Decoupling the Entity-Component-System Pattern using Semantic Traits for Reusable Realtime Interactive Systems

Dennis Wiebusch*
Universität Würzburg

Marc Erich Latoschik†
Universität Würzburg

Abstract

The Entity-Component-System (ECS) pattern has become a major design pattern used in modern architectures for Real-Time Interactive System (RIS) frameworks. The pattern decouples different aspects of a simulation like graphics, physics, or AI vertically. Its main purpose is to separate algorithms, provided by high-level tailored modules or engines, from the object structure of the low-level entities simulated by those engines. In this context, it retains advantages of object-oriented programming (OOP) like encapsulation and access control. Still, the OOP paradigm introduces coupling when it comes to the low-level implementation details, thus negatively affecting reusability of such systems.

To address these issues we propose a semantics-based approach which facilitates to escape the rigid structures imposed by OOP. Our approach introduces the concept of *semantic traits*, which enable retrospective classification of entities. The utilization of semantic traits facilitates reuse in the context of ECS-based systems by further decoupling objects from their class definition. The applicability of the approach is validated by examples from a prototypical integration into a recently developed RIS.

Index Terms: D.1.m [Software]: Programming Techniques—Miscellaneous; D.2.11 [Software]: Software Engineering—Software Architectures; D.2.13 [Software]: Software Engineering—Reusable Software

1 Introduction

Software ages [27]. Development teams change, hardware in use is replaced, necessary base software parts are updated (e.g., the underlying operating systems or system libraries), and new projects are paid more attention than previous ones.

This observation is particularly true for highly interactive systems, e.g., in the context of Virtual or Augmented Reality or game development, where maintenance and reusability often is affected by insufficient decoupling. These Real-Time Interactive Systems (RISs) are typically composed by numerous special purpose software modules for necessary aspects of a simulation, like hardware abstraction, input/Output (I/O), simulation logic, graphics, physics, or AI. Often, these modules incorporate individual data structures and control flows which have to work in concert to produce coherent simulations. As a result, RISs often become particularly prone to changes, caused by software aging.

Eventually, this leads to the replacement or rewriting of complete software components, even though only little change might be required to solve arising problems. Some major causes for this lack of maintainability are the incomprehensibility of the internal structure of existing software libraries, missing documentation, high coupling, and possibly ill-defined software interfaces. We argue that this is partly accounted for by restrictions that are imposed by the object-oriented programming (OOP) paradigm.

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

To address decoupling and hence modularization requirements, recent RIS architectures often are based on the Entity-Component-System (ECS) pattern. This pattern is strongly related to the component [26] as well as to the strategy pattern. Furthermore, in terms of its goals of code and data separation, it has many similarities to the visitor and observer pattern [11]. As one result of these relations to typical OOP patterns, ECS implementations tend to be strongly oriented towards the OOP paradigm and are often implemented using an OOP language.

Following the ECS pattern, an *entity* is partitioned into multiple *components*, each relating the entity to a certain required aspect of the simulation. Components hold the required data for these aspects, whereas the algorithms working on this data and implementing the associated aspects are located in *systems*, which are separate simulation modules (also called sub-systems or engines in RIS and game engineering).

Depending on the specific implementation, components can be fine-grained down to individual properties, like the position of an entity. In this pattern, entities merely become more or less loosely associated collections of data or properties. This allows for high modularity, since simulation modules, each addressing a particular aspect and hence using the respective component of entities, can be implemented independently from each other.

However, the dependency on the component data types leads to close coupling between sub-systems, wherefore independently developed sub-systems are likely to be incompatible to one another. In addition, implementing entities as dynamic collections of components enforces the utilization of runtime type checking and/or custom-made naming schemes, the latter of which again potentially leads to sub-system incompatibility. These issues will be discussed in more detail in the course of the paper.

We present the concept of *semantic traits*, which extends the object-oriented paradigm by a semantically augmented view of object instances. This alternative approach allows for the dynamic extension of entities at runtime. In addition, it provides the feature of semantic type checking at compile time, thus enabling the early detection of misused values. By means of *relations* between entities more expressive power is added.

In this way, semantic traits allow to overcome some of the problems arising in the context of OOP and the ECS pattern. As illustrated in the course of this work, the approach allows for subsequent classification of object instances and, hence, furthers reusability and maintainability of RISs.

The rest of this paper is structured as follows: Section 2 discusses related work in the context of RIS and software engineering research. Section 3 illustrates the restrictions imposed by the OOP approach, which then are addressed by the introduction of semantic traits in section 4. The practicability of this alternative approach is validated by an exemplary integration into an existing research framework, which is presented in section 5. The paper is concluded and an outlook of future work is given in section 6.

2 Related Work

Many frameworks that are developed in the research context incorporate an entity model [3, 10, 14, 19]. Others build upon the concept of application graphs [2, 15] while a third group favors

message-passing and event-based models [10, 12, 19]. Often no clear distinction between these approaches can be made since they are commonly mixed to create a flexible RIS framework.

The presented approach is conceptually based on an entity model, as suggested by [22, 31]. Nevertheless, it can be integrated with the above-mentioned application models, since it rather is extending the applied programming paradigm than providing a new model.

In the context of RIS frameworks object-oriented programming is the prevailing paradigm. Its inherent support for the creation of type hierarchies facilitates the modeling of virtual entities and environments. In this context, different programming patterns that facilitate the implementation of reoccurring tasks have been identified [11].

In the area of software engineering, object-oriented programming is considered an enabling technology for creating interchangeable and reusable software components [24, 30]. In this context, the application of OOP techniques often is misconceived as a sufficient means to create reusable software assets, which was found to be one of the reasons for software reuse programs to fail [25, 28, 30].

The approach proposed in this work involves the integration of a Knowledge Representation Layer (KRL). This concept is commonly found in the context of Intelligent Virtual Environments (IVEs [16, 20]). Such a KRL often is added on top of an existing application [13, 17], wherefore the access to and synchronization with the current state can be complicated. KRLs are commonly utilized, e.g., in the context of natural language processing [4], multimodal interaction [17], semantic modeling [5, 6], virtual storytelling [21], and game-like simulations [33].

Similar to [18], the approach presented in this work provides a methodology to integrate the content of the virtual environment with a KRL. Since the presented technique is meant to foster reuse of existing libraries and simulation modules, it adopts a wrapper-like nature.

3 OOP and ECS in the RIS Context

The OOP paradigm is a prominent approach to implement RIS architectures regardless of the underlying application model. At a first glance, OOP provides convenient features due to the close resemblance between OOP-based objects and inheritance as well as many commonly established RIS concepts, e.g., scene-graphs, event systems, or entity models etc. But, as illustrated in the following sections, a straight-forward modeling of conceptual objects, e.g., scene-graph nodes, events, or entities, as OOP objects and their specialization using inheritance also has its drawbacks with respect to decoupling.

3.1 Type Hierarchies and Naming Schemes

Strong type systems and encapsulation are central software engineering features, which are adopted by most OOP languages. Both are highly beneficial for an enhanced comprehensibility and flexibility of program code. Type systems also enhance run-time reliability for compiled languages since they provide favorable compile-time checks of type compatibility.

OOP languages encourage developers to represent the required categorization of the simulated conceptual objects in a type hierarchy. But in the RIS context, there often are different and sometimes even incompatible categorizations of these objects with respect to different requirements of a specific application. In addition, the tree-like hierarchical structures of type systems often are insufficient for more elaborated categorization schemes. These are, for example, necessary for entity systems, which often resemble graphs instead of trees. Thus, the type hierarchy implemented for a certain application may be inappropriate for another one. This commonly results in re-design and possibly complete re-implementation of parts of the software in question.

An object's properties as well as its functions are addressed using references in the form of variable and function names. Not knowing a reference or the associated name thus is equivalent to not being able to access the respective element. Hence, the reusability of a program depends on the developer's knowledge on the applied symbols. Since these symbols are arbitrarily chosen by the developer who is implementing the program, the enforcement of a uniform naming scheme is virtually impossible. Existing approaches, like JavaBeans (see [9, pp. 322]), do dictate certain schemes for naming accessor functions. However, since these depend on the name of the accessed property, the issue remains. The only way to realize such schemes would require an agreed on vocabulary which is strictly adhered to.

Both, a fixed type hierarchy and the arbitrarily named functions, are sources of multiple problems. On the one hand, a complex class hierarchy impedes adaption to later extensions and often results in close coupling. On the other hand, unknown or misinterpreted variable and function names can result in incorrect use of the underlying elements. Both aspects often result in partial reimplementation, either because the desired feature was not found, or because it could not be used without major adjustments.

3.2 Entity-Component-Systems

To overcome some of these issues, recent RIS frameworks, especially in the area of computer games, adopt the Entity-Component-System (ECS) pattern. In such systems, simulated objects are represented by entities, which consist of multiple components. These components represent the different properties of the entity, e.g., physical or graphical properties with respect to certain simulation aspects. (Sub-)Systems are simulation modules or engines that are responsible for a certain aspect of the simulation, for example, a physics engine or a rendering module. If an entity contains the set of components that is required for a certain system to perform simulations, it is integrated into the given sub-systems simulation loop.

Benefits of frameworks adopting the ECS pattern include high separation of concerns and the possibility to add and remove components at runtime. This is facilitated by fact that entities are *composed* of components instead of inheriting their properties. Therefore, frameworks implementing the ECS pattern are highly flexible and extensible.

Still, adopting the ECS pattern does not automatically solve all coupling problems. For example, typed OOP languages motivate to model the entities, i.e. the collections of the components, as objects and attributes from the given OOP language to benefit from type checking, preferably at compile-time (if available). However, inheritance and the rigid class hierarchies that are imposed by OOP development lead to a high coupling with the particular implementation of the system in use. Hence, the components have to exactly match the expected structure and adhere to the defined software interface. Modifications to such an interface, e.g., to change functionality (and thus components) requires modification of class hierarchies and can become complex. This is due to the fact that dependencies between sub-systems and components are resolved by investigating the type hierarchy at compile-time (as opposed to investigating the properties a component is composed of at runtime).

On the other hand, alternative implementations do model the entities and the required collections of properties as dynamic associations to also provide changes during run-time. Now, these systems have to incorporate run-time type checking to benefit from the type system available or—again—they have to restrict the dynamic aspect to setting-up all collections before compile time, e.g., using a template systems similar to the one found in C++.

In addition, allowing for a completely unguided modification of collections by adding or removing components leads to an addi-

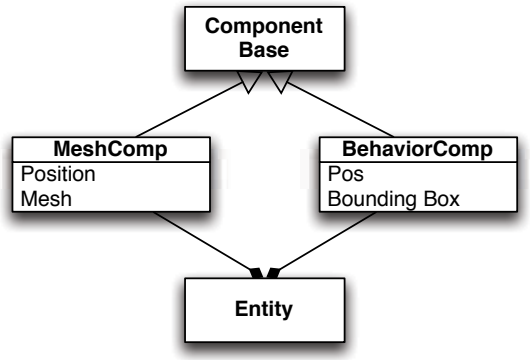

Figure 1: Example for problematic aggregation of components: Due to naming and hierarchy issues the `Position` attribute of `MeshComp` is not compatible with the `Pos` attribute of `BehaviorComp`.

tional problem: If the types of components and properties are not available any more, a way to associate components with the modules and engines is required. Naturally, in static collections this is achieved by either the type system (if available) and/or the naming schemes of the expressions (variables, access methods); in dynamic collections it requires an additional dynamic naming, usually provided by an additional registry to set-up the association between components and engines. Now, there is no compile-time check for correct naming which, in turn, can easily result in a plethora of individual naming, redundant work or even inconsistencies.

The arising issues become evident in situations in which two systems, that have been developed independently, shall be used in a single application. For example, assume that a behavior simulation system, which modifies the position of an entity, was developed together with a certain rendering system. If that behavior simulation system shall be combined with a different rendering system, both have to use compatible component types for an entity's position property (cf. figure 1). In cases where the component types are different this will require the creation of an adapter, which is used to make the component used by one system available for the other. Depending on the particular implementation, this can cause huge efforts, since the usage of the component in question may be distributed across the program.

As long as the set of used simulation systems is rarely modified, e.g., as often is true for a game engine, these issues are of lesser importance. Here, changes to functionality and hence new or modified engines and components often require a lot of development work anyways, which often results in a completely new version of the core framework and all supporting elements of the content chain. Of course, in this context it is also favorable to support reusability for commercial reasons. Compared to a research environment one can still observe that for complete game engines the developed API is (and has to be) fixed in advance, the change intervals are longer, and the development teams tend to be bigger.

3.3 Observations

In summary, the OOP paradigm introduces beneficial aspects, e.g., encapsulation. However, the strict type hierarchies that it imposes hinder reusability, since—once implemented—they cannot be adapted to the specific requirements of an application. The ECS pattern improves this situation by shifting the type hierarchy issue from entities into components. In fact, the achieved finer-grained partitioning of entities does ease the problem. Nevertheless, in order to integrate existing and newly implemented simulation modules, data types and applied naming schemes have to be known. Acquiring the necessary knowledge can be a tedious task and possibly has to be repeated for every system that is developed.

In addition, such systems do not support the programmer in retrieving certain functionality: In order to detect a method that modifies the application state in the desired way, the providing system, the method's name, and its signature have to be known.

Hence, in order to overcome above-mentioned issues and facilitate reusability, it is desirable to

- provide a central repository of applicable symbols,
- perform compile time checks based on these symbols,
- wrap existing elements in order to reuse them,
- keep beneficial aspects of OOP,
- allow to break out of fixed type hierarchies, and to
- specify the desired state instead of manually invoking the respective software function.

4 Semantic Traits

In order to address the issues that were discussed in the previous sections, we suggest the integration of a knowledge representation layer (KRL) on a core level. The proposed approach is based on the object oriented paradigm, thus allowing to utilize its beneficial aspects. Building thereon, the idea of composing entities of different components instead of inheriting their features is seized. In the next sections the elements of the KRL are introduced.

4.1 Ontological Grounding

The most basic addition on top of the object-oriented approach is the integration of a central repository of symbols to be used. While a simple dictionary would be sufficient for this intent, some features of the integrated KRL require to specify relations between the stored concepts (see below).

The compilation of such a dictionary is a complex and laborious task. Especially the agreement on the symbols used for specific concepts is difficult, since different programmers tend to use different terminology for the same object. At best, an ontology for all concepts and their relations can be established, which is used by every programmer who is working with the developed framework. Due to their ontological grounding, the symbols that identify those concepts will be called *grounded symbols*.

While some concepts share a terminology among different groups of developers, others do not. Similarly, some concepts may be irrelevant to certain developers, whereas others are shared among most of them. A practicable way to approach this issue is to partition the ontology into common and domain specific parts. The former will then be applied by every user of an eventual framework, whereas the latter can be selected to satisfy specific needs. The web ontology language (OWL [35]) is adequate for the creation of such an ontology, since it supports the intent of partitioning it into multiple files.

Using OWL to create an ontology of symbols, the issue of ambiguity between symbols can be partially solved. The language allows to declare equivalent concepts, individuals, and roles, which can be performed at the time multiple ontologies are combined to create the KRL for a particular application. However, this remains a tedious task and possible ways to automate the process are still to be investigated.

Although being well suited, an OWL ontology cannot be directly used for programming tasks. Hence, its content has to be transformed into a format that can be accessed in program code. The creation of program code from ontologies is not an uncommon approach (cf. [32, 34]) and is, for example, supported by the protégé

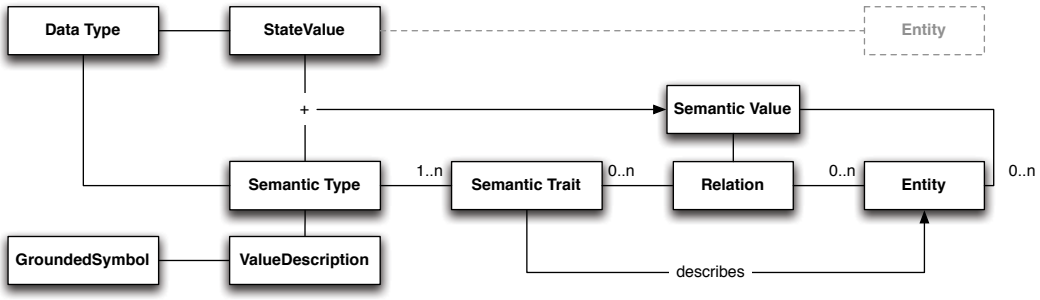

Figure 2: Conceptual overview of the semantics-based approach. The top row shows the common way of data representation in an object-oriented design. The bottom row shows the elements added by the approach presented in section 4.1 – 4.4.

OWL editor [1]. Multiple aspects are addressed by such a transformation: For one, the approach becomes independent of the utilized programming language. In addition, features of code editors like autocompletion or suggestions can be used by accumulating the generated symbols inside a dedicated namespace. Furthermore, errors due to misspelling of symbols are reduced to a minimum, since the compiler can take over the spell-checking task.

Since a grounded symbol does represent a certain concept, it is more than a simple variable identifier. Due to being taken from an ontology, in which it is put in context with other symbols, a grounded symbol does carry meaning. Thus, it can be used to assign meaning to a value by creating a semantically enriched description, which will be referred to as a *value description*.

Figure 2 provides an overview of the concepts and relations that are introduced in this and the following sections.

4.2 Semantic Values

Value descriptions are used to describe the properties of an entity, which will be referred to as *state values* below. The term state value is referring to the simulation state which can be seen as the set of all properties (implemented as components) of all simulated entities. By combining a value description with a data type, a *semantic type* is created. Moreover, the combination of a state value with an appropriate semantic type produces a *semantic value*.

To get an idea of the application of these concepts, assume the following example: A floating point value, which represents the radius of an object (e.g., a sphere), can now be assigned the value description RADIUS. This does not add valuable information for a program or the compiler, yet, but a programmer will be able to distinguish between the meanings of RADIUS and, e.g., DIAMETER.

Of course, this can also be achieved by choosing appropriate variable identifiers. However, those identifiers will only be valid in a limited scope. For example, it cannot be ensured that the identifier will stay the same when the value is passed to a method. Furthermore, a compiler does not check for semantic correctness of values behind some variable names, hence it can not detect the mismatch between the float value representing a radius and a float value representing a diameter in the example above. Using semantic values, the identifiers of the used variables become less important, since their value itself carries a meaningful symbol.

To some extent, the compiler can now be utilized to perform semantic checks by utilizing the programming language's type system: A method parameter, for example, could have the type `SemanticType[Radius]` (`Radius` being the type parameter of `SemanticType`), thus allowing only semantic values with this exact value description to be passed.

4.3 Semantic Traits

Whereas semantic values help to integrate semantics into a programming language, the concept itself is quite limited. The logical next step is to allow the creation of more complex descriptions from existing ones; an idea being very similar to the inheritance concept found in object-oriented languages.

A central concept, that is exploited in combining OOP techniques with the proposed approach, is related to the idea of traits [7]. Developed as a means to increase reusability by facilitating multiple inheritance, traits allow for a more finely granulated modularization of a class's functionality.

We introduce the concept of *semantic traits*, which allow to combine multiple value descriptions, in turn creating a new kind of value description. However, the values described by semantic traits are different to those described by value descriptions: Whereas a value description is meant to entirely describe the value it is associated with, a semantic trait does rather constitute a set of requirements to be fulfilled by the described value.

For simplicity, we assume that no conversion between representations is required (e.g., all simulation modules apply a left-handed coordinate system, etc.). However, an automatic type conversion mechanism as presented in [36] could be integrated into the concept by specifying a common ground for types, too.

The idea of *semantic traits* aims at decomposing objects into their (semantic) properties. It conceptually inherits from ECS components and extends them with a trait-like nature, mixing-in value descriptions. For example, a virtual object could be decomposed into a color trait and a shape trait. This does not include part-of (or other) relations directly: a `wheel` object is not a semantic trait of a `car` object. However, the car trait could specify the requirement of a part-of relation between a car object and a wheel object.

4.4 Relations

In order to represent such and other associations, the concept of a *relation* is introduced. Relations are used to link a semantic trait or semantic value to an entity or another semantic value. In the car and wheel example from the previous section the two semantic values `wheel` and `car` would be connected by a PARTOF relation.

As shown in figure 3, a relation is a semantic value itself. This is a reasonable representation, since all interface elements are meant to be represented by semantic values, and a relation might belong to the described entity's interface to the application. Accordingly, the associated grounded symbol is the name of the relation inside the ontology (assigned via the associated relation description).

In addition to relations, figure 3 shows two further predefined classes: `SemanticEntity` and `ValueDescription`. The requirement to represent entities inside an application is met by the in-

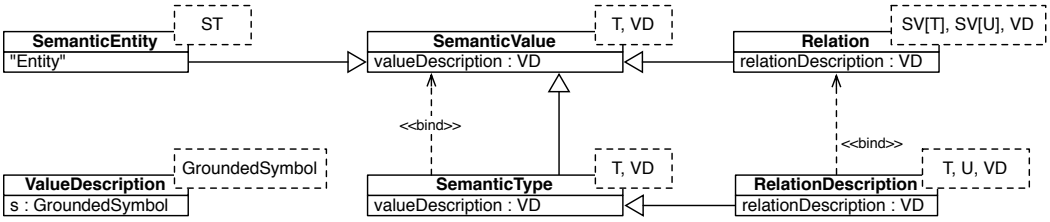

Figure 3: Overview of the defined concepts: Relations, semantic types and semantic entities are instances of semantic values. Each value description is associated a semantic value. Both relations and semantic type instances are created by means of associated descriptions. The used abbreviations for type parameters are VD = ValueDescription, ST = SemanticType, SV = SemanticValue. T and U represent arbitrary data types.

troduction of the `SemanticEntity` class. Each semantic entity is an instance of the `SemanticValue` class that is described by an `EntityDescription` (being a specific `SemanticType`). As opposed to common OOP-based approaches, entities of different types do not create a class hierarchy in program code. Their interconnections and assertions of attributes are achieved by the use of relations and semantic values.

At first it might seem counterintuitive to derive the class `SemanticType` from the `SemanticValue` class, since the first is conceptually used to describe the latter. However, creating a hierarchy like this does enable very flexible definitions, as shown in the following examples.

Assume that the concepts ENTITY and COLOR as well as the relations HAS and HASVALUE are defined in the ontology. The automatic process of transforming the ontology contents into program code will then generate the relation descriptions `has` and `hasValue` as well as the value descriptions `Entity` and `Color`. A new semantic value representing the entity can then be created by combining the generated value description with an entity object. The same holds true for the color description and its possible combination with three floating point numbers. In this way, the elements of the underlying framework are wrapped by instances of the `SemanticValue` class and can be linked by means of the `hasValue` relation.

But the approach allows also to express the fact that the entity has a color, without specifying its value. To do so, an entity and the semantic type color are linked by the `has` relation. The fact that each relation in turn is a semantic value allows to specify values for relations: The entity-has-color relation can, e.g., be linked with a semantic value containing color.

4.5 Methods

So far, the semantics-based approach only allows for the description of entities, their attributes, and relations between them. However, every program also consists of methods that (especially in the object-oriented case) are often tailored for specific objects.

The description of such functions is especially relevant in the planning domain. Different approaches have been proposed [8, 23, 29], most of which focus on the description of preconditions and (side-)effects of the respective method. In order to be able to benefit from previous achievements in this field, the presented approach adopts this representation.

With regard to entities, semantic traits can be used to specify both preconditions and effects of a method. Since a semantic trait captures a (possibly partial) view on an entity but also allows to describe single values, all parameters of a method can be specified in the form of such traits. This way of describing parameters also captures the need to specify preconditions which have to be met before executing the respective method.

Similarly, effects of a function can be described using semantic traits. While the description of preconditions is for the most part achieved by means of the method's parameters, effects have to be stored externally. In the end, two sets are created for each method, one containing the semantic traits representing its preconditions and one representing its effects (on the entities in the parameter list).

In contrast to an object-oriented approach, a method would not be part of an entity, but an object itself. This approach is also found in program languages adopting the functional paradigm, wherefore opting for a programming language that supports the functional paradigm is favorable for a possible realization.

The implementation of a particular method, which normally would be part of a class, can be integrated into a semantic trait. In that case, the semantic trait represents the view on the entity which supports the specific method. E.g., a `Vehicle` semantic trait could be accompanied by a method `moveTo(location)`. It has to be ensured that every entity that matches this trait is compatible to the method in the same way that systems are matched against a set of components in ECS-based frameworks. Then the `Vehicle` trait allows to reuse the `moveTo` method with every compatible entity.

Finally, the semantic traits describing (parts of) a method's preconditions and effects can be used to allow for the automated retrieval and invocation of methods. On a small scale, this even could enable the automatic combination of methods to create a more specific one, assuming an elaborated description of entities and methods.

4.6 Annotations

The above-mentioned building blocks are sufficient to overcome some of the issues that were found with OOP-based designs. However, a programmer might feel restricted being forced to use a limited set of symbols. Since this set should not be extended as rashly as new variables are defined but requires the extension of the underlying ontology, its application will possibly be perceived as hindering.

To allow more specific descriptions of objects without creating specific symbols for every possible case, the `hasAnnotation` relation is introduced. Annotations basically are semantic values that are related to another semantic value using this relation. This feature can be used to specify more details about the annotated value than the amount that is given by the associated grounded symbol. For example, a value could be added a timestamp by annotating it with an appropriate semantic value.

4.7 Discussion

As with ECS-based frameworks, entities in the presented approach are not required to be structured in a class hierarchy. The reason for this is the fact that the associated semantic value, and not the entity itself, does contain its description. Hence, the class affiliation of an

```
// semantic types
case object Location extends
     SemanticType(classOf[String],
     ValueDescription(symbols.shape))
case object Position extends
     SemanticType(classOf[ConstVec3],
     ValueDescription(symbols.position))
case object Shape extends SemanticType(classOf[String],
     ValueDescription(symbols.shape))
case object SteeringBehavior extends
     SemanticType(classOf[Boolean],
     ValueDescription(symbols.steering))

// relations
case object has extends
     RelationDescription(classOf[Entity.DataType],
     classOf[Any], symbols.has)

// semantic traits
object Wheel extends SemanticTrait
      [Position.Type with Scale.Type]
      (has(Shape("round")))

object Vehicle extends
     SpecificSemanticTrait[Position.Type](
     has(SteeringBehavior), has( _ >= 4, Wheel) )
{
  final type SpecificEntityType = SemanticEntity[Type]{
    def moveTo(s : Location.SemanticType)
  }

  protected def createEntity(e: Entity.SemanticType) =
    new SemanticEntity[Type](e) {
      def moveTo(s : Location.SemanticType) =
        println(entity + " is now in/at " +
          (entity attain has(s)))
  }
}
```

Listing 1: Implementation of elements used in the examples shown in listings 2 and 3.

entity can be detected by matching that semantic value against a semantic trait. This implies that an object that was not designed to be an instance of a specific class in the first place can become such by being added the missing semantic values and relations at runtime. On the other hand, a new semantic trait can match preexisting entities when the required semantic values and relations are matched. This way, the approach extends the ECS pattern, since the naming and hierarchy issues with components can be resolved.

5 Prototypical Integration

The approach described above was implemented into a currently developed research framework [19]. A beneficial aspect at this point is the fact, that the used framework builds on an entity model. Therefore, adjustments are limited to the implementation of the above mentioned concepts, which are wrapped around existing architecture elements.

5.1 Relations, Semantic Values, and Semantic Traits

Listing 1 exemplifies the implementation of semantic types, relations, and semantic traits which will be used for the subsequent examples in listings 2 and 3. The given code corresponds to the example mentioned above: A semantic trait named `Vehicle` is specified to have a position, a `SteeringBehavior`, and at least four wheels. A wheel in turn is specified to have an arbitrary value for the semantic values position and scale and to have the value "round" for its shape value.

```
object SetLocationAction extends Action(
  parameters = List[Action.Param](Vehicle, Location),
  returnType = Location,
  preconditions = Set(),
  effects = Set(0 -> has -> 1)
) {
  def apply(parameters: Array[Value[_]]) = {
    println("Setting location of " +
       parameters(0) + " to " + parameters(1))
    parameters(0) set has(parameters(1))
    Location(parameters(1))
  }
}
```

Listing 2: Exemplary implementation of an `Action`. The integer values in the effect definition represent indices in the parameter list.

The `has` relation that is used in the example connects semantic values containing entities with arbitrary other semantic values. In this context, the semantic types `Location`, `Position`, `Shape`, and `SteeringBehavior` are used.

The presented implementation makes heavy use of type parameters to allow for type checking at compile time. For example, the semantic trait `Wheel` is specified to involve a position value in this way. Therefore, eventual functions can be defined to only accept values of type `SemanticTrait[Position.Type with Scale.Type]` or `Wheel.Type`. This feature is facilitated by the programming language in use (Scala). Depending of the respective language used for an implementation this might be harder to achieve. Nevertheless, the advantages of the use of semantic traits remains, even if compile time support can not be fully realized.

While the `Wheel` trait is a simple semantic trait, the `Vehicle` trait creates a wrapper class that provides a `moveTo` method. The latter is implemented to print out a string, containing the entity and the result of an invocation of its `attain` method. The attain method is implemented by each semantic value. Its single parameter is a list of semantic values, which specify the desired effects. In this case, the entity shall be involved in the relation `has(s)`, `s` being the location that was passed to the `moveTo` method.

Note that the type of the `moveTo` method's parameter is specified as `Location.SemanticType` (which is the type of the semantic values that is instantiated by the semantic type `Location`). Hence, even though the data type of a location was defined to be an instance of the `String` class in listing 1, only semantic values that were created using the semantic type `Location` can be passed to the `moveTo` method.

5.2 Methods

The above-mentioned attain function automatically searches for a method that can fulfill the specified requirements. For this purpose, a repository containing registered `Actions` is utilized.

In listing 2 an exemplary implementation of such an action is shown. As mentioned in section 4.5, these are defined by means of their parameters, preconditions and effects. In this case, the parameters include the `Vehicle` trait and a `Location`. Each action is automatically registered on creation, wherefore it can be retrieved by specifying the current as well as the desired state (i.e. specifying preconditions and effects).

This lookup functionality can either be manually implemented or be performed by a planning component (e.g., a PDDL [23] planner). In the latter case, plans consisting of multiple, sequentially executed actions can be automatically retrieved and executed. The connection to such planner is straight forward, since the proposed approach already uses symbols for most elements. A layer that abstracts from concrete values by assigning each distinct value a

```
1  val carEntity = createEntity()
2  val wheels = for (i <- 1 to 4) yield createEntity()

4  wheels.foreach{ wheel =>
5    wheel set has(Shape("round"))
6    wheel set Position(Vec3f.Zero)
7    wheel set Scale(0.5f)
8    carEntity set has(wheel)
9  }

11 carEntity set has(SteeringBehavior)
12 carEntity set Position(Vec3f.Zero)

14 carEntity as Vehicle moveTo Location("London")
15 carEntity attain has(Location("Paris"))

17 println("car is at " + (carEntity get has(Location)))
```

Listing 3: Example usage of the implemented elements.

unique symbol can be used to address the issue of a planner's limited support for the actual values.

The action that is implemented in the example from listing 2 prints a string to the console and returns the location it was provided in the parameter list.

5.3 Usage

The usage of the elements from listings 1 and 2 is exemplified in listing 3: Five (empty) entities are created in lines 1 and 2. Four of these are assigned the relation `has(Shape("round"))`, a position, and a scale. Each of them is added to the remaining fifth entity, which gets assigned the relation `has(SteeringBehavior)` and also is added a position.

For the sake of brevity the assigned values are all the same, in a real application at least the position would be different for all entities. In the same way, only the `has` relations is utilized, whereas `hasPart` and `isAt` relations could be used to create more specific content.

Lines 14 and 15 of listing 3 show two ways in which the defined semantic traits can be used: The entity either can be wrapped by the `Vehicle` trait, whereby the moveTo method becomes available. Alternatively, the entity's `attain` method can be invoked, passing the desired state.

In the first case, the entity is checked for the required features at the time the wrapping occurs. If the requirements are not met an exception is thrown, wherefore appropriate exception handling is necessary. Using the `attain` method the verification is implicitly performed when matching methods are retrieved. If no matching method is found the value `None` is returned.

The first method allows for more control over the methods that are eventually invoked. In this example, the `moveTo` method from listing 1 invokes the `attain` method, but a more concrete implementation would remove the thus introduced uncertainty. The second is more flexible and possibly allows to reuse methods without knowing their exact signature or implementation. Finally, the value of the entity's `Location` value is accessed in line 17 of listing 3.

6 Conclusion & Future Work

By applying the presented techniques, it is possible to break out of the strict class hierarchy imposed in an OOP context. Similar to ECS-based applications, this is achieved by composing entities of multiple components. In contrast to the ECS pattern, the proposed approach allows to detect an entity's class affiliation at runtime. Using the concept of semantic traits, an entity can be augmented with further functionality, according to the detected components. Thus, an object that did not comply with a certain semantic trait in the first place can be modified at runtime and then be assigned additional functionality.

Furthermore, the utilization of semantic values allows for semantically enhanced type checking at compile time. This prohibits the misuse of values with the same data type but different meaning, a source of programming errors that are hard to detect.

Depending on the programming language in use even more semantic checks can be performed at compile time: The opportunity to specify type parameters to be covariant and dynamically specify the type parameters of return types allows to reduce the required runtime checks to the verification of required values and relations.

The utilization of a coherent vocabulary, grounded in an OWL ontology, fosters reusability of developed programs. Not only does the understandability of program code increase but also does the support for retrieving classes and functions. In addition, this allows for the integration of symbol-based artificial intelligence methods, e.g., reasoning and planning modules.

The opportunity to register functions by specifying their preconditions and effects in terms of semantic traits allows for their automatic invocation. In this way, the amount of required knowledge about the method's signature is rendered unnecessary.

Future work will include benchmarking the overhead introduced by the required runtime type checks. In addition, the integration of a reasoning component to allow for more complex requests to the application state remains an open topic. Finally, the implementation of a more complex scenario has to be accomplished to test and extend the integration of a planning component.

References

[1] protégé. `http://protege.stanford.edu`. Last access: 2015-01-22.

[2] 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.

[3] S. Bilas. A data-driven game object system. In *Game Developers Conference Proceedings*, 2002.

[4] M. Cavazza and I. Palmer. High-level interpretation in virtual environments. *Applied Artificial Intelligence*, 14(1):125–144, 2000.

[5] 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.

[6] K. Coninx, O. De Troyer, C. Raymaekers, and F. Kleinermann. Vr-demo: a tool-supported approach facilitating flexible development of virtual environments using conceptual modelling. *Proc. of Virtual Concept 2006*, 2006.

[7] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.

[8] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3):189–208, 1972.

[9] D. Flanagan. *Java in a Nutshell*, page 320 ff. O'Reilly Media, Inc., 2005.

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

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[12] 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.

[13] E. Kalogerakis, S. Christodoulakis, and N. Moumoutzis. Coupling Ontologies with Graphics Content for Knowledge Driven Visualization. In *Virtual Reality Conference*, pages 43–50. IEEE, 2006.

[14] 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.

[15] 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.

[16] M. Latoschik. Semantic reflection–knowledge based design of intelligent simulation environments. *KI 2007: Advances in Artificial Intelligence*, pages 481–484, 2007.

[17] M. Latoschik, P. Biermann, and I. Wachsmuth. Knowledge in the loop: Semantics representation for multimodal simulative environments. In *Smart Graphics*, pages 924–924. Springer, 2005.

[18] 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.

[19] 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.

[20] M. Luck and R. Aylett. Applying Artificial Intelligence to Virtual Reality: Intelligent Virtual Environments. *Applied Artificial Intelligence*, 14(1):3–32, 2000.

[21] J.-L. Lugrin and M. Cavazza. Ai-based world behaviour for emergent narratives. In *Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology*, page 25. ACM, 2006.

[22] 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.

[23] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl — the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[24] H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *Software Engineering, IEEE Transactions on*, 21(6):528–562, 1995.

[25] M. Morisio, M. Ezran, and C. Tully. Success and failure factors in software reuse. *Software Engineering, IEEE Transactions on*, 28(4):340–357, 2002.

[26] R. Nystrom. *Game programming patterns*. Genever Benning, 2014.

[27] D. L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[28] J. Sametinger. *Software engineering with reusable components*. Springer, 1997.

[29] W. Schuler, L. Zhao, and M. Palmer. Parameterized action representation for virtual human agents. *Embodied conversational agents*, page 256, 2000.

[30] K. Sherif and A. Vinze. Barriers to adoption of software reuse: a qualitative study. *Information & Management*, 41(2):159–175, 2003.

[31] 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.

[32] G. Stevenson and S. Dobson. Sapphire: Generating Java Runtime Artefacts from OWL Ontologies. In *Proceedings of the 3rd International Workshop on Ontology-Driven Information Systems Engineering*, 2011.

[33] T. Tutenel, R. Bidarra, R. Smelik, and K. Kraker. The role of semantics in games and simulations. *Computers in Entertainment (CIE)*, 6(4):57, 2008.

[34] M. Völkel and Y. Sure. RDFReactor-From Ontologies to Programmatic Data Access. In *Poster Proceedings of the 4th International Semantic Web Conference*, page 55, 2005.

[35] W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview. Technical report, W3C, October 2009.

[36] 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.