send messages to other modules
4. Spawn new modules

The following example shows that these features suffice for a task-specific dedicated simulation module. The goal is the simulation of a physically animated scene with a graphical and accoustic output and user-interaction using the SCIVE framework.

Figure 2 shows the three independent modules and a central controlling instance, called *SCIVE*, which includes the knowledge representation layer used for the semantic reflection, as well as the data exchange mechanism between the different modules [6]. Every action that is happening in this example can be reduced to the operations used in the actor model. The modules send messages to each other in response to received messages or internal events. They also do local computations or execute local actions independently from each other, and the controlling instance is able to spawn new required modules at runtime (see the *audio* module).

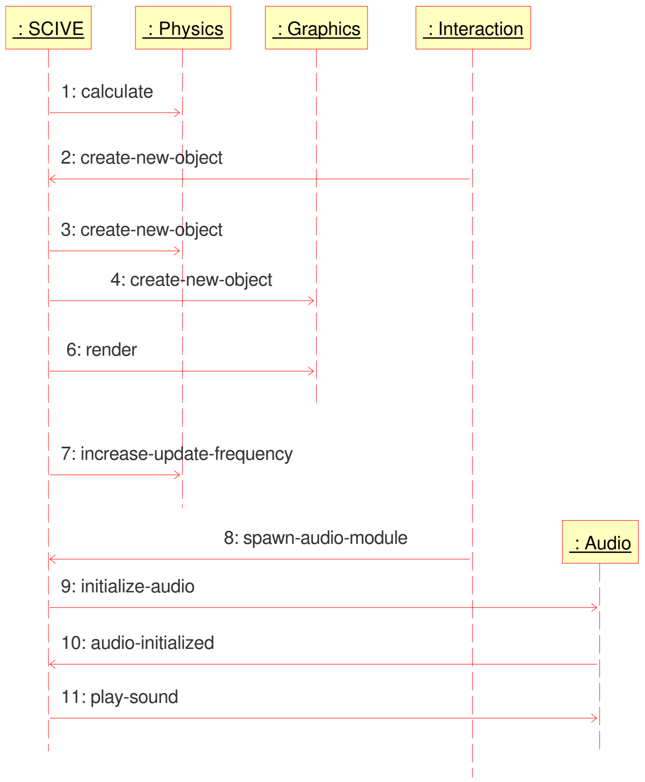

Figure 2: Illustration of message passing between different modules in SCIVE.

The actor model itself is inherently concurrent. Messages are being sent through channels between two actors, whereas most messages include a continuation, that allows a reply to be sent. Sending messages happens asynchronous, meaning there is no need for the sending actor to wait for the receiving one to handle the message, in other words, it always returns immediately. As soon as the message is sent, the sending actor goes on with it's normal computation. Incoming messages are stored inside an actor's buffer, from where they are handled with respect to absolute fairness, that is, no message has to wait indefinitely to be processed. Concurrency is also an important aspect for real-time simulation systems, since it represents a significant performance issue. To guarantee a consistent world state, there must be some order in sending and receiving messages, as well as in executing concurrent calculations, especially when it comes to reading and writing shared data objects, e.g. an object's position, which is influenced by an user's interaction, as well as the calculation of physical laws done by the physics simulation engine. The next chapter will elucidate how this issue can be handled on a semantic level using SCIVE's semantic network as central knowledge representation layer in conjunction with the concept of semantic reflection.

3 Modelling Functions and Temporal Processes on a Semantic Level

Temporal relations have been explored in depth by Allen [2]. He defined 13 temporal interval primitives, that can easily be mapped onto a semantic network. To be able to do this, first of all the actions taking place inside the simulation have to be reflected on the network. The actions can consist of message passing between the actors/modules or for example function calls executed by a specific actor. Figure 3 shows some C++ function call definitions modelled on the functional extendable semantic network used in SCIVE.

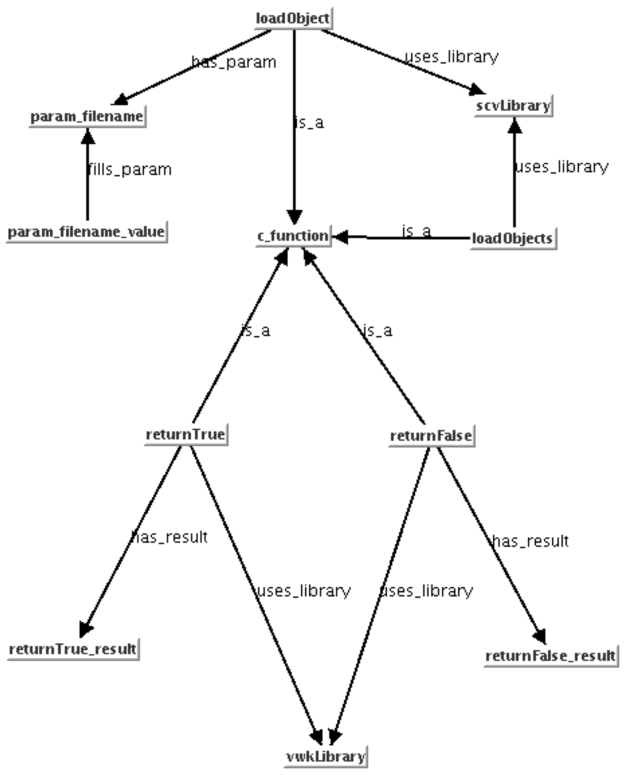

Figure 3: Semantic reflection of a C++ function call on a semantic network

The function calls are modelled by means of inheritance, derived from a basic *c_function* concept through the *is_a* relation. It is possible to model function calls without parameters, like the *returnTrue* or *returnFalse* functions, or with parameters, such as the *loadObject* definition. A function can also have more than one parameter because it is possible to sort the parameter nodes by assigning a number to a specific slot inside the node, so the function call knows in which order the parameters have to be used. The function itself is called at runtime by small specialized traverser programs, which traverse the network in search of specific node or relation types. When a function traverser has found it's desired target, it constructs the function call by traversing the relations connected to the function node to get it's parameters and return value. The constructed call is executed with the help of the *libffi* (foreign function interface), which is included with the gcc-distribution. The *uses_library* relation indicates which shared object the traverser has to open to execute the call.

To model temporal processes by means of semantic reflection we use Allen's temporal interval primitives as relations between certain actions. Let's assume we have two actions X and Y. Allen defined his 13 primitives as follows:

1. before: X takes place before Y

2. meets: X meets Y (Y starts with the end of X)

3. overlaps: X overlaps Y (Y starts while X is still in execution)

4. starts with: X starts with Y (X and Y start at the same time, while X finishes first)

5. during: X during Y (X takes place, while Y is in execution)

6. finishes with: X finishes with Y (X and Y finish at the same time)

7. equal: X is equal to Y (X and Y start and end at the same time)

Items 1 to 6 are considered to be bidirectional, which leads to 13 primitves instead of the listed seven. These 13 relations can be very useful when it comes to modelling temporal sequences on the semantic level. Figure 4 shows an example of how temporal processes are modelled in SCIVE.

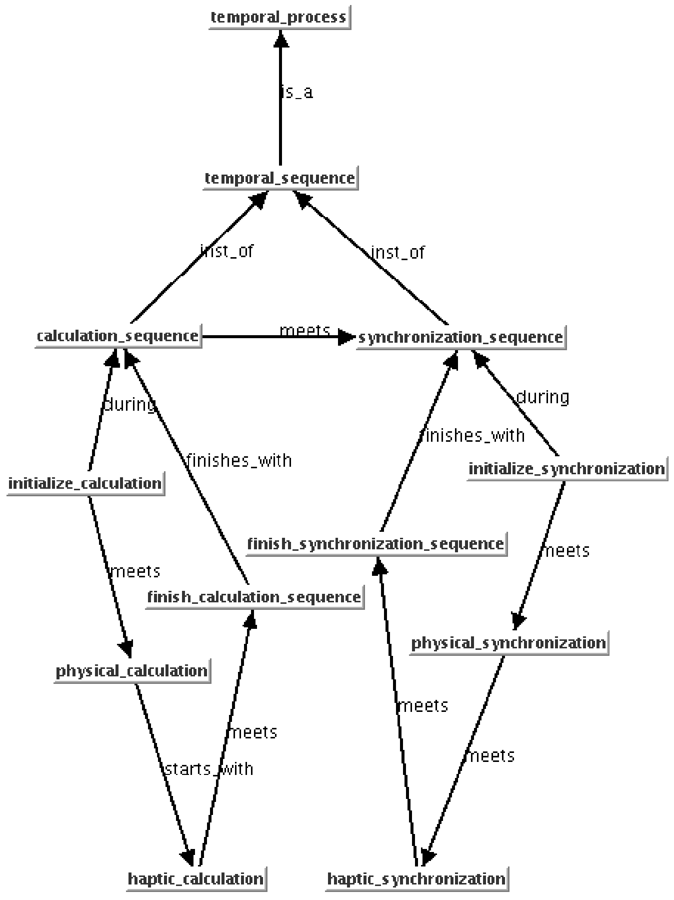

Figure 4: Temporal sequences of actions taking place in SCIVE using semantic reflection

Two sequences of the type *temporal_sequence*, which is derived from the general type *temporal_process*, are instanciated. These sequences initialize the calculation of the simulation modules on the one hand, and on the other hand the synchronization of their results. After the initialization of the *calculation_sequence* the various modules start their concurrent computation loops. When finished, the *finish_calculation_sequence* function is called and the sequence gets finalized. The two sequences are interconnected via a *meets* relation, meaning the *synchronization_sequence* starts with the end of the calculations. In this case the synchronization of data happens not concurrently, because the nodes are connected through the *meets* relation, which requires the source action to be finished, before the target function can be called. The relations between the nodes determine how the sequence is executed by the network traversers. The sequences can be changed at runtime by changing the network's topology via a scripting interface in *Scheme* (the interface is generated via *swig*, so other languages are also possible), or by the actions themselves.

4 Conclusion

Incorporating the actor model into the development of real-time interactive Virtual Reality simulation systems is a powerful and promising approach. Mapping the model onto a higher level semantic structure offers several advantages to prior proprietary programming language implementations. The semantic mapping strongly reduces dependencies to a specific implementation language. Process flow and temporal relations used to be tightly bound to the specific simulation engines. For example, the logic behind data flow processes used in field graph approaches is usually not available to application designers. Changes to such internal processes was—if possible at all—only available on a low-level base and required a deep understanding of the simulation engines' internals. In contrast, a semantic modelling as described here provides this low-level knowledge even to high-level designers.

An actor's interface is simple–viewed from the technical point of view. Every actor needs a message passing interface to communicate with other actors. This interface contains the complex event processing logic, which is similar to the event processing in Service Orientated Architectures (SOA). The handling of this complex event logic is subject to the modelling on the semantic level, using SCIVE's KRL.

The use of Allen's temporal interval relations has proven to be sufficient and reasonable. The 13 primitves map naturaly onto a semantic network, using the actions as nodes and the temporal interval definitions as relations between them. They provide an adequate and on the other hand simple description metaphor for all actions required in typical simulations. On the other hand, the decomposition of semantic descriptions on the KRL and specialized traversers for their interpretation guarantees portability and extensibility, e.g., new temporal primitives and their traversers can be added later on if needed.

Outlook As a next step we will explore the implementation of the actor model as the base for world entity definitions and interactions between entities. This means, every entity must have the capability to send, receive and process messages. On this fine-grained level, the actor model will simplify the communication between different entities. Futhermore this will enable entities to perform concurrent actions and enhance the secure concurrency of the overall simulation. The idea of a semantic model as an engineering paradigm is currently explored in several directions, temporal relations and process flow being only one aspect where it proved to be very promising. In general, our goal is to reflect any simulation aspect on a knowledge level to provide a new paradigm in the development of complex intelligent interactive systems.

References

[1] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. Flowvr: a middleware for large scale virtual reality applications. In *Proceedings of Euro-par 2004*, Pisa, Italia, August 2004.

[2] J. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[3] A. D. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: A Virtual Platform for Virtual Reality Application Development virtual platform for virtual reality application development. In *IEEE Virtual Reality 2001 conference proceedings*, pages 89–96, Yokohama, Japan, 2001. IEEE Press.

[4] R. Blach, J. Landauer, A. Rsch, and A. Simon. A Highly Flexible Virtual Reality System. In *Future Generation Computer Systems Special Issue on Virtual Environments*. Elsevier Amsterdam, 1998.

[5] O. Hagsand. Interactive MultiUser VEs in the DIVE system. *IEEE Multimedia Magazine*, 3(1), 1996.

[6] G. Heumer, M. Schilling, and M. E. Latoschik. Automatic data exchange and synchronization for knowledge-based intelligent virtual environments. In *Proceedings of the IEEE VR2005*, pages 43–50, Bonn, Germany, 2005.

[7] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[8] ISO/IEC, JTC 1/SC 24. X3d abstract. Technical Report 19775-1:2004, ISO/IEC, 2004.

[9] M. E. Latoschik and C. Fröhlich. Towards intelligent VR: Multilayered semantic reflection for Intelligent Virtual Environments. In *Proceedings of the Graphics and Applications GRAPP 2007*, pages 249–259, Barcelona, Spain, 2007.

[10] M. E. Latoschik, C. Fröhlich, and A. Wendler. Scene Synchronization in Close Coupled World Representations using SCIVE. *The International Journal of Virtual Reality*, 5(3):47–52, 2006.

[11] D. Reiners, G. Voß, and J. Behr. OpenSG: Basic Concepts. www.opensg.org/OpenSGPLUS/symposium/-Papers2002/Reiners_Basics.pdf, february 2002.

[12] P. S. Strauss and R. Carey. An object-oriented 3D graphics toolkit. In *Computer Graphics*, volume 26 of *SIGGRAPH Proceedings*, pages 341–349, 1992.

[13] H. Tramberend. A distributed virtual reality framework. In *IEEE Virtual Reality Conference*, pages 14–21, 1999.