Using Semantic Traversers to create persistent knowledge-based Virtual Reality applications

Peuser, Nils
University of Bielefeld, Faculty of Technology, AI and VR Lab
P.O. Box 10 01 31, 33501 Bielefeld
Tel.: +49 (0) 521 / 5602083
E-Mail: nils@peusers.org

Latoschik, Marc Erich
FHTW Berlin, Department 4, International Media Informatics
Treskowallee 8, 10318 Berlin
Tel.: +49 (0) 171 / 7863265
E-Mail: marc.latoschik@fhtw-berlin.de

Abstract: This article introduces the concept of Semantic Traversers (STs) and exemplarily illustrates its utilization inside a Virtual Reality (VR) platform for multimodal construction. The development of reusable and parameterizable routines which are based on the concept of Semantic Reflection is described. These routines work on the Knowledge Representation Layer (KRL) of a simulation framework to realize complex application logic and data flow concepts like field routing. The KRL is implemented by a functionally extended semantic network. The ST development in C++ preserves real-time capabilities while the abstract description of data structures and application logic realizes a persistent and platform-independent representation of programs. The advantages of traverser representation through Semantic Reflection are elaborated. Finally, an editing tool is presented that enables developers to visualize and modify the semantic networks which are used to describe the application.

Keywords: Intelligent Virtual Environment, Semantic Network, Semantic Traverser, Knowledge Based Simulation, Data Flow Processes

1 Introduction

Today's simulation frameworks for Virtual and Augmented Reality (VR/AR) require complex data models. As demands for such applications are constantly increasing, additional features are incorporated to intensify immersion or to create believable interactive worlds. Such features include perceivable simulation aspects like sound systems, physical simulations, speech processing and multimodal input as well as technical matters like distributed rendering. On top of that, application logic and a multitude of Artificial Intelligence (AI) methods have been

integrated into simulation frameworks by means of field routing, decision trees, planning and reasoning mechanisms. Each of these features requires a specially tailored model to maintain its data effectively. Simulation frameworks can be either monolithic, offering a fixed set of features, or modular, ensuring extensibility. In both approaches the data models of different modules must be combined at some point.

Traditional visualization toolkits, for example OpenSG [ReiVoBe02] and OpenGL Performer [RohHe94], use scene graphs as a central data model. This was adopted by VR-specific development tools (FlowVR [AllGoLe04], AVANGO [Tra99]) and enhanced by powerful data flow features like field routing, messaging and event systems working within or on the scene graph. While this is sufficient for applications centered around visualization and unimodal input through pointing devices or gestures, various ideas have been presented which provide alternative advanced methods for application layout and modeling [LatFrWe06] [CavPa00] [KaChMo06] of Intelligent Virtual Environments (IVEs) [AylLu00].

Our latest research on this topic introduced the concept of Semantic Reflection [LatFr07], combining the concept of semantic representation and the traditional principle of reflection known from object oriented programming (OOP). Application logic, knowledge bases or scene graphs are combined in a unified representation based on a semantic network (see section 2) without forcing different modules of a simulation framework to give up their own data models [LatFrWe06]. This semantic network is used as a Knowledge Representation Layer (KRL) and enables developers to create reusable and parameterizable representations of applications and data. In addition, the network is independent of programming languages and abstractly describes an application that is consistent across different computer systems. The essential elements of the application are made persistent for long-time usage and to overcome varying system requirements.

The work presented here introduces an additional abstraction concept called Semantic Traversers (STs, see section 3) that is targeted at a unified description of algorithmic processes working on the KRL. Traversers operate within the semantic context that is given by the KRL and have a semantic representation in the KRL themselves. This enables simulation modules as well as STs to consider actions of other traversers in their decisions, based on the reflected properties. Furthermore, software developers using the semantic description of components are able to debug and optimize applications more easily.

In the following sections we explain how such a unified representation is achieved, without impeding the use of traditional programming approaches at the same time. The utilization of STs is illustrated inside a multimodal virtual construction environment called *Virtuelle Werkstatt* [BiJuLa02]. We will also present a tool that allows user-centered interaction with the given high-level abstract data model.

2 Using semantic networks as a KRL

After elaborating a suitable database system to incorporate all required features from VR tools and AI systems [LatSc03] [HeuScLa05], the semantic network [Brach79] was chosen as a

general data exchange facility. A semantic network's data structures are nodes and relations. The former usually denote concepts representing abstract knowledge or objects which are instances. Relations are directed and labeled, connecting the nodes to create their semantic context and eventually the knowledge base. An example taken from a virtual construction scenario is shown in Figure 1. To cope with the challenges of current and future VR/AR systems the semantic network formalism was extended. The main aspects of this extension are a frame-like [Minsky74] node structure and the consequent use of a domain concept. The latter is useful for maintaining nodes from many different simulation modules, data- and knowledge bases. Both of these extensions are outlined below. A third extension lifts relations to first order objects by allowing them to be connected by other relations.

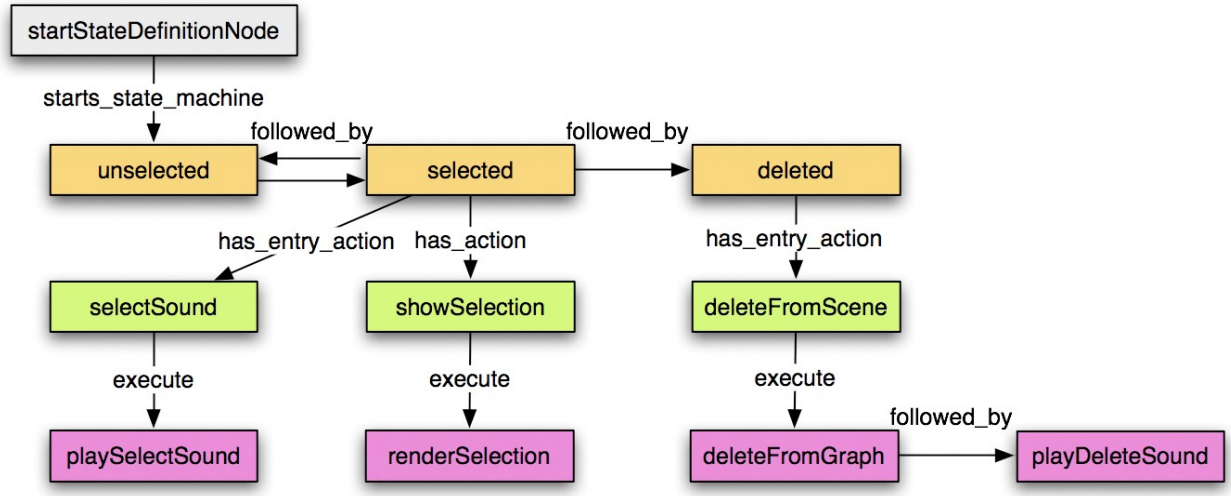

Figure 1: A simple semantic network excerpt depicting a finite state machine for a single part used inside a Virtual Reality construction scenario. The part can be selected and then deleted or unselected. Appropriate actions are connected to the respective nodes. Conditions and hierarchy nodes are hidden. Nodes from different subdomains are shaded differently.

Based on the AI concept of frames, nodes can have attributes or *slots* which represent additional properties not modeled in the semantic network. This frame-like structure enables nodes to contain application- and programming-specific data which cannot be described by a semantic network efficiently for use in simulations. This includes text strings, numbers, 2D images or point lists from complex 3D models which do not contain semantic annotations. These slots are represented by a name and a corresponding value, which can be of any type.

Another benefit of graph based models is the intuitive representation that is easily understandable by humans. To preserve this advantage, while working with complex semantic networks containing hundreds or thousands of nodes, a grouping and selection mechanism must be integrated. The notion of subdomains was introduced into the semantic net by means of Semantic Reflection itself (see Figure 2). Instead of a grouping mechanism that is hardcoded into the implementation of our semantic net, the membership of nodes to different subdomains is indicated by one or more "belongs_to" relations which link them to nodes representing the corresponding subdomains. Subdomains provide a commonly found concept used in modern

database systems and software architectures called a *view* [ElNa00]. Views in the KRL allow designers to concentrate on their task by masking out unnecessary portions of the semantic net. They are also used to group data structures from different simulation modules logically and to make complex network structures readable and comprehensible for humans [LatSc03] [HeuScLa05] [LatFr07].
Having assembled an adequate representation, the next step was to create reusable routines working on ontologies and knowledge bases described by the semantic network. Such routines are called Semantic Traversers and their development and use is explained in more detail in the following section.

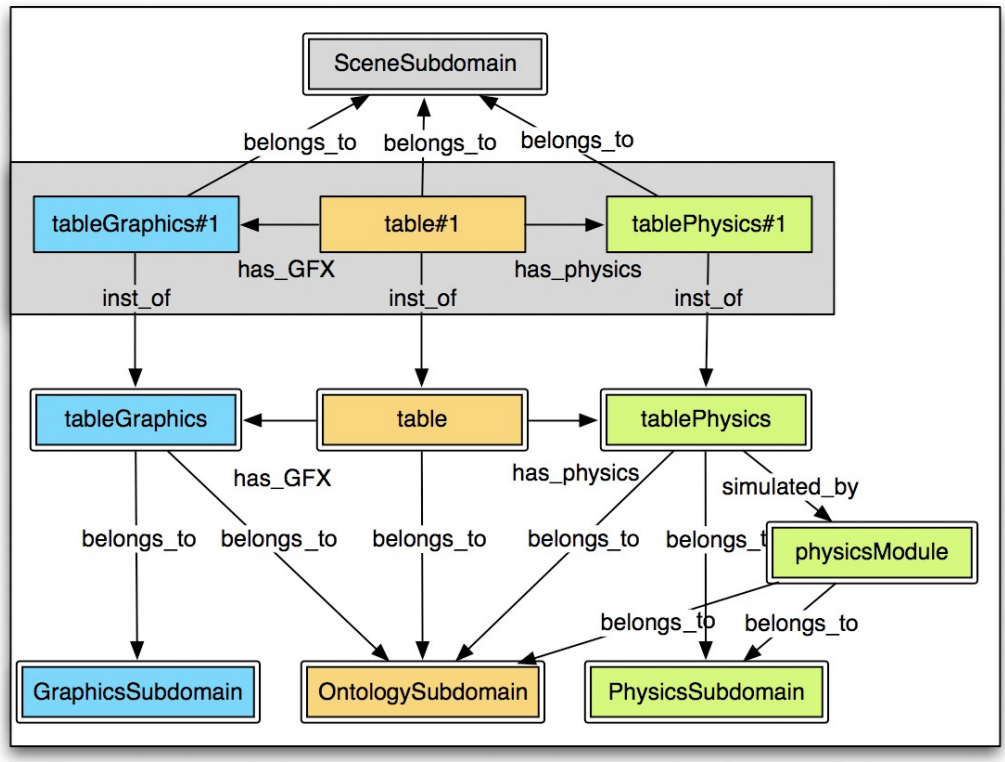

Figure 2: Subdomains of a semantic network in a VR scenario. This concept allows the arrangement of nodes into logical groups. The membership is realized through the „belongs_to“ relation. The colors of nodes indicate the subdomains they are in. At the top, scene knowledge is shown, containing information on a specific table inside the scenario. Abstract knowledge about tables in general is located in the lower part.

3 Semantic Traversers

3.1 General properties of Semantic Traversers

The central routines working on the semantic network are Semantic Traversers. As the name “traverser” suggests, these programs work their way along nodes and relations while reading and writing attributes and assembling results as partial semantic networks. They can also create new

relations and nodes inside the semantic network they are working on. Thereby STs are able to expose new knowledge and semantic relations between existing abstract concepts and concrete objects.

Following the idea of Semantic Reflection, each traverser is represented as a node or a set of nodes in the semantic net. This has several benefits. By integrating traverser objects into the knowledge base their access encompasses all semantically reflected objects and module-related data, including the other STs and their processing results. Hence, interface specification between cooperating or dependent STs is reduced to a minimum. The position of traversers can be visualized and their behavior is visible which simplifies debugging and prototyping of programs and STs.

Semantic Traversers provide a straightforward representation of their current and preceding actions. Depending on the level of Semantic Reflection that the developer wants to achieve, STs can create a detailed trail of their actions or just indicate their current position and status inside the semantic net. For collections of similar traversers working on the same set of nodes, the trails should be detailed enough to distinguish between the effects of different traversers. Highly collaborative traversers must also provide hints for other traversers or programs that may use the knowledge base. Traverser routines can become complex applications and perform knowledge-based operations for VR/AR systems on the given networks.

The use of subroutines, classes and developed libraries have become the foundation of classical programming. Hence, we adopted this concept for the STs on the presented semantic networks to provide established approaches in addition to the new Semantic Reflection paradigm.

By using a low-level C++ implementation for the STs and for the semantic network, real-time applicability for VR/AR environments is preserved. The general ST interface is very compact:

Class `Traverser`

```
1  Traverser(bool debuggingEnabled)
2  SemNet applyTo(SemNet execDom, SemNet krl)
3  SemNet applyTo(Node n, SemNet krl,  SemNet context = 0, SemNet metaInfo = 0)
```

A traverser is applied to a node or an execution domain. If a traverser does not start its operation at a specific node, the latter is used. In most other cases, a node and the knowledge representation layer are passed to the traverser. In addition, context information and meta information can be passed to a traverser. Implemented traversers hold the names of the relations they work with as constants to improve consistency and documentation.

The individual behavior of STs depends on the domain of application, but they follow a general pattern. STs have a specific set of relation types which are followed. The designated nodes are then checked for their type and specific STs are applied w.r.t. the node type. If no STs exist for the respective node type, node attributes and other semantic information is used to assemble the result or to arrange the next traversals.

In the following we will present a set of basic traverser routines which have been subsequently developed and tested. They offer simple usage and further provide the means to synthesize application logic or data flow constructs.

3.2 Traverser manager

To manage the multitude of traversers which simultaneously acts on a semantic network, a traverser manager was developed. Figure 3 shows the manager which is semantically reflected and maintains traverser instances and relations to them. To utilize a traverser, the manager is queried and then returns the necessary reference.

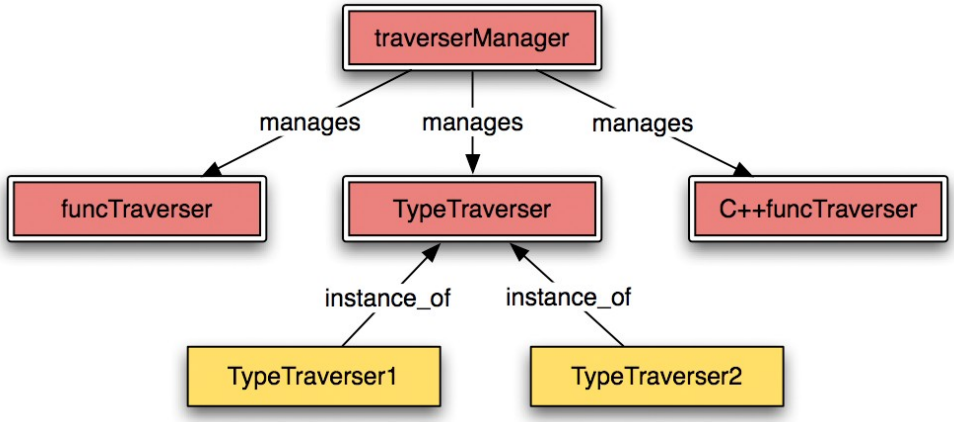

Figure 3: The traverser manager can reference all traversers and is responsible for the instantiation and administration of active and inactive traversers.

3.3 Type hierarchy traverser

A *type hierarchy traverser* is a traverser that performs basic type checking or type collection tasks. The relations it uses are the two classical AI relations "is_a" and "instance_of". It is assigned to a node to search for a specific parent type (checking if a given node is of the type "C++ function") or to collect all parent types of a node in a resulting semantic network. Two more useful search types have been realized. This traverser accounts for tasks which correspond to those of runtime-type systems (RTTS) and is utilized by almost all traversers. It is a foundation of knowledge-driven access to concepts, objects, properties and ontologies, since the type of a node restricts the actions which can be performed with it. The type traverser functionality is described by the following pseudo code fragment, when passing a specific starting search node, a KRL and a semantic context network to it:

Function `applyTo`

Input: search start node (node), Knowledge Representation Layer (krl), context network (context)

Output: Semantic network containing all nodes which are found by the type traverser w.r.t. the search type (result)

1 **Function:** applyTo(node, krl, context)

2 **begin**

```
    Get search type from context
    Connect this traverser and search start node with relation "checking_type"
    foreach node that is reached from the start node through „inst_of" or „is_a" relations do
        if current node is not marked as visited by this traverser then
            Connect current node and this traverser with relation „visited_by"
            switch search type do
                case FIND FIRST SUPERTYPE
                    Add current node to result
                    Return result
                case FIND FUNDAMENTAL SUPERTYPE
                    if current node has no outgoing "is_a" and "inst_of" relations then
                        Add current node to result
                end
                case FIND ALL
                    Add current node to result
                end
                case FIND SPECIFIC TYPE
                    if current node has SPECIFIC TYPE then
                        Add current node to result
                        Return result
            end switch
        end if
    end foreach
end
```

3.4 Function traverser

The *function traverser* can be applied to an executable node. This means it is a subtype of "function". The traverser checks for the corresponding programming language and then assigns an appropriate Semantic Traverser to conduct the actual function execution.
An active *type traverser* that checks a C++-function node is shown in Figure 4. The traverser is checking the node "playSelectionSound" for its type to utilize an adequate ST. In this case it would be the *C function traverser* (see next section).

3.5 C function traverser

Figure 4 depicts a semantic network that can be traversed by a specialized *C function traverser* to execute functions from the C and C++ programming languages. The used functions can be imported from arbitrary libraries by connecting the function node to a specific library node. This includes very simple functions with no results and parameters as well as ones with multiple arguments and default arguments. The result of the executed function is returned by the traverser inside a node, using the previously explained slot system. By coupling parameter to result nodes

they can be used as parameters for following functions the same way they can be combined in classical programming.

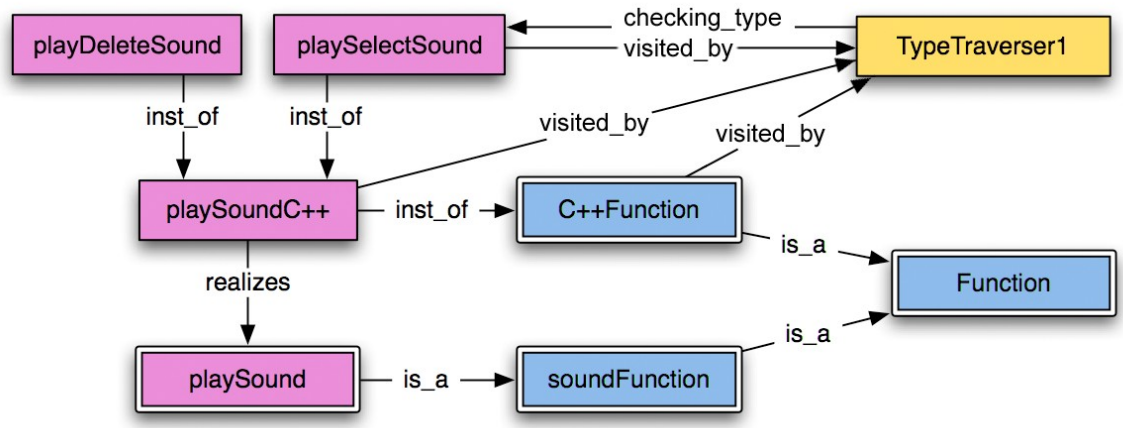

Figure 4: A function call ontology that is used by the semantic *C function traverser*. The two nodes „playDeleteSound“ and „playSelectSound“ are two calls of the same function with different parameters (e.g. a sound file). They are executed in the finite state machine shown in Figure 1. Parameter and result nodes are omitted.

3.6 Semantic network function traverser

A *semantic network function traverser* performs actions on the semantic network itself. It can instantiate, delete and modify nodes and relations and alter slot information of nodes, thus closing the loop to full Semantic Reflection. By using this traverser, one can use the semantic network to modify itself. Though it would be possible for other traversers to perform all of their actions on the semantic net through this traverser, for performance and comfort reasons they usually don’t.

3.7 Condition traverser

The *condition traverser* can be applied to a node to check for conditions at certain points in the control flow of an application. It collects type information of condition nodes, which might be nested through logical operations like *AND*, *NOT* and *OR*. At these nodes, functions are executed and their results combined to return the final state of the condition. If the conditions are fulfilled, an associated function node will be executed.

3.8 Finite state machine traverser

Finally, a *state machine traverser* was developed as a case study to test and demonstrate the traverser routines mentioned above. It combines the different tasks of the basic traversers to realize a well-known behavior model of AI [Mill06]. This traverser can operate on a semantic network similar to Figure 1.

3.9 Traverser for multimodal interaction in virtual environments

The temporary Augmented Transition Network (tATN) [Lato02] is a core component for multimodal interaction inside the *Virtuelle Werkstatt*. The integration of speech and gesture input into VR/AR simulations is a key element in creating immersive experiences. We migrated the existing ATN into the semantic network that makes up the KRL. Afterwards, an ST was developed that implemented the functionality of the existing library. This Semantic tATN Traverser can now be used as one module of a complex VR application to realize multimodal interaction.
Additional Semantic Traversers which implement further classic AI methods are developed in ongoing works at our lab. These traversers include field routing algorithms, decision trees, path planning (A* with interchangeable heuristics) and neuroinformatic learning approaches like artificial neural networks. A scripting interface for traversers and our semantic network implementation has been made available as well.
By creating a modular set of traversers, that uses the semantic net as its primary data structure, it is possible to create application logic ranging from simple AI methods to Intelligent Virtual Environments. The application can use the semantic network as a knowledge database and module connector. This supports homogeneous design and greatly simplifies debugging. Furthermore, it enables developers to focus on very few tools without requiring in-depth knowledge of programming techniques, as we shall see in the next sections.

4 Modeling Traversers with Semantic Reflection

An additional major strength of Semantic Reflection as a design principle is its intuitive usage. Semantically reflected applications or routines already incorporate a visualization and thus it is very easy for human minds to comprehend relations between abstract concepts and objects. Moreover, there are various visualization schemes which enable humans to see patterns which would not be found as easily when using linear rules [Sowa87].
Following a simple concept, Semantic Reflection can be achieved for traversers by providing as much meaningful information to the knowledge base as is available. It is possible to reflect internal variables of routines inside the semantic network, but its usefulness is limited if no knowledge- or simulation related actions are performed with them.
The different subdomains and their respective nodes should be chosen to reflect the distinct partitioning given by the data of different modules, general knowledge bases, lexical databases and other components which are used in the simulation.
Careful consideration must be given to the creation of new ontologies and traverser types, as the use of relation types like “executes”, “is_a” and “followed_by” should be consistent throughout the whole knowledge base and traverser hierarchy. When working with an appropriate editing tool and existing ontologies, such problems are solved by creating the necessary ontology links automatically (see section 5). The documentation of existing and future Semantic Traversers explains the relation types and their use inside the traversers.

Though Semantic Reflection is strongly encouraged, it is not enforced by the chosen approach. This leaves experienced designers the choice to use any combination of programming techniques and existing libraries or toolkits. Semantic Traversers can easily communicate through the same or even different semantic networks. But if required, they might also be interconnected by another top level application that controls them and manages data interchange between their routines directly.

Multiple functions of *one* programming language being subsequently called by using different nodes of the semantic network could be replaced by combining them into one function call that invokes all of the other functions. Apparently leaving the concept of Semantic Reflection at this point would have drawbacks, as it is more difficult to reflect system changes in the knowledge base with a monolithic function call. Furthermore, successive function calls from *different* programming languages are only a matter of available traversers in our system while it would require external libraries and cautious design otherwise.

We have illustrated the principles of designing knowledge bases and Semantic Traversers based on Semantic Reflection. In the next section we will specify the requirements which arise from these principles. We will also present a tool that combines the necessary functions to form a simple and powerful editor to interact with the enriched semantic networks which we have developed.

5 Creating an adequate editing tool

The full potential of Semantic Reflection is utilized by offering a proper visualization and editing software. With such a tool it is easy to incorporate application data and logic into the knowledge base. The illustration of program flow enhances debugging capabilities as well as it allows an easy understanding of application logic. Simultaneous editing features make it possible to test and reconfigure applications or VR/AR simulations at runtime. For example, a single change of one relation can then toggle between two completely different sets of simulation modules.

The respective tool developed is depicted in Figure 5. It provides intuitive visualization as well as simple usage to support work with a diversity of different ontologies and concepts. Instantiation of abstract concepts to concrete objects is accompanied by selection and grouping features to store existing and new elements inside subdomains and nodes. Selection, duplication, deletion and movement of nodes and relations follow common interface rules.

Basic ontologies are loaded into a repository. Concepts and object templates can then be dragged into the workspace where the necessary connections are created automatically. Such ontologies and templates are created as semantic networks with the same editor.

The editor facilitates editing of static semantic networks to create applications. In addition, an active semantic network inside a running application may be passed to the editor to allow keeping track of the system state and its data flow. It is possible to modify running processes by changing variable values as known from debugging tools. To improve virtual prototyping efforts, changing the whole application architecture during runtime can also be achieved in our approach.

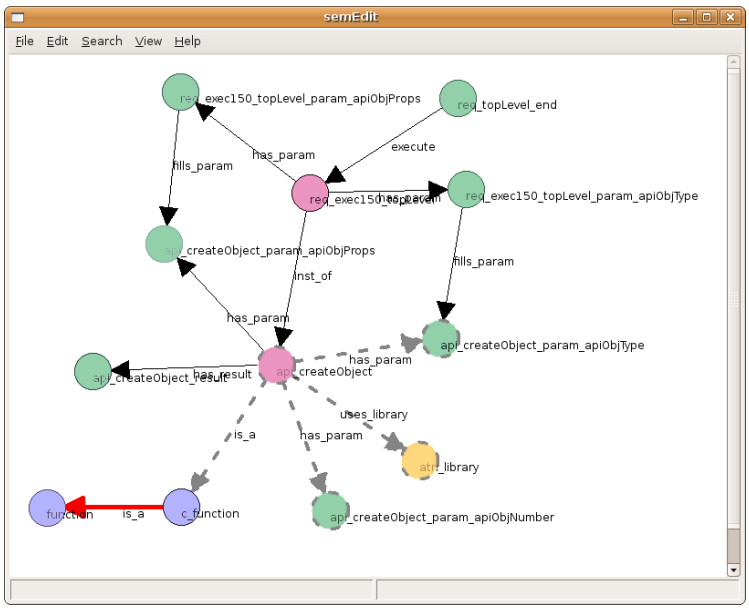

Figure 5: The semantic network editor. This screenshot displays a sample network loaded and arranged by an early version of the editor. Selected nodes and relations are dotted and animated. A thick, red relation indicates it is being traversed.

6 Conclusion

By offering a general and intuitive representation of application logic, scene graphs and a diversity of simulation-related module data, the development of IVEs is significantly simplified. Semantic Traversers provide the basic routines to create such a representation and perform calculations on it while implementing the principle of Semantic Reflection.

This formalism allows the creation of long-term-usable applications which are described in an abstract manner. The semantic representation is platform-independent and easily modified without requiring knowledge of specific programming languages. Reusable traversers offer parameterizable access to the semantic network.

Extensibility is ensured by a common and unrestricted interface to all components. This interface is the KRL. It is also possible to integrate proprietary software into the framework to permit usage of existing libraries.

The versatility of STs was shown in several examples. These examples covered simple AI routines as well as complex multimodal integration modules for VR/AR simulations.

To create and maintain large scale Virtual Reality environments and intelligent agents, a suitable tool was presented that handles and visualizes the necessary data structures.

References

[AllGoLe04] Allard, J., Gouranton, V., Lecointre, L., Limet, S., Melin, E., Raffin, B., Robert, S.: FlowVR: a Middleware for Large Scale Virtual Reality Applications. Proc. of Euro-par 2004, pp. 497-505, Pisa, Italia, 2004.

[AylLu00] Aylett, R., Luck, M.: Applying Artificial Intelligence to VR: Intelligent Virtual Environments. Applied Artificial Intelligence, Volume 14(1), pp. 3-32, 2000.

[BiJuLa02] Biermann, P., Jung, B., Latoschik, M.E., Wachsmuth, I.: Virtuelle Werkstatt: A Platform for Multimodal Assembly in VR. Proceedings of the fourth Virtual Reality International Conference (VRIC 2002), pp. 53-62, Laval, France, 2002.

[Brach79] Brachman, J. R.: On the epistemological status of semantic networks. Associative Networks, (Ed. Findler, N. V.), pp. 3-50, Academic Press, New York, 1979.

[CavPa00] Cavazza, M., Palmer, I.: High-Level Interpretation in Dynamic Virtual Environments. Applied Artificial Intelligence, Volume 14(1), pp. 125-144, 2000.

[ElNa00] Elmasri, R., Navathe, S. B.: Fundamentals of Database Systems, 3rd edition, Addison Wesley Longman, 2000.

[HeuScLa05] Heumer, G., Schilling, M., Latoschik, M. E.: Automatic Data Exchange and Synchronization for Knowledge-Based Intelligent Virtual Environments. Proceedings of the IEEE VR, pp. 43-50, Bonn, Germany, 2005.

[KaChMo06] Kalogerakis, E., Christodoulakis, S., Moumoutzis, Nektarios: Coupling Ontologies with Graphics Content for Knowledge Driven Visualization. VR '06: Proc. of the IEEE conference on Virtual Reality, pp. 43-50, Washington, DC, 2006.

[LatFr07] Latoschik, M. E., Fröhlich, C.: Towards Intelligent VR: Multi-Layered Semantic Reflection for Intelligent Virtual Environments. Proceedings of the Graphics and Applications 2007, pp. 249-259, Barcelona, Spain, 2007.

[LatFrWe06] Latoschik, M. E., Fröhlich, C., Wendler, A.: Scene Synchronization in Close Coupled World Representations using SCIVE. The International Journal of Virtual Reality, Volume 5(3), pp. 47-52, 2006.

[Lato02] Latoschik, M. E.: Designing Transition Networks for Multimodal VR-Interactions Using a Markup Language. Proceedings of the 4th IEEE International Conference on Multimodal Interfaces ICMI, pp. 411-416, Pittsburgh, Pennsylvania, 2002.

[LatSc03] Latoschik, M. E., Schilling, M.: Incorporating VR Databases into AI Knowledge Representations: A Framework for Intelligent Graphics Applications. Proc. of the 6th International Conference on Computer Graphics and Imaging, pp. 79-84, 2003.

[Mill06] Millington, I.: Artificial Intelligence for Games. 1st edition, Morgan-Kaufmann, SF, CA, 2006.

[Minsky74] Minsky, M.: A Framework for Representing Knowledge. AI Memo 306, MIT AI Laboratory, Cambridge, 1974.

[ReiVoBe02] Reiners, D., Voss, G., Behr, J.: OpenSG: Basic concepts. 1. OpenSG Symposium, www.opensg.org/OpenSGPLUS/symposium/Papers2002/Reiners_Basics.pdf, 2002.

[RohHe94] Rohlf, J., Helman, J.: IRIS performer: a high performance multiprocessing toolkit for real-time 3D graphics. SIGGRAPH '94: Proc. of the 21st annual conference on Computer graphics and interactive techniques, pp. 381-394, New York, NY, 1994.

[Sowa87] Sowa, J. F: Semantic Networks. Encyclopedia of Artificial Intelligence, (Ed. Shapiro, S.C.), Wiley and Sons, New York, 1987.

[Tra99] Tramberend, H.: AVOCADO - A distributed Virtual Environment Framework. Proceedings of IEEE Virtual Reality 1999, pp. 14-21, Houston, Texas, 1999.