# A Uniform Semantic-based Access Model for Realtime Interactive Systems

Dennis Wiebusch* Marc Erich Latoschik†

University of Würzburg

## Abstract

This research presents a uniform semantic simulation state representation and access model for realtime interactive systems (RIS) in the field of Virtual, Augmented, and Mixed Reality. The role of this model is to provide a uniform interface to a centralized virtual world state, and simple mechanisms to manage all simulation components acting on it. It addresses the low maintainability and reusability of the traditional non-uniform world access schemes. The proposed model is based on two fundamental requirements: sharing a common simulation state and updating it via events. The state is structured around an entity-model, which is combined with a central registry that provides symbol-based semantic access.

**Index Terms:** D.2.13 [Software Engineering]: Reusable Software; K.6.3 [Management of Computing and Information Systems]: Software Management—Software development

## 1 Introduction

Realtime Interactive Systems (RIS) built for Virtual, Augmented, and Mixed Reality often combine a wide variety of different hard- and software elements. A plethora of input and output devices as well as dedicated software exists, including low-level device drivers, special-purpose libraries for, e.g., graphics, physics, or artificial intelligence, and complete high-level software frameworks.

As a result, RIS applications often are closely coupled in order to fulfill a specific use case. This prevents modularity, which in turn considerably hinders maintainability, reusability, and extensibility of RISs. Modification of such systems requires a holistic understanding of the interrelationship of the involved software parts, which are often complex, not documented, and only fully understood by few members of the core development team.

In addition, RIS applications need to exploit available computational resources due to their realtime nature. Scalability on parallel architectures often relies on shared-memory and multi-threading concepts. Hence, the need for synchronization mechanisms instantly arises, often resulting in even more incomprehensible, closely coupled program code.

RIS-architectures should aim for decoupling and high cohesion to allow specialized developers to concentrate on their area of expertise and to alleviate the need for an understanding of the underlying complex system. A common approach to the coupling problem is to adopt some form of component-based development (CBD) and to define software interfaces in order to decouple simulation engines and application content. But, given such an interface, there still remain numerous different ways to handle and manipulate application-content (e.g., virtual objects), simulation engine settings (e.g., global gravity), and hardware configuration (e.g., available sensors or cluster nodes).

*e-mail: dennis.wiebusch@uni-wuerzburg.de
†e-mail:marc.latoschik@uni-wuerzburg.de

While component-based approaches facilitate reusability and parallelization, they do not ensure the creation of comprehensible application code. Reasons for this include low cohesion, cryptic identifiers, and vast, complex software interfaces.

In this paper we present our approach of a uniform symbol-based access model for Realtime Interactive Systems. Its human-readable, cohesion-furthering, minimalistic nature aims at easing the development and maintenance of RIS applications for simulation engine developers, application developers, as well as game designers. At the same time, it facilitates the creation of extensible, concurrent programs and furthers decoupling of simulation components. In addition, its inherent link to an ontology provides a semantic grounding for all aspects of a RIS. This is highly beneficial for symbolic AI approaches, which are necessary for, e.g., virtual agents with reasoning capabilities or advanced human-machine interfaces using speech and gesture.

## 2 Related Work

### 2.1 Aspects of RIS Development

A plethora of RIS frameworks and toolkits has been developed in the past decades [20]. Despite the huge amount of research and software development work on this topic, many unsolved problems still exist [25, 29]. For example, the complexity of large virtual reality (VR) applications is addressed by [2]. The authors argue that application modularity is beneficial for coping with complexity. Even though early work focused on decoupling components of RISs [23], close coupling and low cohesion was still identified as a problem for RIS frameworks in more recent publications [15, 22].

The realtime nature of RIS applications requires extensive hardware resources, which nowadays is addressed mainly by large-scale distribution and concurrent architectures. Similarly, different update rates of sensors and simulation modules inherently require concurrency [23]. Distribution and concurrency complicate the development process [17]. Possible solutions use client/server architectures [23] or adopt the actor model [7, 9, 16].

Four approaches have shown to be highly beneficial for RIS architectures:

1. **Event-based** communication facilities [6, 24, 26] provide a decoupled control flow mechanism.

2. **Entity-centered** content models [6, 18, 24] provide an object-oriented view on and access to the virtual scene.

3. **Aspect-based** subdivision of entities [3, 28] fosters cohesion for entities simulated by several different components.

4. **Ontology-based** semantic layers [3, 28] facilitate integration of entities within an application and provide inherent links to symbolic AI models. Conceptual [5] and semantic [4, 13] modeling approaches also provide potential solutions to the reported lack of high-level design facilities for VR applications [8, 19].

The next section will discuss some prominent examples of VR frameworks. We will specifically focus on the simulation state representation and access, the general application model, support for events, creation of simulation components, and coupling between simulation components. At the same time we will look at advantages and disadvantages of these aspects with respect to comprehensibility and flexibility of an application created with the respective framework.

## 2.2 Existing RIS Frameworks

### 2.2.1 Object Oriented Frameworks

Our first example is VRJuggler [10] an object oriented VR framework. At runtime, VRJuggler-based applications are managed by the framework's microkernel. This kernel abstracts from the underlying hardware as well as from input/output modules (e.g., tracking systems and rendering components). For this purpose, fully independent managers, which are provided by the framework, are plugged into the kernel.

VRJuggler applications can access input and output facilities through the kernel. The configuration of the respective device is facilitated by so called *chunks*, which consist of named and typed fields. Representation of and access to the simulation state is entirely designed by the application developer. VRJuggler does not provide an event system for simulation events.

The advantages of an object oriented framework implementing a microkernel-based plugin system lie in the abstraction of the hardware layer and execution schemes. No synchronization mechanisms need to be used by the application developer when accessing components of the framework. Due to the object oriented paradigm, devices and other objects can be interpreted as instances of classes, a very understandable representation. The utilization of a flexible configuration mechanism based on chunks is another important feature. Besides software interfaces that have to be implemented, VRJuggler does not impose restrictions on the developer, but also does not give further guidance for structuring an application. Hence, the management of the simulation state and creation of simulation components is completely left to the developer. Consequently, the comprehensibility and flexibility of VRJuggler applications mostly depend on the design choices made by their developers.

### 2.2.2 Data Flow Graph-based Frameworks

The next examples are FlowVR [1] and Avango [27] (as well as its successor AvangoNG [12]). Both frameworks utilize the concept of a data flow graph to specify contents of an application. Such a graph is composed of modules, which contain parts of the application content and logic. The communication between modules is achieved by connections of their input and output fields. Both frameworks allow for instantiation and integration (by connecting fields) of new modules at runtime.

FlowVR modules run a (potentially endless) concurrent loop, which occasionally stops and waits for new data arriving at the module's input fields. Events are supported in the form of special fields called event ports.

In Avango a module can either directly handle new incoming data or access the current values of its input fields in an evaluate method which is called once per frame. Event handling again is mapped to specialized input fields.

Data flow graph-based applications have multiple advantages: The computation code is forced to be modularized, hence modules can be reused and coupling between simulation components is reduced. Furthermore, the concept of composing an application is easily understandable and the application graph can be graphically represented. In general this is beneficial for comprehensibility of created programs. The modularized nature of data flow graph based applications benefits their reusability and extensibility. However, the application state is not explicitly represented and large graphs tend to become incomprehensible. In addition, circular graphs and concurrent operations often pose a problem, restricting the available execution schemes.

### 2.2.3 Message-based Frameworks

A successful message-based framework is DIVE [6, 24], which mainly focusses on distributed virtual environments. Distribution is realized using sophisticated peer-to-peer networking techniques utilizing IP multicasting. It supports streams of video, audio and internal updates of the virtual environment. Simulated virtual environments are represented by means of an entity model, which is stored in a distributed database. Entities are hierarchically grouped in a scene-graph like structure, allowing for multifaceted data transfer optimizations. Updates of the database are communicated utilizing the integrated message-based event system.

In subsequent work, the initially thorough programming interface was extended to support C, C++, and Java as well as the TCL scripting language, providing the programmer with rich interfaces. Furthermore, the event-based architecture of DIVE was extended by component-based aspects.

The clear advantage of the DIVE framework is its inherent decoupling of processes, due to its event-based communication architecture. The use of shared entities provides a unified view on the application state, which is managed by the DIVE framework. DIVE's revised and extended programming interface does not impose a specific way of implementing applications. Hence, as with VRJuggler, comprehensibility and flexibility of developed applications depend on the design choices taken by their developers.

The I4D framework [8], like the DIVE, implements message-based communication but also can be categorized as a component-based framework. It represents both scene content and simulation components as independent actors, whose state is defined by their attributes. Actors can perform actions by modifying these attributes. In addition, actions which can run concurrently and modify the attributes of actors can be defined. Events are supported via multicast and broadcast messages. All I4D components are managed in a scenegraph-like hierarchy. Inter-component communication is realized using a string-based message interface or by accessing public attributes of actors. New components can be instantiated at any time using a factory pattern-like mechanism.

The advantages of the I4D framework lie in the combination of a message passing architecture with a component-based design. This allows for decoupled, reusable components and a comprehensible representation of the virtual scene. The restriction to string based messages slightly reduces the flexibility of the framework and requires a mechanism to structure the message content. Due to low coupling, I4D applications are highly flexible. Their comprehensibility depends on the way messages and events which are created by the developers are handled.

### 2.2.4 Component-based Frameworks

NPSNET-V [11] is a component-based framework that separates loosely coupled simulation components (called modules) from the application state, which is represented using an entity model. Furthermore, it features a publish/subscribe event system, especially used to signal property changes of modules and entities. Both, the modules and application state are managed in containment hierarchies. The hierarchical management of modules allows to easily enable and disable modules sharing the same functional role. New modules can be created at runtime by adding them to an container in the module hierarchy. Data is stored in so-called *properties*, which can belong to modules as well as to entities. In addition to a set of basic properties, a developer can choose to define new ones.

Entities have to be implemented according to the well known model-view-controller pattern. Controllers (e.g., KeyController) have write access to an entity's properties, whereas the views (e.g., GLView) are responsible for displaying it. Entities as well as modules can be accessed by reference to the path under which they are registered in the respective hierarchical data structure.

The clear separation of simulation and data is beneficial for the flexibility of applications and a hierarchical structure to manage elements of the application is a sensible approach, since it relieves the developer from creating self-developed management structures. Given an understandable naming scheme, the comprehensibility of an application is furthered by this approach.

VHD++ [21] is a highly elaborated, component-based framework using the object oriented programming paradigm. It features an elaborated event system, in which events are split into system and simulation events. As in I4D (section 2.2.3) single, multi, or broadcast events are supported. The application state is reflected by a hierarchy of properties, managed by property managers. Simulation modules are represented by decoupled services, which can access the simulation state via property managers. The creation and registration of properties and services is handled by designated loaders that have to be provided by the respective developer.

The benefits of a component-based framework, like VHD++ and NPSNET-V, are their flexibility and inherent separation of concerns. By defining software interfaces for components and decoupling components from the data representation, reusable software elements can be created. On the downside, interfaces for components tend to be complex and developers need to spend much time on understanding them.

#### 2.2.5 Observations

Besides the diversities of the presented frameworks and approaches, some equivalences can be observed. For example, the connections of a data flow graph can be interpreted as message channels and the field of the modules in a data flow graph can be seen as properties in VHD++ or attributes in I4D. Similarly, an actor in I4D can be interpreted as a FlowVR module or the combination of an NPSNET-V entity and its controller.

Our examination of existing frameworks reveals beneficial aspects for comprehensibility of a framework and the applications developed with it: The object oriented paradigm facilitates encapsulation and allows to create understandable representations. Data flow graphs provide the possibility to visualize an application graph and foster modularity and thus decoupling and cohesion. Component-based approaches emphasize separation of concerns and, hence, further decoupling and reusability by providing interfaces for the simulation components to be developed. This allows to independently develop software modules which can then be reused by other developers. A message-based approach facilitates decoupling by cutting down software interfaces to support for message types. In this context, an extensible mechanism to define naming schemes and data types is a desirable feature.

Our Simulator X framework[16] is hard to classify into the mentioned categories, since it already combines component-based development, message-based communication and object-oriented programming (in terms of its entity model). Although emphasizing decoupling [28], it fails to provide a unified way to access components and application state. Consequently, the mere combination of the examined approaches does not inherently result in comprehensible applications.

But comprehensibility of a framework, for component developers as well as for application developers, is one of the most important aspects regarding a short learning curve and reusability of created applications. Hence, we aim at a minimal interface that provides as many of the beneficial features reported above as possible.

In the next section we will develop a model of RIS applications upon which this minimal interface shall be built. It should be noted that the next section provides concepts rather than instructions for an actual implementation, which would include optimizations to maintain realtime capabilities.

## 3 Unified Access Model

### 3.1 The Core of RIS Applications

Every (RIS) application simulates some kind of virtual environment. Although most instances do, such environments do not necessarily have a VR-like character, nor do they incorporate 3D content. Since being simulated, each of those environments inevitably contains virtual aspects. But, due to its interactive nature, at least the user introduces a real aspect. Consequently, each simulated environment always combines virtual and real aspects.

Examples for virtual objects are simulated entities in the virtual scene, virtual sensors (like random number generators or application time), but also parts of the system architecture (like simulation engines). On the other hand, input devices (like keyboards or tracking targets), hardware devices (like the tracking system or a screen), as well as the user himself serve as examples of real objects involved in a typical RIS application. Hence, our initial observation is that a unified access model will have to treat virtual objects in the same way as it treats real objects.

Looking on RIS applications as the simulation of environments, the representation of the environment's *state* constitutes a central building block. Due to the discrete nature of computer programs, a program run can be interpreted as a sequence of such states. In this context, each simulation state conceptually consists of a number of properties the representation of which we will call *state variables*, which entirely define all aspects of each object that is part of the simulation:

$$State_n = \bigcup_{i \in I_n} StateVariable_i$$

In the presented equations $I_n$ depicts the index set that contains the indices of all state variables belonging to a given state $n$.

The set of state variables changes each time a component of the simulation computes updated values or a sensor provides new data. Thus, we will count the application as a set of properties (the current simulation state) and changes to those properties.

Consequently, a way to represent changes to the simulation state, which consists of elements that remain stable until the next state change, has to be found. Each change can—again due to discretization—be viewed as a singular *event*. Such events can represent a simple property change but also depict certain incidents, for example, the collision of two objects.

In contrast to state variables events do not have a lifetime. Otherwise they would not identify a specifc point in time (e.g., a *collision* event) but a part of the state (e.g., an *inCollision* state variable).

### 3.2 Changes to the Application State: Events

As stated above, changes to the application state are caused by either the arrival of new sensor data or the computation results of a simulation component. Aiming at a unified view of RIS applications, we will treat both as an event. Regarding an application state $n$, an event $i$ can be represented as the two sets $Add_i$ and $Remove_i$ of added and removed state variables (updating is represented as removing and re-adding, though an actual implementation would update contents of state variables):

$$Event_i^n = \{Add_i \not\subset I_n,\ Remove_i \subset I_n\}$$

Given a certain state $n$, the observation of an event will result in the transition into a subsequent state $n+1$:

$$State_n \circ Event_i^n = State_n \setminus Remove_i \cup Add_i = State_{n+1}$$

Further events, like the before-mentioned collision of two objects, can be used to trigger reactions in simulation components. Although the occurrence of such an event does not directly affect state variables, its processing will eventually result in an update of the application state.

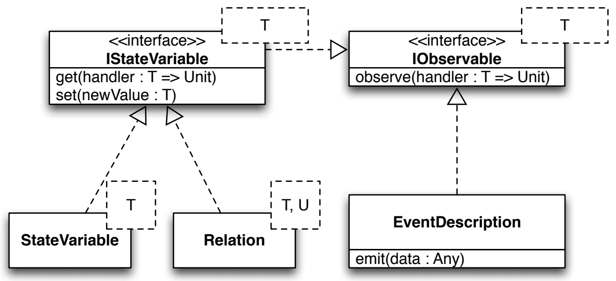

Figure 1: Software interfaces for state variables (see section 3.1), relations (see section 3.5), and events (see section 3.2)

Based on the two concepts of state variables and events, all structures commonly used in RIS applications (e.g., state machines, scene graphs, or data-flow graphs) can be modeled. Since the state variable concept only encapsulates the processing of value update events, its implementation is optional. More precisely, starting with an empty state, the current state can be seen as the chain of events since the program started. However, we recommend its implementation in order to relieve the developer of keeping track of the state change events and to provide a more common interface.

Consequently, the event system is the only interface between the application and simulation components as well as between the simulation components themselves. Because the pure, unstructured handling of events would result in unmaintainable, incomprehensible applications, a mechanism to allow structured processing has to be provided. For that purpose, hiding of the event processing as done by state variables can be applied more generally: Using a unified software interface, callbacks (which we will call *handlers*) can be registered for handling events of any kind. This way, event handling code can be attached directly to a respective event source (which may be an updated state variable). This interface is implemented by state variable and event representations (see figure 1).

## 3.3 Subdivision of the Application State: Entities

Although it would be sufficient to combine the presented two building blocks—state variables and events—in order to connect simulation components and create RIS applications, the resulting program code would not be comprehensible. Thus, state variables are logically grouped into *entities*, thereby creating an intuitive representation of simulated objects. This complies with the object-oriented programming paradigm and the requests for entity centered models, since all properties that belong to one entity are bundled. No state-variable that does not belong to an entity must exist and state-variables must not be directly shared by two entities:

$$Entity_k^n = \bigcup_{j \in J_k^n} StateVariable_j \quad \text{where} \quad \dot{\bigcup_l} J_l^n = I_n$$

However, indirect sharing by incorporation of other entities is feasible, as a state variable may hold the reference to an entity. In consequence, everything that is relevant to the application state is represented as an entity (a set of state variables), e.g., virtual objects, the user, input devices, or simulation components. This satisfies our initial request: Real and virtual objects can be treated equally.

While representing real and virtual objects as entities consisting of properties is quite an accepted approach, it is less common to treat the user, input devices, and especially simulation components the same way. Yet, all of those objects can be defined by means of associated state variables. For input devices, for example, this could be the state of buttons or a tracking-target's position, for simulation components it could be the respective component's configuration.

Adopting this representation, a programmer just has to learn what can be modified, instead of what *and* how it can be modified. Besides uniform access to all relevant properties of any aspect of the simulation, it enables uniform creation, handling and removal of all architecture elements.

## 3.4 Setting up the Application State: Entity Creation

Having developed a unified way to access the simulation state via entities, the creation of entities remains an open question: Obviously, the initial values for the entity to be created, more precisely for the state variables it contains, have to be specified. In some cases more information than the plain state values is required (like configuration or model files that have to be parsed). Since this is usually done by the application developer, a description-based mechanism is a reasonable approach (see section 4.2).

The instantiation of the entity then is triggered by a designated event, containing the above-mentioned initial values. Depending on this information the responsible authority has to be detected: For the creation of new components this is a central element of the respective RIS framework, e.g., a central component registry. For any other entity the involved simulation components have to be requested to instantiate their local representation of the entity.

Here, the essential observation is that the only information required to instantiate a new entity—be it a component, an input device or any other object—is the set of initial values information on the involved components. Consequently, all entities can be created using the same mechanism.

## 3.5 Linking the Application State: Relations

Given the subdivision of the application state into state variables and entities, a means of relating all parts to each other is required. For this reason we introduce the concept of *relations* between entities: A relation associates two entities which satisfy the relation. For example, the *has-part* relation identifies all entities that are a part of another entity. Depending on its implementation, a relation may be of transitive nature: A has-part relation could identify either only direct parts or all parts and sub-parts of an entity.

A relation has two characteristics: First, it can be regarded as a state variable, since it represents a part of the application state. Hence, the interface implemented by a relation is the same as the interface of state variables (*get*, *set*, and *observe*). In addition, it provides a means to query relations between entities. For this purpose it can be partially specified, leaving the queried aspects open. In the above example the subject of the has-part relation could be undefined to query all entities which the given object is a part of.

Due to the similarity to state variables, the unified access model is not restricted by the existence of relations. The second characteristic provides developers with a tool to define and query relations of entities in an application.

## 3.6 Modifying the Application State: Components

Every RIS application comes to life by means of simulation components, like physics engines, artificial intelligence (AI) components, or input/output modules. Often data flow graphs are implemented to control the course of events in an application (cf. section 2.2.2). This approach integrates well with scenegraph-based rendering, but often forces developers to apply a serialized execution scheme. Although this is sufficient for most applications, integrating modules that perform lengthy, possibly concurrent calculations (e.g., AI components) can become difficult. In addition, it complicates the development of components that have to access the whole simulation state.

Frameworks that adopt component-based design are less often subject to such limitations, since they inherently aim at separation of computation tasks. However, specifying the execution scheme mostly is a complex task when using a component-based approach.

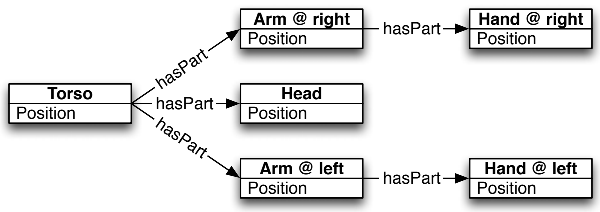

Figure 2: Simplified representation of the user's upper body.

We suggest to keep a graph-based approach to specify the execution scheme, but to combine it with component-based design and incorporate it into the entity model: Each component (like any other object) is represented by an entity. The component is configured by the entity's set of state variables, which in this case contains a *successors* state variable. The set of all successors defines the application graph. If desired, a complete data-flow network can be emulated: After finishing its (asynchronous) simulation step, all components in the successors variable are notified (via an designated event) to start their simulation step.

Using component entities, the application developer can focus on one interface (the entity) to configure the whole application: The configuration of simulation components as well as the state of the virtual (and real) environment can be read and modified via state variables, which are accessible via the encapsulating entity. Application logic can be defined by observing state variables of any entity as well as publishing and subscribing to events/state changes.

### 3.7 Understanding the Application State: Semantics

The subdivision of the application state into entities results in a more comprehensible view, but the identification of state variables and their meaning remains ambiguous. Without annotating the variables they contain featureless values, the meaning of which is only identifiable based on their names within program code. Even if the developers choose meaningful names, their uniformity cannot be ensured. As a result automated comparison of properties, integration of artificial intelligence methods, and understanding the application's structure would become a needlessly complicated task.

Since ontologies are a common means to represent concepts and their relationships using a defined vocabulary, specifying names (symbols) in an ontology to annotate state variables seems natural. In this way, human-readable information on the state variables' semantics, data type, and relations to other state variables can be stored. The names used in such an ontology can easily be converted into program code by creating a data structure containing variables having the same name as their counterpart in the ontology. This accounts for the observation that an extensible mechanism to define naming schemes and data types is a desirable feature for message-based approaches (see section 2.2.5). In addition, the transformation of ontology contents into program code facilitates the verification of semantic correctness at compile time by creating new data types combined with semantics. For example, requiring a parameter to be of the type `Position[Vec]` prevents accidental misuse of a method by passing a `Vec` representing a direction.

There are further advantages in using ontologies to represent meta-data about an application: A reasoner can check the ontology for consistency and identify misconfiguration of entities. An ontology can easily be extended by reusing existing concepts and it also allows to query existing concepts. Being independent from the programming language used, one ontology could be used by multiple frameworks. As ideally the entire application state is defined by the union of all entities/state variables, a snapshot of the current state can be saved in the ontology and be restored later.

```scala
val hasPart = Relations.get(HasPart)
val userDescription = EntityDescription(Torso, "user1")(
 GraphicalModelAspect(file = "assets/body.dae"),
 hasPart(EntityDescription(Head, "head1")(
    VRPNTargetAspect( id = 1 )
 )),
 hasPart(EntityDescription(Arm :@ left, "arm1")(
   hasPart(EntityDescription(Hand :@ left, "hand1")(
     GraphicalModelAspect( file = "assets/hand-l.dae" ),
       VRPNTargetAspect( id = 2 )
     ))
 )),
 hasPart(EntityDescription(Arm :@ right, "arm2")(
   hasPart(EntityDescription(Hand :@ right, "hand2")(
     GraphicalModelAspect( file = "assets/hand-r.dae" ),
       VRPNTargetAspect( id = 3 )
   ))
 ))
)
```

Listing 1: Representation of the user's body introduced in figure 2 written in Scala code by means of `EntityDescriptions`.

Integrating semantics into the representation of state variables and events allows to access properties on a semantic level. Hence, application logic can be defined via rules in an external format by implementing a thin wrapping layer. This would also be beneficial for eventual AI components.

In addition to the main semantics, which identify the type of the denoted state variable, *annotations* can be added. These allow to specify the semantics of the state variable in greater detail using symbols from the ontology. Figure 2 exemplifies this concept on the basis of the representation of the user's upper body, being subdivided into multiple entities.

The next section will detail our implementation with code examples taken from an application in which the collision of the user's left hand and a virtual object (a car) is detected. The lines of code can be executed in an arbitrary order (obviously with the exception that variables have to be defined before they are used). Although our implementation adopts a message-based design, there is no preferred style of implementation of the presented model.

## 4 IMPLEMENTATION

### 4.1 Entity Descriptions

As mentioned above, we propose a description-based interface. Our implementation features aspect-based descriptions for entities, enabling the flexible specification of entities. Listing 1 exemplifies the description of the torso in figure 2: An `EntityDescription` consists of a type with optional annotations (both defined in the ontology) and a name, as well as an arbitrary number of `Aspects` (see [28] for details on aspects). Relations like `hasPart` are used to create a hierarchical description, representing the structure of the torso. In the example many default values are used (e.g., scale for a graphical model).

There are multiple advantages to a descriptive approach using aspects: First, the resulting code used to describe entities is easily understandable. Second, the use of aspects abstracts component specific implementations. This makes aspects reusable for other components of the same type (e.g., a different renderer) and facilitates replacing of components. In addition, entities can be flexibly created: The combination of different aspects and relations enables the specification of a multitude of entity characteristics. Adding a physical aspect to the screen entity description in listing 2, e.g., could enable detection of collisions of virtual objects and the screen.

The required initial values for different aspects of the simulation can now be specified by the application developer. In the next section we will detail how entities are created from such descriptions.

```
1  val vrpnClientDescription =
2    new EntityDescription(Component, "vrpn-tracker0")(
3      VRPNComponentAspect( url = "tracker0@localhost" ))

5  val graphicsDescription =
6    new EntityDescription(Component, "renderer")(
7      RendererAspect( stereo = true ))

9  val screenDescription =
10   new EntityDescription(Screen, "screen-0")(
11     ScreenAspect( width = 2400, height = 1500 ))

13 val carDescription =
14   new EntityDescription(Hand, "myCar")(
15     GraphicalModelAspect( file = "assets/car.dae" ))

17 execute vrpnClientDescription.realize andThen
18   graphicsDescription.realize andThen
19   screenDescription.realize andThen
20   userDescription.realize andThen
21   carDescription.realize{
22   _ => println("created components and entities")
23 }
```

Listing 2: Examples for entity descriptions: Two components (VRPN client and renderer), the screen, the user entity, and a virtual car are created. The created components update the positions of the entities and visualize them after they were instantiated.

## 4.2 Unified Creation Mechanism

Listing 2 shows an extract of a simple application featuring a VRPN client and a renderer. Besides those two components, the screen on which the virtual scene will be displayed as well as a virtual car is created. All four entities are instantiated by defining entity descriptions containing initial values. The call to the `realize` methods starts the instantiation process. An (optional) handler is executed after the respective entity is created. The implementation-specific keyword `andThen` (used for brevity) queues calls to the `realize` methods, causing the serialized instantiation of entities.

Lines 1-7 exemplify that components are created exactly the same way as other entities (see lines 9-15). This reduces the number of concepts to be learned by a developer who is new to the framework.

As mentioned in section 3.4 the creation process depends on the contained aspects: Events containing the respective aspect are emitted to the associated components, which create their internal representation and the associated state variables. Finally, the entity is registered with the framework (cf. section 4.5).

## 4.3 Simulation Component Definition

Each simulation component has to implement a simple interface which is used by internal event handlers: It contains a method that provides initial values for an entity to be created. Most of these values will be contained in the aspect assigned to the component, but, for example, special loaders that are part of the component could read more information from files. In addition, a callback has to be implemented by the component, which is executed each time a new entity is created. Finally, a method to handle external triggering of a new simulation step has to be provided.

Since the component itself (as every other object) is represented by an entity, that entity's state variables are the interface to its configuration. In this way, the configuration of a component is handled using the exact same interface as the configuration of any other element of the simulation. The next sections will introduce our implementation of that unified interface.

Note that there is no assumption about a component's execution scheme, however, it can be triggered externally by adding a corresponding event handler, if required (compare section 3.6).

```
1  var (carPos, handPos) = (Vec(0,0,0), Vec(0,1,0))
2  val theCar : Entity //= for entity access see listing 4

4  Events.observe(Collision){ event : CollisionEvent =>
5    println("observed collision: " + event)
6  }
7  theCar.observe(Position){ pos : Vec => carPos = pos }

9  theCar.set(Position(Vec(0, 0, 0)))

11 def checkCollision(){
12   if (length(handPos - carPos) < 0.1)
13     Events.emit(Collision(hand, theCar))
14 }
```

Listing 3: Uniform access to a collision event (via the `Events` object) and a state variable (via an entity): The same software interface is used and handlers are installed at the respective object, allowing for understandable, localized code. Identical access to elements for write access results in an easily understandable interface.

## 4.4 Unified Access

The realization of the presented methods requires the implementation of the interfaces shown in figure 1. State variables, relations, and events implement the method `observe` (see listing 3), which encapsulates event handling code into a more common callback-style interface. In our example, the interface of state variables, relations, and events is also implemented by entities and the `Events` object. Addressing a state variable (or an event, respectively) by using an additional first parameter, a handler can be installed to the element (event or state variable) selected this way. This allows to hide the underlying implementation from component and application developers. Hence, a typical object-oriented approach, a message passing library (e.g., an implementation of the actor model with FIFO delivery semantics), or any other approach can be used. Our implementation builds on the underlying Simulator X framework (for details see [16] and [28]).

Regarding locality, the code benefits from the use of anonymous functions (lambda functions) since handler code can be defined at the same location as respective entity or event is accessed (compare listings 2, 3 and 4). Consequently, our model benefits from the use of functional languages, but can also be implemented using the more common approaches like callback functions. Such localized way of implementing event handlers constitutes an intuitive interface which follows the object-oriented paradigm.

Lines 4-5 of listing 3 exemplify the access to the description of an event: The `Events` object contains proxy objects for events, which enable the developer to install handlers for the underlying events and emit events of that type (see line 13). In this example, the object `Collision` is one of the concepts generated from the ontology mentioned in section 3.7 (for details see [28]). Besides its name, which serves as an identifier, it contains information on the data type of the event's payload (also specified in the ontology).

Lines 4-13 of listing 3 reveal the similarity of handling events and state variable updates: While events are accessed via the `Events` object, the same applies for entities and state variables. The value `Position` used in line 7, like the `Collision` variable from line 4, originates from the ontology and contains type information. Thus, the compiler can infer the datatype and semantics of the state variable and ensure the use of a correctly typed handler.

Listing 3 furthermore shows the interface for updating state variables (line 9) and publishing events (line 13): The accessed proxy for the collision event allows to `emit` events. Each event can carry payload, in the example this are the two colliding entities. Due to the type information attached to the `Collision` variable, the type of the data carried by the event can be checked by the compiler. This way, an event proxy object hides the creation and dispatching of the

```
1  lookupEntities(Torso :: Nil, "user1"){ body : Entity =>
2    body.get(hasPart -> ?){ bodyParts : AnnotatedMap =>
3      leftHand = bodyParts.getFirst(Hand :@ left)
4      leftHand.observe(Position){ position : Vec =>
5        handPos = position
6        checkCollision()
7  } }
8  // set stereo property of the renderer
9  lookupEntities(Component :: Nil, "renderer"){
10   renderer => renderer.set(StereoMode(false))
11 }
```

Listing 4: Example showing access to the torso entity from figure 2 by means of the world interface. The names "user1" and "renderer" of the entities were chosen by the application developer in their descriptions (see listings 1 and 2).

actual event, reducing error prone code and fostering comprehensibility. Similar publish/subscribe mechanisms for event handling can be hidden by the framework (see section 4.5) letting developers focus on implementation tasks. The same is true for creation and dispatching of state variable change events: Calling the `set` method (line 9 of listing 3) results in the creation of a respective update event. That event is automatically dispatched and the correct handlers are called.

A major advantage regarding comprehensibility is the fact that no difference (except for the method names `set` and `emit`) is made between the handling of events, state variables, and relations.

The next section details the registration of entities as well as later access to them and exemplifies the uniform access to virtual objects, input devices and components.

### 4.5 World Interface

Using the presented techniques a developer would have to keep track of all created entities. To address this, a central registry that provides access to entities, similar to the hierarchical structure used in the NPSNET-V framework, takes over this task. In addition, the publish/subscribe mechanism for events needs to be provided by the framework. A third centralized task is the instantiation of new simulation components. The central component which provides these services in our implementation is the *world interface*.

It internally handles the instantiation of components: After receiving a component aspect, it instantiates the component and creates the associated entity. The creation of this (as well as of any other entity) ends with its registration with the world interface.

Listing 4 shows an example for accessing registered entities: The `lookupEntites` method in line 1 receives a path which specifies to retrieve all entities that are torsos. Note that the types that make up the path are the same that are used in the entity descriptions from above. The returned entity is checked for its parts (which were also defined in the entity description) using the `hasPart` relation. As shown in line 9, accessing components works the same way.

The publish/subscribe mechanism for events is also hidden from the developer: A call to the `observe` method of an event description will notify the world interface about the request of this type of event. The world interface will then register the event handler and look for matching event providers. If such providers are found or as soon as the `emit` method of that event is called, a handshaking process between the provider and the handler will be initiated by the world interface. Afterwards the event provider will automatically send its events to the event handler.

### 4.6 Adopting the Actor Model

Due to the message-based nature of the presented approach, we suggest to apply the actor model [9]. Since the presented approach is entirely based on events, which can be represented as messages, it is a sensible choice. Each component and application can consist of multiple actors which can run concurrently and exploit available hardware resources. In this context, the model obviates synchronization mechanisms. Furthermore, some implementations provide a transparent network layer which immensely reduces the amount of work for the creation of a distributed system.

On the downside the actor model introduces a programming model which most of the developers are not accustomed to. However, this characteristic can be hidden from new developers by applying the proposed approach.

A second drawback is the fact that event-based models commonly miss the feature of transactional state updates. This can be implemented on top of the presented model by implementing events that simultaneously update multiple values before associated handlers are executed, or by serializing the execution of some components' simulation threads, when necessary (cf. section 3.6).

## 5 Conclusion

We presented our work towards a unified semantic-based access model, which allows to treat any entity of the system in the same way—be it software, a virtual object, or a real object. It was developed with respect to requests and experiences reported by the community in the last years. In this context, seven other frameworks were reviewed with regard to their realization of state representations, means of decoupling, and event models.

Our model is based on two fundamental requirements: Storing the simulation state and representing changes to it. The basis for our model is established by storing the entire state in state variables and applying changes to the state by the execution of event handlers. In this context, the application state is conceptually divided into entities, which are used to represent any object that is involved in the simulation. Following the well understood object-oriented paradigm, entities provide a uniform, intuitive way to access the entire simulation and its state.

Based on these essential elements, a framework implementing the proposed model can integrate existing simulation engines by connecting their internal representations to state variables. Adopting component-based development, our model facilitates separation of concerns and modularization of computation tasks. Due to its event-based nature, the application state can be distributed over multiple nodes by serializing the events and distributing them over network. In this context, a message-based implementation (e.g., applying the actor model) furthers decoupling by cutting interfaces down to message-based communication.

A common ground for naming entities is established by using semantic symbols. These are defined in an ontology which is incorporated into the framework and meant to be extended by application and component developers. The ontology is furthermore used to store relations between entities and thus supports the integration of AI components.

Due to type information defined in and generated from the ontology, our model allows to overcome the often observed problem of lacking type safety and hierarchies. The centralized, human-readable description of available types and events in the ontology eases the process of getting an overview of the systems capabilities without requiring in-depth knowledge of utilized components. By building a synchronization layer on top of the model, even the problem of transactional updates could be tackled. The concept of attaching value change handlers in the form of anonymous functions directly to entities of the simulation fosters locality and, hence, comprehensibility.

In contrast to other middleware, a component developer can hide the complexity of instantiating and controlling a created component from other developers. In combination with the model's very lightweight software interface this facilitates maintainability of applications and seamless replacement of components, usually being a very demanding task.

Altogether, the application of our model relieves a developer from figuring out how to access a specific aspect of the simulation and allows to focus on the implementation of application logic. We thus believe that our unified access model shortens the learning curve for fathoming a RIS framework and developed applications.

## 6 Future Work

The presented work creates the basis for decoupled RIS applications, written in easily understandable program code. However, attaching callbacks that observe changes of state variables and occurrences of events only poses the lowest layer for a framework adopting the presented model. Future work will include adding further layers to ease developers' efforts, combine handlers, and strengthen cohesion to further improve comprehensibility of the code.

Usability studies that consider the developer's learning curve and comprehensibility of application code should be developed and conducted to make out the best combination of such layers. Building on the results of such studies, the next step in developing a unified access model can be taken.

Furthermore, the usability of our approach for artificial intelligence methods has to be evaluated. The deeply integrated ontology poses an ideal basis for paradigms like semantic reflection [14]. Semantic [4] and conceptual modeling [5] approaches will benefit from this integration, as well.

Finally, the approach could be used to connect existing frameworks and engines, for example, Unity, the Unreal Engine, the Havoc physics engine, and many more. This way, a growing set of attachable simulation components could be created and, at the same time, this would prove the flexibility and applicability of our model.

## 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 *Euro-par 2004 Parallel Processing*, pages 497–505, 2004.

[2] J. Allard, J. Lesage, and B. Raffin. Modularity for Large Virtual Reality Applications. *Presence: Teleoperators and Virtual Environments*, 19(2):142–161, 2010.

[3] G. Anastassakis and T. Panayiotopoulos. A Unified Model for Representing Objects with Physical Properties, Semantics and Functionality in Virtual Environments. *Intelligent Decision Technologies*, 6(2):123–137, 2012.

[4] P. Chevaillier, T. Trinh, M. Barange, P. De Loor, F. Devillers, J. Soler, and R. Querrec. Semantic Modeling of Virtual Environments using MASCARET. In *Workshop on Software Engineering and Architectures for Realtime Interactive Systems*, pages 1–8. IEEE, 2012.

[5] O. De Troyer, F. Kleinermann, B. Pellens, and W. Bille. Conceptual Modeling for Virtual Reality. In *Tutorials, Posters, Panels and Industrial Contributions at the 26th International Conference on Conceptual Modeling - Volume 83*, pages 3–18, 2007.

[6] E. Frécon. *DIVE on the Internet*. PhD thesis, University of Göteborg, 2004.

[7] C. Fröhlich and M. E. Latoschik. Incorporating the Actor Model into SCIVE on an Abstract Semantic Level. In *IEEE VR Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 61–64, 2008.

[8] C. Geiger, V. Paelke, C. Reimann, and W. Rosenbach. A framework for the structured design of VR/AR content. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 75–82, 2000.

[9] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[10] C. Just, A. Bierbaum, A. Baker, and C. Cruz-Neira. VR Juggler: A Framework for Virtual Reality Development. In *2nd Immersive Projection Technology Workshop (IPT98)*, pages 89–96, 1998.

[11] A. Kapolka, D. McGregor, and M. Capps. A Unified Component Framework for Dynamically Extensible Virtual Environments. In *Proceedings of the 4th International Conference on Collaborative Virtual Environments*, pages 64–71, 2002.

[12] R. Kuck, J. Wind, K. Riege, and M. Bogen. Improving the Avango VR/AR Framework: Lessons Learned. In *Workshop Virtuelle und Erweiterte Realität*, pages 209–220, 2008.

[13] M. E. Latoschik and R. Blach. Semantic modelling for virtual worlds a novel paradigm for realtime interactive systems? In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 17–20, 2008.

[14] M. E. Latoschik and C. Fröhlich. Towards Intelligent VR: Multi-Layered Semantic Reflection for Intelligent Virtual Environments. In *Proceedings of the International Conference on Computer Graphics Theory and Applications*, pages 249–259, 2007.

[15] M. E. Latoschik and H. Tramberend. Engineering Realtime Interactive Systems: Coupling & Cohesion of Architecture Mechanisms. In *Proceedings of the Joint Virtual Reality Conference of EuroVR–EGVE–VEC*, pages 25–28, 2010.

[16] M. E. Latoschik and H. Tramberend. Simulator X: A Scalable and Concurrent Architecture for Intelligent Realtime Interactive Systems. In *IEEE Virtual Reality Conference*, pages 171–174, 2011.

[17] E. Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.

[18] F. Mannuß, A. Hinkenjann, and J. Maiero. From Scene Graph Centered to Entity Centered Virtual Environments. In *IEEE VR Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 37–40, 2008.

[19] B. Pellens, F. Kleinermann, and O. De Troyer. An Approach Facilitating 3D/VR System Development Using Behavior Design Patterns. In *In IEEE VR Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 17–24, 2010.

[20] M. Ponder. *Component-Based Methodology and Development Framework for Virtual and Augmented Reality Systems*. PhD thesis, EPFL, Lausanne, 2004.

[21] M. Ponder, G. Papagiannakis, T. Molet, N. Magnenat-Thalmann, and D. Thalmann. VHD++ Development Framework: Towards extendible, component based VR/AR simulation engine featuring advanced virtual character technologies. In *Computer Graphics International*, pages 96–104, 2003.

[22] F. Rodrigues, R. Ferraz, M. Cabral, F. Teubl, O. Belloc, M. Kondo, M. Zuffo, and R. Lopes. Coupling virtual reality open source software using message oriented middleware. In *IEEE VR Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, 2009.

[23] C. Shaw, M. Green, J. Liang, and Y. Sun. Decoupled simulation in virtual reality with the MR toolkit. *ACM Transactions on Information Systems (TOIS)*, 11(3):287–317, 1993.

[24] A. Steed. Some Useful Abstractions for Re-Usable Virtual Environment Platforms. In *IEEE VR Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 33–36, 2008.

[25] A. Steed, D. Reiners, and M. Latoschik. Reflections on the design and implementation of virtual environment systems. *Presence: Teleoperators and Virtual Environments*, 19(2), 2010.

[26] R. M. Taylor, J. Jerald, C. VanderKnyff, J. Wendt, D. Borland, D. Marshburn, W. R. Sherman, and M. C. Whitton. Lessons about Virtual Environment Software Systems from 20 Years of VE Building. *Presence: Teleoperators and Virtual Environments*, 19(2):162–178, 2010.

[27] H. Tramberend. Avocado: A Distributed Virtual Reality Framework. In *IEEE Virtual Reality Conference*, pages 14–21, 1999.

[28] D. Wiebusch and M. E. Latoschik. Enhanced Decoupling of Components in Intelligent Realtime Interactive Systems using Ontologies. In *IEEE VR Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 43–51, 2012.

[29] C. A. Wingrave and J. J. LaViola. Reflecting on the Design and Implementation Issues of Virtual Environments. *Presence: Teleoperators and Virtual Environments*, 19(2):179–195, 2010.