Semantic Entity-Component State Management Techniques to Enhance Software Quality for Multimodal VR-Systems

Martin Fischbach, Dennis Wiebusch, *Member, IEEE*, and Marc Erich Latoschik, *Member, IEEE*

Fig. 1. Multimodal interface demonstrations realized using the presented techniques: (left) an immersive multimodal game [11], (center) a multimodal mixed reality strategy game developed within student projects [25], and (right) a result of a master level course [43].

Abstract—Modularity, modifiability, reusability, and API usability are important software qualities that determine the maintainability of software architectures. Virtual, Augmented, and Mixed Reality (VR, AR, MR) systems, modern computer games, as well as interactive human-robot systems often include various dedicated input-, output-, and processing subsystems. These subsystems collectively maintain a real-time simulation of a coherent application state. The resulting interdependencies between individual state representations, mutual state access, overall synchronization, and flow of control implies a conceptual close coupling whereas software quality asks for a decoupling to develop maintainable solutions. This article presents five semantics-based software techniques that address this contradiction: Semantic grounding, code from semantics, grounded actions, semantic queries, and decoupling by semantics. These techniques are applied to extend the well-established entity-component-system (ECS) pattern to overcome some of this pattern's deficits with respect to the implied state access. A walk-through of central implementation aspects of a multimodal (speech and gesture) VR-interface is used to highlight the techniques' benefits. This use-case is chosen as a prototypical example of complex architectures with multiple interacting subsystems found in many VR, AR and MR architectures. Finally, implementation hints are given, lessons learned regarding maintainability pointed-out, and performance implications discussed.

Index Terms—Real-time interactive systems, virtual reality systems, software architecture, multimodal processing

1 Introduction

Software architectures for Real-Time Interactive Systems (RIS) often consist of multiple subsystems for graphics, physics, sound, input, Artificial Intelligence (AI), and many more. They collectively fulfill the necessary functional requirements of todays applications of Virtual, Augmented and Mixed Reality (VR, AR, and MR), computer games, and also robotics. Subsystems commonly include specific data structures for performance reasons but also exhibit a close semantic and temporal coupling between each other for consistency reasons. In contrast, non-functional software quality requirements like reusability, modifiability, and low development effort motivate decoupling, a contradiction known for some time as the *coupling dilemma* [23].

The Entity-Component-System (ECS) pattern [1] has become a prominent approach to the coupling dilemma (e.g., [16, 18, 37]). This pattern organizes the data (the components) associated with subsystems (the systems) in an object-centered view (the entities) using composition over inheritance. This composition greatly enhances decoupling. Problems arise in cases where subsystems need mutual access to components outside of their primary data association. Typical examples are AI subsystems incorporating data models which reflect the overall application state, e.g., to provide inference capabilities or semantic grounding, and subsystems for multimodal interfaces (MMI) as promising alternative interaction techniques for RIS applications [2, 5, 21].

Specific MMI frameworks, like [2, 5, 14, 24, 34, 36, 39], usually need real-time state access to information ranging from raw sensor input at the data level to semantic information of a given context at the decision level. At the same time, they have to be decoupled from the main simulation loop to not compromise the overall simulation. Hence, the increased state access and decoupling requirements of the MMI-RIS combination are particularly suited as use-cases for developing improved solutions to the challenging software quality problem [10, 21, 22] of similarly complex RIS architectures.

This article presents five semantics-based techniques that extend the well-established ECS pattern and explicitly target maintainability, i.e. modularity, modifiability, reusability, and API usability [31] for VR, AR and MR-systems. The techniques provide a unified access scheme to the application state and facilitate the integration of symbolic AI methods as illustrated for decision level multimodal processing.

2 Example Interactions

Throughout this paper we will use a typical instruction-based speech and gesture interaction example to identify the resulting requirements and, subsequently, point out the benefits of the presented approach: A user furnishes a virtual room in an architectural modeling application. At first she utters "*Put* [deictic gesture] *that green chair near* [deictic gesture] *this table*." (cf. [3]) followed by "*Turn it* [kinemimic gesture] *this way*." (cf. [20]).

For the realization of both examples, access to representations of the user, her communicative utterances, e.g., posture and spoken words,

- *Martin Fischbach is with the HCI chair at the University of Würzburg. E-mail: e-mail: martin.fischbach@uni-wuerzburg.de.*
- *Dennis Wiebusch is with the Department of Applied Cognitive Psychology at Ulm University. E-mail: dennis.wiebusch@uni-ulm.de.*
- *Marc Erich Latoschik is head of the HCI chair at the University of Würzburg. E-mail: marc.latoschik@uni-wuerzburg.de.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

and the virtual environment in the application state is required (requirements R_{acc} and R_{rep}). The virtual environment comprises objects like the table and the chair, including properties like positions, orientations, velocities, textures, and bounding boxes. Sensor data is typically provided in different abstraction levels by drivers or SDKs, e.g., positions and orientations from a tracking system or a text representation of spoken words from an Automatic Speech Recognizer (ASR). It will be integrated into the application state by input subsystems. Some of it may have to be accessed (R_{acc}) and processed (R_{proc}) further before it can be used for decision-level multimodal fusion, e.g., by determining meaningful gestures from the users positions and orientations over time or by annotating text tokens with their lexical category. The result of this processing is typically event-like symbolic data.

During multimodal fusion, this data has to be checked if it satisfies syntactic, temporal, and semantic constraints. Syntactic correctness involves checking of allowed successions, e.g., if a verb is followed by a noun phrase. Temporal correctness involves checking of co-occurrence between multimodal channels as in "[deictic gesture] *that*". Semantic correctness involves checking if analyzed expressions make sense in the scope of the application, e.g., if the pointing gesture really denotes a green chair or if the chair is movable.

Finally, a successfully analyzed utterance has to result in an appropriate reaction that is executed and perceivable by the user (R_{exe}). For example, the chair should appear next to the table or rotate as long as the user performs the respective gesture with her hand. In addition to deriving and executing an instruction, it is desirable to provide feedback to the user at the time of processing. This could include presenting intermediate results, like highlighting the green chair after the first part of the first example has been uttered. In order to successfully process the second example, two additional aspects have to be considered. Firstly, former utterances or rather the past discourse has to be represented, e.g., the fact that the green chair has been selected in the first utterance, in order to resolve the anaphora *it*. Secondly, the realization of the kinemimic rotation gesture requires a continuous mapping of posture features to a virtual object, i.e. the chair's orientation. Ultimately, all these functional requirements have to be fulfilled with respect to non-functional RIS performance requirements (R_{per}), including low latency between user actions and system reactions as well as overall data throughput.

3 Related Work

Starting with Bolt's *"Put-That-There"* [3] in 1980, software realizing a multimodal interface was comprised of a fixed set of subsystems assembled for a dedicated application. These *demonstrations* showcased that it is feasible to achieve the processing and performance required for such interfaces. Around the millennium, non-functional software qualities like reusability, modifiability, and modularity as well as the ease of development started to gain importance. This lead to *domain dependent frameworks*, suitable for the implementation of various applications in specific domains, e.g., desktop or mobile applications, interactive surfaces, virtual environments, or robot control. These frameworks used underlying subsystems to realize domain specific features, like application logic, simulation, or rendering, and were tailored to the feature and performance requirements defined by the domain. *Independent frameworks*, on the other hand, further strengthened the ideas of reusability, modifiability, and modularity: they provide multimodal processing capabilities without dedication to a specific application area (see Table 1 column 1–2).

With respect to its applicability for RIS, a framework either can be independent or has to be integrated into a RIS. An independent framework, like [7, 14, 24, 28, 34–36] or [39], depicted in the left part of Fig. 2, will typically provide means for representing data (**A**), integrating sensor data, as well as processing and fusing data from different modalities. The communication of obtained results, however, is left to the application developer and typically achieved by adding a dedicated "processing subsystem" to the framework, e.g., by using sockets. Similarly, a RIS (Fig. 2 right) will provide features like means for representing data (**B**), input integration and input processing (e.g., for head tracking), as well as simulation and rendering subsystems. However,

Table 1. Multimodal processing frameworks categorized into demonstrations (demo), domain dependent frameworks (dom.), and independent frameworks (ind.). The third column indicates if the framework explicitly supports RIS applications, subdivided in support for virtual environments (VE) and robot control (RC). The forth column indicates if the framework is available for research, i.e. if the source code can be obtained and if it is running on current hardware platforms (*refer to the text for details).

Name	Type	Explicit RIS support	Available
Put that There [3]	demo	no	no
Cubricon [30]	demo	no	no
eXpert TRAnslator [40]	demo	no	no
ICONIC [15]	demo	yes (VE)	no
QuickSet [6]	dom.	no	no
SGIM & virtuelle Werkstatt [19, 21]	dom.	yes (VE)	no*
ICARE/FACET [4]	dom.	yes	no
OpenInterface [34]	ind.	-	yes
SKEMMI (OpenInterface) [24]	ind.	-	yes
Meanings4Fusion (OpenInterface) [28]	ind.	-	yes
HephaisTK [7]	ind.	-	yes
i*Chameleon [36]	ind.	-	no
Mudra [14]	ind.	-	no
unnamed framework using COLD [2]	dom.	yes (RC)	no
HCI^2 [35]	ind.	-	yes
SSI [39]	ind.	-	yes
M3I [29]	dom.	no	yes
Simulator X/miPro [22]	dom.	yes (VE)	yes
unnamed framework [5]	dom.	yes (RC)	no

the underlying data models most likely will not be the same. Thus a conversion $\mathbf{A} \rightarrow \mathbf{B}$ is necessary. In addition, some semantic constraint checks require access to properties of the virtual environment, like resolution of a pointing gesture. They can either be performed within the RIS, if the communicated result contains additional parameters for later resolution (e.g., pointing ray and timestamp), or within the multimodal framework, if required properties are communicated ($\mathbf{A} \leftarrow \mathbf{B}$). Requirements like continuous mapping of posture features exacerbate the communication needs. Moreover, some requirements necessitate adaptations in both systems, like semantic constraint checks and feedback at the time of processing. This decreases coherence and thus maintainability. The same holds true for features that can be realized within both systems (dashed rectangles in Fig. 2). For instance, a dialog management subsystem can be implemented within the multimodal framework, if respective wildcards for not yet resolvable referents are communicated or if relevant application state properties are synchronized. It can also be implemented within the RIS, e.g., to increase coherence if the system supports virtual agents. Altogether, the access

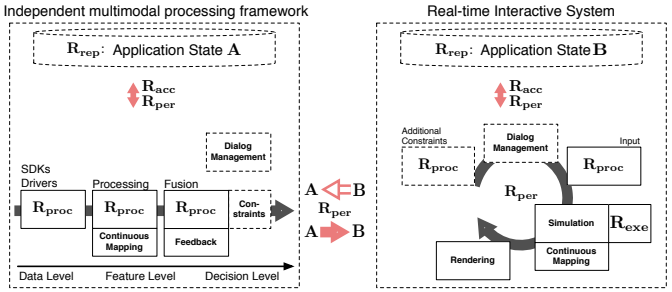

Fig. 2. Expectable architecture and requirements for the utilization of an independent multimodal processing framework in combination with a RIS. Refer to the text for details.

and synchronization of application states (red arrows in Fig. 2) is critical in terms of maintainability and performance. Framework-internal access is least problematic in this case. Synchronization, however, is a potential source for ad-hoc implementations with poor maintainability (due to required conversion) and low performance (due to required serialization).

RIS dependent multimodal frameworks [4, 19, 21] usually provide a uniform access scheme to the application state. However, such frameworks depend on the maintainability of the underlying RIS, which often suffers from the coupling dilemma due to inherently complex system architectures. In the case of Latoschik's work, the integration of a multimodal processing framework using temporal Augmented Transition Networks (tATNs) as well as the Multimodal Integration Markup Language (MIML) [20] into the RIS *AVANGO* led to groundbreaking results, like the *virtuelle Werkstatt* and the multimodal interface to the virtual agent *Max* [21]. However, the heavy dependency on AVANGO, its initial underlying proprietary scene graph system (SGI's Performer), and the utilized scripting language, hindered long-lasting reusability. For example, extensive close coupling by inheritance prevented easy adoption of other rendering systems, e.g., OpenSceneGraph or Guacamole as used by AVANGO's successors. Even worse, it prohibited researchers to build upon these results since the *virtuelle Werkstatt* has no currently running build or successor; a common issue especially in the RIS area (see Table 1 column 3-4).

Close coupling caused by inheritance is a well-known problem by now. In general, it is understood that decoupling will favor composition over inheritance. Too often, the latter has been a primary method of software architectures based on the object-oriented paradigm. As a result, the ECS pattern [1] has more and more been used in RIS architectures [16, 18, 37] since it fosters low coupling and hence increases maintainability [41, 45]. However, it still possesses some deficits: Component type information is not necessarily available at runtime and components from different applications are mostly incompatible with each other. More importantly, in application areas that heavily depend on symbolic AI methods, like multimodal processing, the ECS pattern's object-centered data model and access is detrimental, since subsystems typically need mutual access to components outside of their primary data association.

4 Semantics-based Software Techniques

This section presents five semantics-based software techniques that improve and extend the ECS pattern to facilitate state representation and access, foster modularity, and enhance overall maintainability. Each subsection is structured as follows: First, the generalized concept of the technique is presented. Afterwards, the technique is showcased on the basis of the example interactions using code samples from a reference implementation. Finally resulting benefits are discussed. The discussion focuses on modularity, modifiability, and reusability, arising from application of the technique to the architecture of a RIS, as well as on the API usability of the presented code samples.

The ECS pattern is central for structuring content and logic: every object that is meaningful to the application, ranging from rendered and simulated objects to input devices or the user(s), is represented as an entity. Entities consist of components (i.e., sets of properties) that are altered and observed by subsystems. The reference implementation is realized using the open source research platform Simulator X [18] and its integrated multimodal processing framework miPro [22].

4.1 Semantic Grounding

The first technique is a grounding mechanism for elementary system and application properties (R_{rep}) at the core level of a platform [44]. Instead of direct variable access or using mutator methods, the technique applies separately defined tokens to access properties (R_{acc}). Those tokens, called *grounded symbol*s, reflect the property's meaning and facilitate an explicit common ground for identifiers, as an improvement over agreeing on variable- and function names in an API definition. As opposed to such techniques, grounded symbols are intended to be defined externally (see Sect. 4.2) and thus provide the opportunity to be used in multiple, possibly unrelated applications. In order to identify

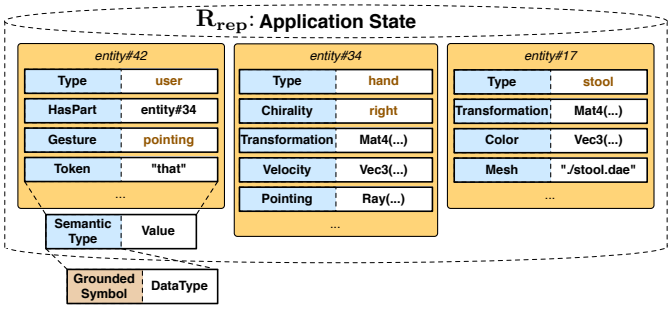

Fig. 3. Grounded symbols (brown), semantic types (blue), semantic values (blue), and entities (orange) are utilized to represent the application state. A potential modeling for the environment of our example is shown. In addition to properties based on common data types, grounded symbols are used as entity property values denoting high-level symbolic values, like `pointing`, and entity references are used to denote relations, cf. `HasPart`. Sensory input is integrated as follows: a VRPN subsystem creates and updates entities each representing one tracked joint (e.g., *entity#34*), comprised of at least one `Transformation` property. Another subsystem wrapping an ASR continuously updates the user entity (*entity#42*) with a property representing the last word spoken.

and query the value of an entity property, a so called *semantic type* is utilized. A semantic type is a tuple, consisting of a grounded symbol and a data type, that allows to create meaningful types. For example, a radius property can be distinguished from a diameter property at compile time (cf. [41, p. 107]), even if both are represented by floating point values. Finally, an entity property is a triple consisting of a grounded symbol, a data type, and a concrete value of that type. Such a triple is called a *semantic value* and can be added to and removed from entities at runtime.

Fig. 3 exemplifies the technique by means of our running example while the API for accessing entity properties is illustrated in Listing 1.

Altogether, this explicit modeling of semantics at the core level centralizes the agreement on identifiers and their meaning and decouples it from the underlying data model. Thus, the technique links the ECS pattern and the concept of semantic entities [21]. Furthermore, it facilitates the reflection of entity properties even if the target programming language does not support that level of introspection. This way, the ECS pattern's lack of component type information at runtime is addressed and the rigidity of components is countered by facilitating access to entity properties. Extensive use of callback functions to handle accessed values enables the interface to be used in synchronous as well as in asynchronous environments, as eventual invocations can occur instantly or may be delayed. Depending on the implementation, this requires utilizing synchronization mechanisms to counter race conditions, which are inherent to multi-threaded asynchronous state access.

Two additional design decisions underlie the code sample in Listing 1: the utilization of the actor model [13] and the use of the Scala programming language. On basic levels, requesting entity properties requires to exchange messages between actors. As a consequence the requested `Transformation` property can not be handled until a respective answer is received. The API reflects this by requiring a handler function that is called as soon as the property is available. Scala's

```
1 def setPointingRay(hand: Entity) {
2   if(hand.has(Transformation)) {
3     hand.get(Transformation){ t =>
4       hand.set(
5         Pointing(new Ray(t.origin, t.zAxis))
6       )
7 }}}
```

Listing 1. Definition of a function that retrieves the `Transformation` property of an entity (line 3), extracts its origin and z-axis (line 5), and uses it to create and set a `Pointing` property (lines 4–6).

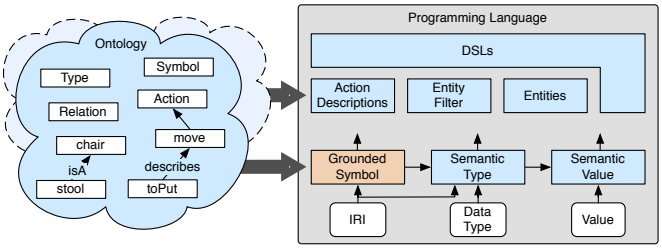

Fig. 4. Conceptual overview of the generation technique. Besides grounded symbols, semantic types, and -values, also higher-level concepts are generated: semantic entities, action descriptions, semantic queries, and finally Domain Specific Languages (DSLs) use the lower-level primitives as building blocks. In the context of the example environment, domain knowledge modeled in OWL can be utilized at runtime, like the fact that a stool is a chair or that the verb *to put* can denote an instruction to move something somewhere.

functional aspects help maintaining concise code by allowing an inline definition using a lambda expression (lines 3–6) or alternatively by passing a function (a first class citizen in Scala).

4.2 Code from Semantics

The second essential technique is the use of an external ontology in combination with a code generation approach: grounded symbols, semantic types, and semantic values are transformed from an external ontology into program code of a target programming language, as suggested in [41, 44]. The generated primitives contain a reference to the associated external concept, facilitating lookups and utilization of ontology content at runtime. If this transformation was performed manually, this technique would correspond to specifying respective function- and variable names in software interfaces. Instead, we propose to run an automated task that is integrated into the development tool-chain to generate said first class citizens.

Fig. 4 illustrates this transformation. An exemplary definition of a grounded symbol `Transformation` and the associated semantic type using the Web Ontology Language (OWL [38]) is shown in Listing 2 and Listing 3, respectively. The definition of the semantic type creates a direct link by referencing the defined grounded symbol (see line 6 of Listing 3). Listing 4 shows the generated first class citizens of the target programming language. After the code generation process these elements can be used as shown in Listing 1 in Sect. 4.1.

```
1  <!-- GroundedSymbols.owl -->
2  <Declaration>
3    <Class IRI="#Transformation"/>
4  </Declaration>
5  <SubClassOf>
6    <Class IRI="#Transformation"/>
7    <Class IRI="Core#GroundedSymbol"/>
8  </SubClassOf>
```

Listing 2. OWL definition of the GroundedSymbol `transformation`.

```
1  <!-- SemanticTypes.owl -->
2  <Declaration>
3    <NamedIndividual IRI="#Transformation"/>
4  </Declaration>
5  <ClassAssertion>
6    <Class IRI="GroundedSymbols#Transformation"/>
7    <NamedIndividual IRI="#Transformation"/>
8  </ClassAssertion>
9  <ObjectPropertyAssertion>
10   <ObjectProperty  IRI="Core#hasDataType"/>
11   <NamedIndividual IRI="#Transformation"/>
12   <NamedIndividual IRI="DataTypes#math.Mat4f"/>
13 </ObjectPropertyAssertion>
```

Listing 3. OWL definition of the semantic type `Transformation`. The definition references the grounded symbol from Listing 2 and specifies the data type of the semantic type.

```
1  object GroundedSymbols {
2    object transformation extends GroundedSymbol(
3      symbol = "Transformation",
4      iri = new IRI("GroundedSymbols#Transformation")
5    )
6  }
7  object SemanticTypes {
8    object Transformation extends SemanticType[Mat4](
9      iri = new IRI("SemanticTypes#Transformation"),
10     symbol = GroundedSymbols.transformation)
11   {
12     def apply(value: Mat4): SemanticValue = {/*...*/}
13 } }
```

Listing 4. Grounded symbol `transformation` and semantic type `Transformation` that are generated from the OWL files shown in Listing 2 and 3. The link to ontology concepts is maintained by specifying the respective Internationalized Resource Identifier (IRI).

Due to the utilization of native programming language primitives that are linked to ontology content, the generation-based approach enables fast access to the knowledge representation at runtime (R_{per}). This comes at the cost of additional time to generate code at or before compile time as well as some slight development overhead for the integration of symbols and types in the ontology (as compared to the traditional creation of variables and types). The latter costs are largely reduced by essential tool support for efficient development. For instance, we use common tools, like Protégé, for which we developed a tailored editor plugin. In addition, reasoning software can be utilized at both compile- and runtime. The internal representation of such a software component can be kept in sync with the simulation state by means of said linked primitives. This way, additional assertions can be inferred from existing information and integrated into the application state representation, omitting the need for manual assertion of such inferable facts. This is beneficial for checking semantic constraints, e.g., during multimodal fusion. Moreover, the externalized definition of identifiers, used for application state access (R_{acc}) as well as for symbolic properties, increases cohesion (regarding the definition of identifiers) and further decouples data model, subsystems, and API description. Finally, it facilitates reuse of the interface description even for other RISs and enables use of common editing tools. This way, the issue that ECS components from different applications are mostly incompatible with each other can be overcome.

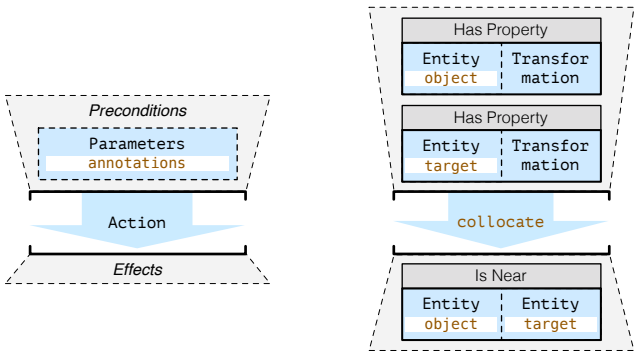

Fig. 5. Illustration of an action description (left) as well as of a concrete practice in the context of the example interaction (right). The grounded action `collocate` is related to the verb *toPut* (cf. Fig. 4) and can be retrieved when processing the respective token. The fusion process is successful and can trigger the action's execution, if further processing of the user's utterance yields an entity that is movable (i.e. has a `Transformation` property) and an entity that it will be moved to.

```
1  object Collocate extends ActionDescription(
2    identifier      = collocate,
3    preConditions   = Set(Entity named subject hasProperty Transformation,
4                          Entity named target  hasProperty Transformation),
5    effects         = Set(Entity named subject is Near the Entity named target)
6  )
7
8  def collocateImplementation(parameters: NamedParameters){/*...*/}
9
10 registerAction(Collocate, collocateImplementation)
11
12 def collocateEntities(subject: Entity, target: Entity){ Planner.accomplish(subject is Near the target) }
```

Listing 5. Action description (lines 1–6), implementation (indicated in line 8), registration (line 10), and execution (using a planning subsystem in line 12) of a grounded `collocate` action that moves one entity (the `subject`) near an other (the `target`). The action description, access to the `parameters` map, as well as the execution utilize grounded symbols and semantic types.

4.3 Grounded Actions

The third essential technique is the semantic description of reusable system operations [12, 46], e.g., the application's reactions to commands the user utters: So called *grounded actions* are implemented as functions in program code and registered by means of action descriptions. These consist of a set of preconditions, a set of parameters, and a set of effects, the contents of which are defined by means of semantic types and -values. In consequence, action descriptions can also be specified in the ontology and transformed into program code.

Fig. 5 and Listing 5 (lines 1–6) illustrate an exemplary action description of a `collocate` action. The implementation of the described function is hinted in line 8 and the registration shown in line 10. Note that any other implementations could be registered with this action description. Hence it is the developer's task to ensure that the specified preconditions and effects match the implementation. Finally, line 12 of Listing 5 illustrates the utilization of a planning subsystem that uses the action description. Since the requested state matches the effects of the `collocate` action description, the planner will verify if the shown preconditions are met and in that case execute the action. Otherwise, it will identify a chain of further registered actions to be executed to satisfy the preconditions or signal an error if no such sequence exists.

Since action descriptions can easily be transformed into fragments of common definition languages, like PDDL [26], the utilization of planning software that is compatible with such languages is facilitated (cf. [25, 46]). If the preconditions of an action are not completely met, action planning subsystems can be applied to automatically derive a sequence of actions that leads to a state that permits the execution of the desired action. This approach allows to react dynamically to a user's requests, even if they were not anticipated by the developers.

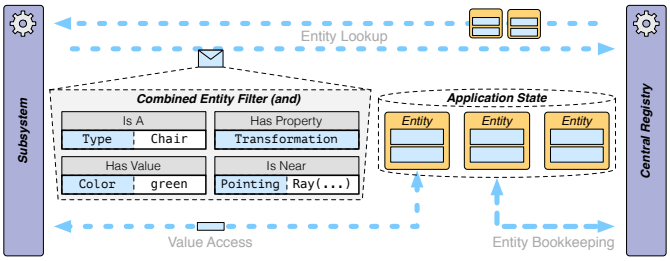

Fig. 6. Conceptual overview of an entity request by a subsystem using a logical combined entity filter. The request is answered with a set of entities that passed the filter. This could be a request necessary during multimodal fusion querying all entities that represent a chair, have a `Transformation` property, are of color green, and are near a given pointing ray. If the application state does not contain a chair fulfilling all requirements, but rather a stool, the implementation of the `IsA` filter can apply a reasoning operation on the associated ontology to determine the user's intention nevertheless, due to available references. Besides initial queries, entity property access does not involve the central registry.

In this context, the definition of concepts in an ontology is beneficial: primitives generated from the ontology (i.e. semantic types) are also utilized for the implementation of the described action. Thus, a uniform interface between the application state, the implementation of actions, and action planning software is achieved via the semantic grounding. Consequently, grounded actions decouple the description of system capabilities, their implementation, and their invocation. This extends the ECS pattern with the possibility to create reusable operations of arbitrary complexity that can be executed by subsystems or stand-alone (R_{exe}). Actions can be reused in other contexts and applications, as long as the required semantic concepts are part of the utilized application ontology. Furthermore, they are beneficial for decision-level multimodal fusion, since grounded actions are typically related to concepts that denote verbs in instruction scenarios. During fusion, processed verbal phrases can be utilized to detect a related grounded action. The action's preconditions and parameters then serve as a frame-like structure that has to be integrated with the rest of the utterance.

4.4 Semantic Queries

The fourth essential technique comprises accessing higher level elements (R_{acc}), i.e. entities and actions, by means of semantic [12, 45] or more general descriptions [46]: On the one hand, this technique deals with requesting one or more entities from a central registry (as opposed to accessing properties of one entity). For this purpose, semantic types and semantic values are utilized to define entity selection filters. A central entity registry can be queried by passing these entity filters. Respective answers contain a list of all matching entities. On the other hand, this technique facilitates grounded action lookup in a similar manner (showcased in Sect. 5).

Fig. 6 illustrates the technique for selecting the green chair from our example. A corresponding code sample is shown in Listing 6.

```
1 def getGreenChairNear(ray: Ray,
2                       handler: Entity => Any) {
3   WorldState apply handler toFirstEntityThat (
4     IsA(Type(chair)) and
5     HasProperty(Transformation) and
6     HasValue(Color(Constants.green)) and
7     IsNear(Pointing(ray))
8   )
9 }
```

Listing 6. A complex entity filter to query the central registry, constructed by passing semantic values as well as -types and combined by applying the function `and` (lines 4–7). By utilizing Scala's capability to omit the dot operator and parentheses in certain cases, the query statement can be written in an easily readable manner. As soon as the central registry has provided the respective answer the `handler` function is called asynchronously. Therein, entity properties can be accessed without involving the registry again. Creating a new matching entity as well as adding or modifying existing entity properties, subsequently matching the filter, will trigger the handler.

```
1  val RightHand = Type(hand) and Chirality(right)
2  Start a new Processor { //that
3    Requires property Transformation from RightHand
4    Updates the properties of entity describedBy RightHand `with` {
5      Position from (Transformation of RightHand)
6  }}
7  Start a new Processor { //that
8    Requires property Position from RightHand
9    Updates the properties of entity describedBy RightHand `with` {
10     val delta = (Position of RightHand at Milliseconds(0)) - (Position of RightHand at Milliseconds(60))
11     Velocity(delta / 60f)
12 }}
```

Listing 7. Semantic values are defined for referencing sinks and sources in line 1. DSL-supported definition of two processors operating on the user's right hand: one that extracts the position from the transformation (lines 2-6) and one that calculates the velocity from the position (lines 7-12).

Besides specific queries, this technique allows subsystems to specify their data sinks and sources without requiring explicit entity references. These, at large, are the observable properties of entities. For instance, a subsystem that processes the transformation of the user's right hand does not need the respective entity reference nor does it need to know the entity's creator. Instead, a query using an entity filter checking for `Type(hand)` and `Chirality(right)` can be used at its creation.

This semantics-based state access decouples subsystems in terms of data sinks and sources. It facilitates entity lookup based on combinations of predicates and symbolic as well as numeric data. This is especially useful for the incorporation of AI methods. Moreover, it facilitates reuse of subsystems in other contexts or applications, as long as the required application state elements exist. In general, the approach is similar to semantic query languages like SPARQL. However, it enables to avoid parsing overhead, if the code generation technique presented above is applied. Semantic queries counter the problem of identifying entities in ECS applications by facilitating mutual access for subsystems to entities outside of their primary data association.

4.5 Decoupling by semantics

The fifth essential technique applies all previous techniques to decouple processing (R_{proc}) in RISs. Instead of passing specific entity references or callback functions on creation, required data sources and sinks as well as application specific operations are semantically described utilizing grounded actions and semantic queries.

Listing 7 exemplifies the definition of two simple processing steps, so called *processors*, that run calculations or algorithms as a reaction to changes of required entity properties [22]. These are specified using semantic values (lines 3 and 8), which are implicitly converted to semantic queries using a conjunction of `HasProperty`. Upon value changes the specified calculation rules (lines 5 and 10–11) are applied, respectively. They are implemented using convenience functions defined for semantic types and values, e.g., the `from` function in line 5 or the `-` function in line 10 (cf. [12]). Their results are each used to set a property of another specified entity. Timestamps and histories of required properties are automatically stored. Access to those locally stored values is facilitated by a combination of a semantic type, a semantic query, and the `of` function, e.g., `Transformation of RightHand` in line 5. Local histories are accessible using the `at` function (line 10). The passed value specifies a point in time relative to the latest value available. In case of multiple requirements with asynchronous update rates this access additionally performs interpolations or extrapolations to guarantee temporal synchronization.

Decoupling by semantics allows the definition of dataflow network-like processing chain elements by means of semantic descriptions and improves the ECS approach to the coupling dilemma. Defined processing steps are highly reusable, due to the utilization of semantic queries and grounded actions (showcased in the next section). Moreover, the technique facilitates high-level APIs that foster usability for developers. By using Scala, such a high-level API can be realized in native code without any further intermediate representation below the DSLs. Thus, IDE features as syntax checks, highlighting, or auto-completion can be exploited, further increasing API usability.

5 Multimodal Input Processing

The realization of multimodal interfaces entails demanding requirements for RISs. Therefore, we use it to further highlight the benefits of the presented techniques. Multimodal input has to be processed on data-, feature-, and decision-level in order to derive an adequate system reaction to a user utterance (see Fig. 7).

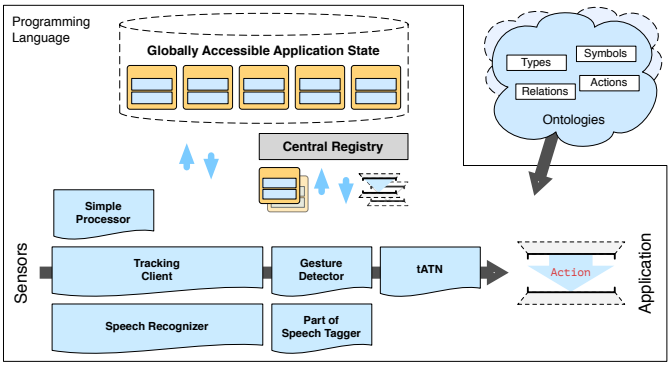

Fig. 7. Overview of an exemplary processing chain. Simple calculations are realized using the basic processors. In addition, more complex techniques or external libraries are integrated by means of specialized processors. All steps utilize semantic queries to specify sources, sinks, and grounded actions as well as semantic types and -values to access entity properties. The final system reaction is represented by an action.

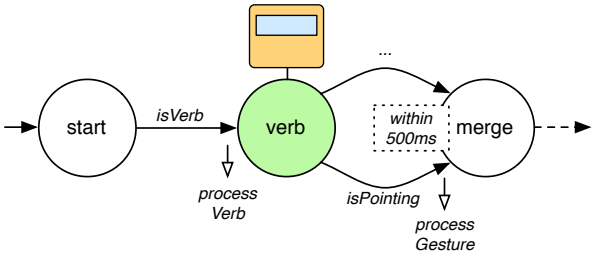

Fig. 8. Excerpt of a tATN capable of parsing "*Put* [pointing] *that*". It is composed of states (circles), transitions (black arrows) with constraints, and functions that are applied if a transition is carried out (white arrows). Cursors (green) represent active interim results. They fill the associated registers (orange) using semantic values (blue) as they move through the tATN: certain application state changes, e.g., updated speech recognition or gesture detection results, trigger the evaluation of the cursors' outgoing transitions. Reaching an end state (not shown) corresponds to a successful parse of a multimodal utterance and should gather all data relevant for an respective action invocation in the cursor's register.

```
1  Start a new SupervisedLearningProcessor { //that
2    Is configuredBy NeuralNetworkConfiguration("pointing.xml")
3    Requires properties (Transformation and Velocity) from RightHand
4    Updates property PointingConfidence of entity describedBy User `with` {prediction}
5  } // Assuming a similar processing step for 'rotate' gestures
6  Start a new Processor { //that
7    Requires properties (PointingConfidence and RotateConfidence) from User
8    Updates the properties of entity describedBy User `with` {
9      if((PointingConfidence of User) > (RotateConfidence of User)) Gesture(pointing) else Gesture(rotate)
10   }}
```

Listing 8. Definition of a simple gesture classification. A specialized processor utilizing a neural network (lines 1-5) is defined to take the tranformation and the velocity of the user's right hand as input. The prediction of the network is stored in the `PointingConfidence` property of the entity representing the user. The configuration of the network is outsourced in *pointing.xml*. A subsequent processing step is defined to do a simple classification task (lines 6-13), by comparing pointing- and rotate confidence and storing the result in the `Gesture` property of the user entity using grounded symbols.

For the example interaction, it is useful to extract additional features from the raw positions and orientations of the user's joints before applying template-based or supervised learning methods to detect gestures like pointing (R_{proc}). In contrast to simple processing steps, where calculation rules can be defined inline, more complex techniques are integrated by means of specialized processors. The typically required additional configuration is outsourced into external files or classes, while requirements and results are defined in analogy to other processing steps. A classification utilizing a neural network is defined in Listing 8 by requiring entity properties as input for the network as well as by specifying a sink property by means of the semantic type `PointingConfidence` that has to be of data type `float`. All required properties are sorted and fed into a suitable network. The prediction of the network is treated as the result of the processing step's calculation rule. The outsourced configuration file contains information like the storage location of training related data and optional parameters that configure parts of the automated behavior.

The final step for processing speech and gesture input typically applies an multimodal fusion approach. Fig. 8 presents an example of this based on an temporal Augmented Transition Network (tATN) [20]. The corresponding processor configuration is showcased in Listing 9.

Functions referenced in the tATN definition can use semantic queries and semantic grounding, e.g., to resolve a pointing ray similar to Listing 6, to alter the application state to provide feedback to the user, or to start a processor to continuously map user input to virtual element properties. Moreover, if a user-desired state is derived during parsing, a planning subsystem can trigger respective grounded actions (cf. Listing 5). An alternative approach realizing semantic queries for a grounded action lookup is showcased in Listing 10. The required dictionary is illustrated in Listing 11. It can be auto-generated from the system's ontology, if a ontology containing words supported by the application is imported and necessary relations to grounded symbols and actions are added.

Altogether, the application of the presented techniques for the realization of multimodal interfaces results in a highly configurable solution suitable for analyzing input on data-, feature-, and decision level. The underlying semantic grounding is highly beneficial for decision level processing and fusion approaches, since they typically deal with symbolic data. Moreover, entity properties required by single steps are described in terms of their semantics and requested from a central registry. They are thus decoupled from other processing steps or subsystems, which fosters exchangeability and reuse. Processors, described using a high-level DSL, support API usability, especially in combination with Scala's syntax capabilities.

```
1  class UniversalInteractions extends AtnLayout {
2   create StartState "start" withArc "verb" toState "split"
3   create SplitState "split"
4     withCondition TimeConstraint(Milliseconds(500))
5     withArc "gesture"    toState "merge"
6     withArc "determiner" toState "merge"
7   create MergeState "merge" //...
8   create Arc "verb" withCondition isVerb andFunc doVerb
9   create Arc "gesture" withCondition isPointing
10    andFunction processGesture /*...*/ }
```

Listing 9. Excerpt of an outsourced tATN configuration implementing Fig. 8. The related processor is defined to require the user's gestures and spoken tokens as input and to update the user entity's `Action` property with successfully fused commands. The internal DSL for defining states (lines 2-6) and arcs (lines 7-10) is intermixed with references to functions (`isVerb`, `processVerb`, `isGesture`, and `processGesture`). The split-merge construct (lines 3-7) is utilized to constrain occurences of the pointing gesture and the determiner *that* to a time window of $500ms$.

```
1  def doVerb(input: Event, register: Register) {
2    val token  = input.get(Token).value
3    val pos    = Dictionary.verbs(token)
4    val action = ActionRegister.lookUp(pos.assocAction)
5    register.put(action)
6  }
```

Listing 10. Definition of a transition function for the tATN of Listing 9. The application state change that triggered the transition (`input`) can be assumed to be a verb, e.g., *put*. It is used to lookup an associated grounded action using a dictionary (lines 2–4). The retrieved action is stored in the cursors register to subsequently facilitate checks if parsed input matches the action's parameters and conditions and to ultimately execute or return the action if an end state is reached.

6 Results

In the previous sections we introduced five techniques to enhance software quality in multimodal RISs and illustrated their application with the aid of a multimodal input processing use-case. They solve the ECS pattern's runtime type as well as incompatibility issues. Moreover, they improve component granularity as well as mutual access to entities by subsystems to match the requirements of complex AI dependent RISs, e.g., realizing multimodal interfaces. In addition, they extend the pattern with a technique for semantically described reusable operations. The showcased reference implementation comprises de-

```
1  object Chair() extends Noun {
2    val entityReference = Type(chair)}
3  object Green() extends Adjective {
4    val associatedProperty = Color(Constants.green))}
5  object Collocate() extends Verb {
6    val actionReference = collocate}
7  object Dictionary {
8    val verbs =
9      Map("put"   -> Collocate() , /*...*/)
10   val adjectives =
11     Map("green" -> Green(), /*...*/)
12   val nouns =
13     Map("chair" -> Chair(), /*...*/)}
```

Listing 11. Explicit mapping of parts of speech [47]: verbs denote actions, adjectives entity properties, and nouns entities.

sign decisions, like the use of the Scala programming language and the internal DSL. The presented techniques, however, are independent of a certain programming language. Table 2 shows their mutual dependencies in case only a subset of them is to be implemented. Additional implementation hints and details can be found in previous work on semantic grounding [44], code from semantics [41,44], grounded actions [12,46], semantic queries [12,45,46], and decoupling by semantics [22]. The remainder of this section will discuss the showcased reference implementation and the impact of the presented techniques on maintainability and performance, including effects on coupling.

6.1 Reference Implementation

The presented techniques can be made use of by two approaches: (1) By low-level system integration. This requires access to the specific application state management to enable semantic grounding. Here, the presence of an entity model is beneficial. (2) By loose coupling of a (partly) closed source system. This requires an existing system that realizes approach (1). Our reference implementation comprises both approaches. Variant (1) is showcased in this paper. In addition, the coupling to two commercial game engines, Unity3D [37] and Unreal 4 [9], yielded first promising results [8,43] for variant (2).

Several design decisions of the reference implementation are not necessary for the presented techniques. On the one hand, Scala was chosen due to its combination of object-oriented and functional paradigms as well as due to its syntactical flexibility. The former is beneficial for addressing a preferably large number of developers while facilitating concise APIs, especially comprising callbacks. The latter greatly facilitates the definition of internal DSLs. On the other hand, actors provide a solution for typical RIS concurrency requirements and foster scalability and extensibility. Their use avoids the necessity for techniques like mutexes to synchronize memory access. High-level synchronization mechanisms, e.g., to handle temporal dependencies between rendering and physics simulation, may still be required and have to be implemented at higher layers. An actor model implementation is natively included in Scala. Both decisions positively influence maintainability [42].

The API showcased throughout the listings is the proposed solution for implementing the presented techniques. Our current DSL consists of about 10 constructs that proved sufficient for our test cases and demonstrations. It can be extended and intermixed with non-DSL API calls and is subject of our ongoing research.

6.2 Maintainability

Evaluating or even comparing maintainability is an expensive endeavor with a limited amount of methods and no obvious choices [32]. This holds especially true for API usability [31], despite its critical impact. Available objective reusability, modularity, and modifiability prediction methods are based on algorithms that automatically analyze source code based on a certain metric [32]. Subjective methods rely on expert judgment in the context of reviews [27,32] and are the most commonly applied form of evaluation for these software qualities [32]. To assess API usability subjective methods are even more essential, e.g., API peer reviews [33] or questionnaires based on programming tasks [31].

However, all subjective measures implicate one crucial requirement: persons that are familiar enough with a certain platform to solve complex tasks or reflect on non-functional qualities. This obviously is a problem in the area of complex RISs. At least, the availability of such experts for widespread platforms is moderate. However, those platforms are often commercial and do not permit changes necessary to conduct research on maintainability. Open-source research platforms facilitate the necessary flexibility with the drawback of an often scattered user base. Given these constraints, we applied several methods to ensure high or rather improved maintainability including API usability: (1) We applied the API peer review method presented in [33]. The results of this process are showcased by the presented code samples. Following the review's idea, readers can assess the API usability themselves with the aid of the code samples and the explanations in this paper. (2) The reference implementation is utilized in several VR and MR research projects as well as for various teaching projects, including a dedicated multimodal interaction course (cf. Fig. 1). This way, we are able to get API usability feedback on dedicated programming tasks in the sense of [31]. (3) We conducted expert reviews accompanying the development of the presented techniques targeting maintainability in general [42]. (4) Objective measures have been conducted showcasing the improved reusability of the presented concepts outside the scope of this paper [41].

The lessons we learned concerning maintainability especially related to API usability can be summarized as follows:

- Good workflow, tool support, and IDE integration are essential.
- The functional paradigm fosters a concise API that is tricky at first but beneficial when used longer and for non-trivial tasks.
- Good code examples greatly ease the issue of missing documentation (due to limited resources).
- API oversimplification will result in decreased understandability.

Especially the first lesson is key to inhibit potential development overhead and to prevent a misuse of concepts resulting in idiosyncratic code and poor maintainability.

		1	2	3	4	Facilitated requirements
semantic grounding	**1**					R_{rep}, R_{acc}
code from semantics	**2**	r				R_{per}, R_{acc}
grounded actions	**3**	r	b			R_{exe}
semantic queries	**4**	r	b	b		R_{acc}
decoupling by semantics	**5**	r	b	b	r	R_{proc}

Table 2. Dependencies between the presented techniques as well as facilitated requirements. Each row represents a technique and indicates which other techniques are required (r) or beneficial (b) for its implementation. Code from semantics is not required to implement the other techniques, since the generation result can also be defined manually. However, utilizing ontologies fosters exchangeability and the use of planning software as described in Sect. 4.2. Similarly, semantic queries and decoupling by semantics can be realized solely upon semantic grounding. However, grounded actions contribute to further decouple function executions, as showcased by the multimodal processing use case. In addition, essential requirements that are facilitated by a certain technique are summarized. See Sect. 4 for details.

6.3 Performance Evaluation

The presented techniques introduce a novel way of representing and accessing the application state (R_{rep} and R_{acc}), the most fundamental, and thus most important operations in a RIS. Naturally, their beneficial aspects come at the cost of some performance overhead, which will be discussed in this section.

Semantic grounding introduces one layer of redirection: symbol-based access of properties. Two representative alternatives are contrasted with the presented approach considering performance overhead (R_{per}), expressivity, and effect on coupling. The fastest possible approach – direct access to variables and invocation of functions – would result in the strongest coupling between subsystems and thus in a maintenance nightmare. Hence, it is not considered any further. The closest alternative to direct access is the utilization of a hash map that links values with some sort of identifier, introducing one layer of redirection. This approach is adopted by most implementations of the ECS pattern, e.g., in the form of wait-free hash maps [17]. Maximum independence is achieved by an approach that uses a centralized or external storage that is accessed via a dedicated query language. This would immensely reduce coupling, but at the cost of parsing overhead.

In order to compare the performance of the different approaches we performed a simple benchmark, measuring a combined read/write operation. The benchmark consisted in invoking a method that updates a floating point vector with three elements and reading the vector back from the storage. The operation was requested by one actor and executed by another, simulating the communication between two

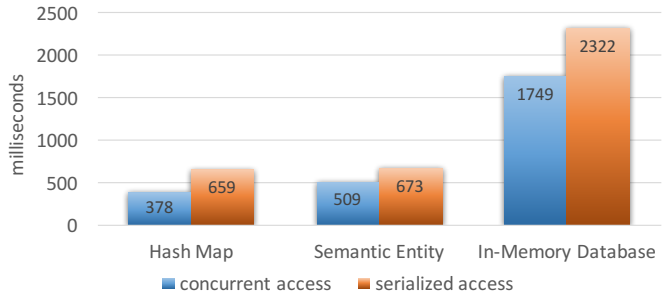

Fig. 9. Performance comparison of three different state representation alternatives: hash maps, semantic entities, and in-memory databases. Times (in milliseconds) were measured for 100,000 read and write operations. Blue columns represent concurrent access, whereas red columns represent sequential read and write operations.

concurrent subsystems. Two different types of access were tested: alternating read and write requests were sent, waiting for one request to be processed before sending the next (serialized access) and all read and write requests were sent at once (concurrent access). In both cases the time between sending the first request and receiving the last response was measured. All benchmarks were conducted on the same hardware within the environment of our reference implementation, which introduces some overhead. Nevertheless, the depicted relative differences allow a performance assessment.

The results for the hash map-based approach are shown in column *HashMap* in Fig. 9. As expected, this variant preformed best. In addition to performance advantages, the use of a programming language, as applied in this case, provides the highest expressive power of the compared approaches. On the downside, relying on application-specific identifiers and function names yields close coupling: often unanticipated changes of function or class signatures necessitate the adaption of large parts of a program; even if the essential functionality is retained.

One solution to the problem of such close coupling is the use of a centralized or external storage, e.g., a database. Access to content then is decoupled via a specific query language like SQL. As compared to general purpose programming languages, modification of data with such query languages is rather limited and mostly restricted to accessing and updating a database's contents. The used relational database consisted of a single table with four columns, one for the identifier and three for the floating point numbers of the vector to be stored. Basic SQL statements like

```
UPDATE sqlData SET x=?,y=?,z=? WHERE id=?
INSERT INTO sqlData (id,x,y,z) VALUES (?,?,?,?)
```

were used to modify the database. To avoid costly I/O operations a SQLite in-memory database was used. Column *In-Memory Database* in Fig. 9 shows the results from our benchmark: the results are about five times worse as compared to the hash map approach using concurrent access and more than three times worse using serialized access (the less common variant in RISs). Consequently, the database approach is problematic regarding high performance applications.

Technique one fosters the use of an external ontology to structure state representation beyond common mechanisms provided by programming languages, e.g., using relations between concepts. The benefits of semantic databases and direct access are combined by integration of a (generated) semantic layer. Resulting performance is close to that of the hash-map approach, see column *Semantic Entity* of Fig. 9.

Code from semantics counters query parsing overhead. It introduces overhead only at compile time, the amount of which depends on the size of the ontology in use.

Grounded actions rely on semantic grounding with similar overhead costs. Apart from this, executing actions equals calling functions.

Semantic queries rely on a lookup in a central registry. However, lookup overhead only occurs at first access, i.e. during bootstrapping in typical RIS applications. Afterwards references to entities or their properties are used for further operations. Additional performance overhead is owed to the used subsystems, e.g., reference resolution and reasoning in multimodal use-cases. It thus is not inherent to the presented techniques but would arise in the same way in applications that do not apply these techniques.

Decoupling by semantics introduces no additional fundamental overhead compared to alternative implementations of threads of execution, since this technique solely defines an API scheme for reoccurring requirements that utilizes the other techniques.

7 Conclusion

In this paper, we presented five semantics-based techniques for state management in RIS applications to increase software quality, i.e. maintainability: semantic grounding, code from semantics, semantic queries, grounded actions, and decoupling by semantics. The techniques improve and extend the ECS pattern, which is widely used in current RIS applications. They enhance the pattern's capabilities of representing and accessing the application state by introducing a uniform semantic access on different levels and facilitate the creation of decoupled subsystems. An elaboration of performance characteristics, including a comparison with alternative state representation approaches, validated the feasibility of the presented techniques. Every aspect except the top-level DSL is independent of a certain programming language. Developers can thus already benefit from the presented ideas, possibly implementing them in their own software.

A reference implementation is showcased by several code samples, which are the results of a conducted API peer review process to improve API usability. Its source code is available to the research community to gather feedback and promote reuse [43]. Since the underlying techniques foster decoupling, modularity is increased and existing features can be (re-)used more easily, even by non-experts. The latter now is confirmed in many topic-related courses and student projects, in which participants successfully performed the highly complex task of implementing multimodal (in that case speech- and gesture-based) VR interfaces using the reference implementation (see Fig. 1).

In our ongoing research we evaluate additional performance aspects, like the resolution of complex semantic entity references or planning tasks. In terms of multimodal processing, we work on more diverse applications as well as on using the reference implementation to conduct a comprehensive evaluation or comparison of alternative multimodal processing techniques, like tATNs, unifications, and finite-state transducers. Finally, we will verify central non-functional software qualities by encouraging more tool support and by applying additional evaluation methods.

References

[1] T. Alatalo. An Entity-Component Model for Extensible Virtual Worlds. *Internet Computing, IEEE*, 15(5):30–37, 2011.

[2] A. Ameri Ekhtiarabadi, B. Akan, B. Çürüklu, and L. Asplund. A General Framework for Incremental Processing of Multimodal Inputs. In *Proceedings of the 13th International Conference on Multimodal Interfaces*, ICMI, pages 225–228. ACM, 2011.

[3] R. A. Bolt. „Put-that-there": Voice and Gesture at the Graphics Interface. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, pages 262–270. ACM, 1980.

[4] J. Bouchet, L. Nigay, and T. Ganille. ICARE Software Components for Rapidly Developing Multimodal Interfaces. In *Proceedings of the 6th International Conference on Multimodal Interfaces*, ICMI, pages 251–258. ACM, 2004.

[5] A. Cherubini, R. Passama, P. Fraisse, and A. Crosnier. A unified multimodal control framework for human–robot interaction. *Robotics and Autonomous Systems*, 70:106–115, 2015.

[6] P. R. Cohen, M. Johnston, D. McGee, S. Oviatt, J. Pittman, I. Smith, L. Chen, and J. Clow. QuickSet: Multimodal Interaction for Distributed Applications. In *Proceedings of the Fifth International Conference on Multimedia*, MULTIMEDIA, pages 31–40. ACM, 1997.

[7] B. Dumas, D. Lalanne, and R. Ingold. HephaisTK: A Toolkit for Rapid Prototyping of Multimodal Interfaces. In *Proceedings of the 2009 International Conference on Multimodal Interfaces*, ICMI, pages 231–232. ACM, 2009.

[8] B. Eckstein, J. L. Lugrin, D. Wiebusch, and M. E. Latoschik. PEARS: Physics extension and representation through semantics. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(2):178–189, 2016.

[9] Epic Games, Inc. Unreal Engine 4. `www.unrealengine.com`, 2016. Last accessed 2016-12-01.

[10] M. Fischbach. Software Techniques for Multimodal Input Processing in Realtime Interactive Systems. In *Proceedings of the 17th International Conference on Multimodal Interaction*, ICMI, pages 623–627. ACM, 2015.

[11] M. Fischbach, D. Wiebusch, A. Giebler-Schubert, M. E. Latoschik, S. Rehfeld, and H. Tramberend. SiXton's curse - Simulator X demonstration. In *Virtual Reality Conference*, VR, pages 255–256. IEEE, 2011.

[12] M. Fischbach, D. Wiebusch, and M. E. Latoschik. Semantics-based Software Techniques for Maintainable Multimodal Input Processing in Real-time Interactive Systems. In *9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems*, SEARIS, pages 1–6. IEEE, 2016.

[13] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[14] L. Hoste, B. Dumas, and B. Signer. Mudra: A Unified Multimodal Interaction Framework. In *Proceedings of the 13th International Conference on Multimodal Interfaces*, ICMI, pages 97–104. ACM, 2011.

[15] D. B. Koons and C. J. Sparrell. Iconic: Speech and Depictive Gestures at the Human-machine Interface. In *Conference Companion on Human Factors in Computing Systems*, CHI, pages 453–454. ACM, 1994.

[16] P. Lange, A. Probst, A. Srinivas, G. G. Peytavi, C. Rachuy, A. Schattel, V. Schwarting, J. Clemens, D. Nakath, M. Echim, et al. Virtual Reality for Simulating Autonomous Deep-Space Navigation and Mining. In *24th International conference on Artificial Reality and Teleexistence*, ICAT-EGVE, pages 15–16. Eurographics Association, 2014.

[17] P. Lange, R. Weller, and G. Zachmann. Wait-Free Hash Maps in the Entity-Component-System Pattern for Realtime Interactive Systems. In *9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems*, SEARIS, pages 1–8. IEEE, 2016.

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

[19] M. E. Latoschik. A General Framework for Multimodal Interaction in Virtual Reality Systems: PrOSA. In *The Future of VR and AR Interfaces - Multimodal, Humanoid, Adaptive and Intelligent.*, pages 21–25. IEEE, 2001.

[20] M. E. Latoschik. Designing Transition Networks for Multimodal VR-Interactions Using a Markup Language. In *Proceedings of the Fourth International Conference on Multimodal Interfaces*, ICMI, pages 411–416. ACM, 2002.

[21] M. E. Latoschik. A User Interface Framework for Multimodal VR Interactions. In *Proceedings of the 7th International Conference on Multimodal Interfaces*, ICMI, pages 76–83. ACM, 2005.

[22] M. E. Latoschik and M. Fischbach. Engineering Variance: Software Techniques for Scalable, Customizable, and Reusable Multimodal Processing. In *Proceedings of the HCI International Conference*, HCII, pages 308–319. Springer, 2014.

[23] M. E. Latoschik and H. Tramberend. Engineering Realtime Interactive Systems: Coupling & Cohesion of Architecture Mechanisms. In *Proceedings of the 16th Eurographics Conference on Virtual Environments & Second Joint Virtual Reality*, EGVE - JVRC, pages 25–28. Eurographics Association, 2010.

[24] J.-Y. L. Lawson, A.-A. Al-Akkad, J. Vanderdonckt, and B. Macq. An Open Source Workbench for Prototyping Multimodal Interactions Based on Off-the-shelf Heterogeneous Components. In *Proceedings of the 1st SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS, pages 245–254. ACM, 2009.

[25] S. Link, B. Barkschat, C. Zimmerer, M. Fischbach, D. Wiebusch, J.-L. Lugrin, and M. E. Latoschik. An Intelligent Multimodal Mixed Reality Real-Time Strategy Game. In *Proceedings of the 23rd IEEE Virtual Reality conference*, VR, pages 223–224. IEEE, 2016.

[26] 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.

[27] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.

[28] H. Mendonça, J.-Y. L. Lawson, O. Vybornova, B. Macq, and J. Vanderdonckt. A Fusion Framework for Multimodal Interactive Applications. In *Proceedings of the International Conference on Multimodal Interfaces*, ICMI-MLMI, pages 161–168. ACM, 2009.

[29] A. Möller, S. Diewald, L. Roalter, and M. Kranz. Supporting Mobile Multimodal Interaction with a Rule-Based Framework. *CoRR*, abs/1406.3225, 2014.

[30] J. G. Neal, C. Y. Thielman, Z. Dobes, S. M. Haller, and S. C. Shapiro. Natural Language with Integrated Deictic and Graphic Gestures. In *Proceedings of the Workshop on Speech and Natural Language*, HLT. Association for Computational Linguistics, 1989.

[31] M. Piccioni, C. A. Furia, and B. Meyer. An Empirical Study of API Usability. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 5–14, Oct 2013.

[32] M. Riaz, E. Mendes, and E. Tempero. A Systematic Review of Software Maintainability Prediction and Metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 367–377. IEEE Computer Society, 2009.

[33] N. Ruiz, F. Chen, and S. Oviatt. Multimodal input. In J.-P. Thiran, F. Marqués, and H. Bourlard, editors, *Multimodal Signal Processing*, chapter 12, pages 231 – 255. Academic Press, Oxford, 2010.

[34] M. Serrano, L. Nigay, J.-Y. L. Lawson, A. Ramsay, R. Murray-Smith, and S. Denef. The Openinterface Framework: A Tool for Multimodal Interaction. In *Extended Abstracts on Human Factors in Computing Systems*, CHI, pages 3501–3506. ACM, 2008.

[35] J. Shen and M. Pantic. HCI^2 Framework: A Software Framework for Multimodal Human-Computer Interaction Systems. *Cybernetics, IEEE Transactions on*, 43(6):1593–1606, 2013.

[36] W. W. Tang, K. W. Lo, A. T. Chan, S. Chan, H. V. Leong, and G. Ngai. I*Chameleon: A Scalable and Extensible Framework for Multimodal Interaction. In *Extended Abstracts on Human Factors in Computing Systems*, CHI, pages 305–310. ACM, 2011.

[37] Unity Technologies. Unity. `http://www.unity3d.com`, 2015. Last accessed 2016-12-01.

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

[39] J. Wagner, F. Lingenfelser, T. Baur, I. Damian, F. Kistler, and E. André. The Social Signal Interpretation (SSI) Framework: Multimodal Signal Processing and Recognition in Real-time. In *Proceedings of the 21st ACM International Conference on Multimedia*, MM, pages 831–834. ACM, 2013.

[40] W. Wahlster. User and Discourse Models for Multimodal Communication. In J. W. Sullivan and S. W. Tyler, editors, *Intelligent User Interfaces*, pages 45–67. ACM Press, 1991.

[41] D. Wiebusch. *Reusability for Intelligent Realtime Interactive Systems*. Würzburg University Press, Würzburg, 2016. urn:nbn:de:bvb:20-opus-121869.

[42] D. Wiebusch, M. Fischbach, M. E. Latoschik, and H. Tramberend. Evaluating Scala, Actors, & Ontologies for Intelligent Realtime Interactive Systems. In *Proceedings of the 18th ACM Symposium on Virtual Reality Software and Technology*, VRST, pages 153–160. ACM, 2012.

[43] D. Wiebusch, M. Fischbach, S. Rehfeld, H. Tramberend, and M. E. Latoschik. Simulator X. `http://www.hci.uni-wuerzburg.de/projects/simulator-x.html`. Last accessed 2016-12-09.

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

[45] D. Wiebusch and M. E. Latoschik. A Uniform semantic-based Access Model for Realtime Interactive Systems. In *Workshop on Software Engineering and Architectures for Realtime Interactive Systems*, SEARIS. IEEE, 2014.

[46] D. Wiebusch and M. E. Latoschik. Decoupling the Entity-Component-System Pattern using Semantic Traits for Reusable Realtime Interactive Systems. In *Workshop on Software Engineering and Architectures for Realtime Interactive Systems*, SEARIS. IEEE, 2015.

[47] C. Zimmerer, M. Fischbach, and M. E. Latoschik. Maintainable Management and Access of Lexical Knowledge for Multimodal Virtual Reality Interfaces. In *Proceeding of the 22nd ACM Symposium on Virtual Reality Software and Technology*, VRST. ACM, 2016.