Engineering Variance: Software Techniques for Scalable, Customizable, and Reusable Multimodal Processing

Marc Erich Latoschik and Martin Fischbach

HCI group, University of Würzburg, Germany,
marc.latoschik@uni-wuerzburg.de,
martin.fischbach@uni-wuerzburg.de
WWW home page: http://hci.uni-wuerzburg.de

Abstract. This article describes four software techniques to enhance the overall quality of multimodal processing software and to include concurrency and variance due to individual characteristics and cultural context. First, the processing steps are decentralized and distributed using the actor model. Second, functor objects decouple domain- and application-specific operations from universal processing methods. Third, domain specific languages are provided inside of specialized feature processing units to define necessary algorithms in a human-readable and comprehensible format. Fourth, constituents of the DSLs (including the functors) are semantically grounded into a common ontology supporting syntactic and semantic correctness checks as well as code-generation capabilities. These techniques provide scalable, customizable, and reusable technical solutions for reoccurring multimodal processing tasks.

Keywords: Multimodal processing, interactive systems, software architecture, actor system, DSL, reactive manifesto, software patterns

1 Introduction

Variance is a central aspect of multimodal utterances. For example, the temporal correlation between the occurrence of deictic markers in gesture and speech has been studied for decades [10]. Such surface patterns identify intervals, not singular points in time, in which a co-occurrence is likely and a semantic relation implied. This variance pervades various aspects of the phenomenology of multimodal utterances. Individual characteristics as given by users' personalities and physiology as well as cultural context have a notable impact on such variances.

As a result, variance has to be a central characteristic at various stages of multimodal processing models. These models must be capable of expressing variance on the surface structures as well as on deeper layers like, e.g., the semantic extend of referential expressions, or the semantic grounding of surface utterances. In addition, multimodal utterances often combine parallel signals from various channels. Sequential processing architectures and strategies often fail to capture this inherent concurrency and have to include technical complexities due to interleaved processing and buffering.

Technical systems for processing of multimodal utterances have to cope with the functional aspects implied by variance and concurrency. In addition, a lack of common technical solutions [16] has a negative impact on the overall progress in the field. This article illustrates four software techniques targeted at scalable, customizable and reusable technical solutions for reoccurring multimodal processing tasks.

2 Related Work

Multimodal processing has to deal with variance from low-level sub-tasks like signal processing to high-level multimodal fusion and analysis tasks. For example, gesture analysis and detection based on machine learning approaches, e.g., neural networks [17, 1] incorporates variance implicitly as predetermined by the variance encoded in the training samples. Template-based approaches like in [14] have to deal with variance explicitly, e.g., by inserting fuzzy constraints.

As a second example, procedural methods (e.g., transition networks [8, 13]), alternative parsing strategies [4], and frame-based approaches [11, 2] gained a lot of interest as fusion methods in the field of interactive systems due to a potential performance advantage compared to unification [9, 7]. They all have to explicitly deal with variance during the central matching operation of two fusion candidates. The advantage of unification is its generalizing aspect due to the abandonment of domain-specific adaptations. Correlations are expressed via the so-called *agreement* between uniform features but agreement is usually mapped to equality which does not consider variance. Similar observations hold true for almost all sub-tasks of multimodal processing.

In addition, software quality requirements have been identified to be a necessity for multimodal processing [16, 12]. Tight coupling of the multimodal processing to a predefined execution scheme as well as hard-coded variance-handling code scattered over the various processing sub-tasks greatly weakens scalability, customizability, and reusability. This includes the overall architecture and execution scheme of the processing, i.e., the data and control flow. For example, initial work was characterized by a tight coupling of the gesture processing and fusion with the execution scheme of the simulation middleware used [14]. The Mudra framework–as a recent approach–is coupled to CLIPS as the central rule-based production system and semantic model [6]. Similar approaches often use staged pipeline models. Such models condense and aggregate data from a sensor layer to a final fusion layer, loosely following an hierarchical *from signal to symbol* approach like in [18].

The follow-up sections will introduce the core ideas of the multimodal interaction processing toolkit miPRO. miPRO contains several design and implementation choices which improve scalability, customizability, and reusability and hence provides a sophisticated middleware for engineering variance during multimodal processing tasks.

3 Decomposition of Tasks: Processors and Meshes

The processing of multimodal input is decomposed into a loosely coupled mesh of modular processing units communicating via events (see figures 1 and 2). These processors transform input into output features, e.g., they may map the raw position of the user's left hand or the spoken words to ontology concepts like LEFT_HAND_UP or SPELL_COMMAND respectively. Input and output of processors range from raw sensor signals to intermediate or even end results, all uniformly accessed using concepts from the ontology. Processors define their input requirements and resulting products in terms of these concepts. Input is always buffered: A time series management facility provides simple temporal look-back, interpolation, as well as advanced aggregation facilities potentially necessary. The processor mesh is rearrangeable and provides individual execution schemes, hence fostering customizability, and reusability.

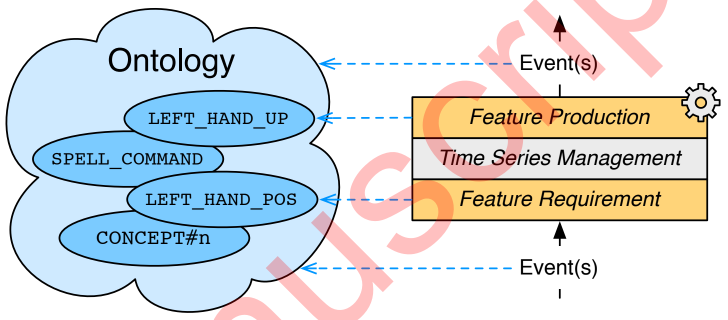

Fig. 1. A single processor (r) and its link into the ontology (l). Processors are executed by individual actors (depicted by the gearwheel) and uniformly communicate with each other by events.

The architecture of the multimodal processing framework is based on Hewitt's actor model [5] to provide scalability based on distribution and concurrency features. It is implemented on top of Simulator X [15], a flexible open-source simulation middleware for realtime interactive systems. This middleware features an entity model to provide an object-centered access to the global simulation state and hence the target domain. In addition, an event system facilitates message passing using a provide/require pattern. Entities and events are grounded into an ontology [19] in order to decouple components, to foster uniform and human readable access patterns, as well as to provide an inherent interface to the domain's semantic description. The latter is highly beneficial for symbolic artificial intelligence (AI) methods, necessary during the semantic and pragmatic interpretation of the multimodal utterances.

Each processor and hence production is concurrently executed by a dedicated actor. Flow of control follows the reactive manifesto. Execution inside of the productions is event-triggered using the underlying actor message system. After receiving new input, the processor executes its local production(s) and sends the result(s) to registered receiving processors in the mesh via events. Currently, the following production methods are supported:

1. **Native:** User-defined tasks (calculus, linear algebra, etc.).
2. **State Wrappers:** State change monitoring.
3. **Native DSL:** User-defined with domain-specific syntax.
4. **Augmented Transition Networks:** Parsing, classification, and fusion.
5. **Unification:** Parsing and fusion.
6. **Supervised learning:** Enhanced classification tasks.

The listed production methods are sorted in increasing levels of abstraction and accompanied decreasing freedom of expressiveness. For example, interaction engineers have to follow a rather predefined syntax inside of the productions of 5 and 6. Here, they don't have to cope with concrete algorithms but have to define parameters for the underlying production method. On the other extreme, the productions inside of native processors (1.) contain code written in the native host language (Scala). Hence, they allow the full scope of potential algorithms and, unfortunately, styles of programming. This expressiveness comes at a price. Missing guidance and standards usually results in highly individual code including self-defined identifiers and idiosyncratic solutions. This typically leads to decreased software quality, i.e., poor comprehensibility, maintainability and reusability.

The production methods 2. to 4. provide an alternative solution to the software quality problem. They restrict the expressiveness to constructs necessary for the domain and hence foster a high comprehensibility. The productions here are based on Domain Specific Languages (DSLs) (see section 5). The DSLs themselves benefit from the semantic grounding of all major constituents involved. Because the building blocks of the DSLs are generated from the ontology in prior, the syntax check–to some extent–simultaneously checks for semantical correctness at compile time. In addition all Scala-capable IDEs provide proper highlighting and auto-completion while editing.

Figure 2 illustrates an excerpt from the processing mesh of a demo application. In SiXton's Curse [3], the user plays a wizard capable of multimodal (speech/gesture) spells. One of the spells in the wizard's arsenal is a protective shield summoned by "*[Move Both hands up] create shield guard*". Movements are tracked by a 6-DOFs (degrees of freedom) sensor and sent to the application via VRPN (Virtual-Reality Peripheral Network). Speech input is captured using the Sphinx speech recognition software. Two components decouple the sensing equipment from the multimodal processing. The input layer of the processing mesh monitors hand and torso movements as well as speech tokens. The intermediate layer consists of two alternative classification methods detecting the occurrence of both hands being above the head. While the neural network processor has learned the variance of the input during training, the fuzzy pattern matcher deals with variance explicitly. Speech tokens are fead into and parsed

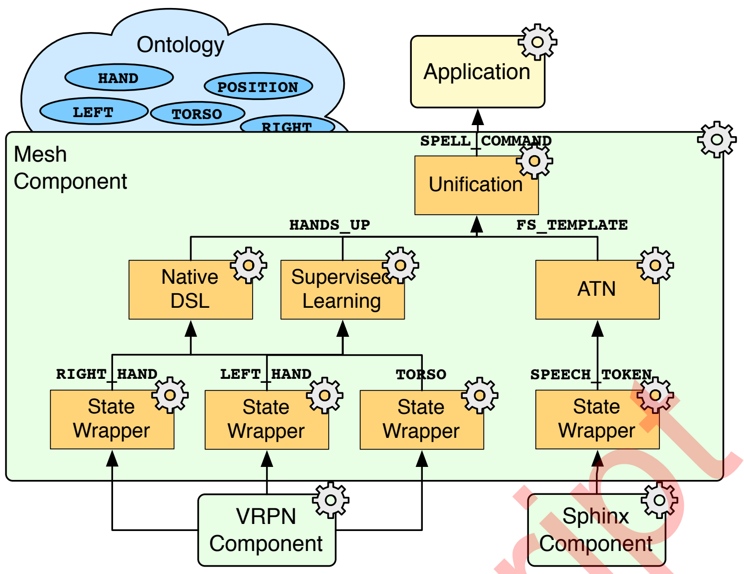

Fig. 2. An excerpt from the processor mesh of a demo application. Input is concurrently processed on two paths with dedicated processors for gesture and speech and finally integrated to form an interaction command provided to the application. Details see text.

by an augmented transition network (ATN). The intermediate layers produce feature structures as input for the multimodal fusion performed by a generalized unification approach. Finally, an application actor registers for changes of the fusion's output, allowing reactions to processed commands.

4 Customizable Operators and Functors

Processing of multimodal utterances has to account for variance even on the level of atomic operations. For example, unification parsers are based on the combination of compatible grammar descriptions called *feature structures*. These structures consist of feature-value pairs or they reference other feature structures recursively. Feature-value pairs represent grammatically relevant aspects like *gender*, *number*, *case*, or–often used in multimodal grammars–*time*. At the lowest level, unification applies a pairwise comparison of features to check if they match, e.g, to check if two parsed words like an article and following noun agree on the *gender* or if the article and the gesture agree on the *time*. Such an agreement is used to define a semantic relation or even an association important for the semantic analysis.

Agreement between matching features is achieved if either only one of the compared features has a value, or if the two compared features have the same value. The *agreement* is the central operator for unification parsers. It relies on equality checks for the atomic feature-value pairs. However, this strict equality does not account for variance: For instance, while a strict syntax check for *gender* or *number* is conceptually sound, real-world utterances often disobey the strict rules of grammar. A similar problem arises for equality checks of *time* features. Technically, strict equality of temporal occurrences is as good as impossible to achieve given a) the independent sample cycles of the underlying hardware, from tracking systems to speech recognition, and b) the internal variances caused by float point representations. In addition, temporal correlations between multimodal streams are observed not to be absolutely precise. If anything, these correlations are better be modeled based on imprecisely delimited time ranges or by some probability distribution.

In summary, comparisons play an important role during multimodal processing. Different types of comparisons are necessary due to inherent or explicit variance. These types could, of course, be programmed individually using tailored code fragments at the appropriate steps, again leading to code hard to maintain with domain-specific parameters scattered around. As a solution to this problem we exploit *functor objects*. Functors are highly customizable. They encapsulate user-defined operations and they make these operations available as first class objects inside of the source code. Functors inherit the type checking feature of the programming language in use, thereby safeguarding against misusage during compile time. In addition, definition and use of functors is now decoupled. This allows developers to build clean and hence maintainable code which, e.g., locates functor definitions at one central place.

A specific processing algorithm like unification now uses functor objects defined like `equals(type_a a, type_b b)` instead of the built-in comparisons of the programming language. These functors are provided for all necessary parameter types and their specific comparison semantic. The choice of the correct functor is automatically performed by the type system, the underlying processing algorithm remains plain and universal. Finally, certain programming languages provide options for an alternative syntax which allows functors to be used as operators. For example, a required fuzzy match functor can be written in Scale like this: `<exp> approximates <exp>`. This feature is highly convenient for the definition of human-readable and comprehensible code, i.e., for the design of domain specific languages as described in the next sections.

5 DSL-supported Feature Processing

The production methods 2. to 6. (see section 3) provide domain specific languages to specify user-defined operations. The constituents of the DSLs are semantically grounded into a common ontology. DSLs are checked for syntactic and (partly) for semantic correctness at compile time. At runtime, terms and expressions are substituted by corresponding implementations of the functors and operators and the resolution result of grounded symbols. Listing 1 defines a set of variables (the left-hand side) used throughout the following examples. The variables further shorten reoccurring access to prominent concepts from the underlying semantic layer.

```
val LEFT_HAND  = Position of (left, hand)
val LH_REL     = LEFT_HAND relativeTo (Transformation of torso)
val LA_LEN     = Length of (left, arm)
val HANDS_UP   = Occurrence of (hands, up)
```

Listing 1. Definition of semantically grounded feature descriptions required for the examples. Definitions for torso, elbow, and shoulder as well as for the right side are implemented analogously.

Concepts from the ontology are generated into corresponding static classes (e.g. `Position` or `Length`) or variables (e.g. `left` or `hand`) and are combined to describe features [19]. In addition, relations between concepts are used to generate corresponding functions that are used to describe processing instructions (e.g. `relativeTo`). The implementation of these generated functions is application dependent and can be used to cope with the variance of multimodal utterances similar to the functor objects in section 4.

5.1 Unrestricted Native Processing

Listing 2 illustrates a DSL-free code snippet for a native approach to detect `HANDS_UP` as motivated in the initial example.

```
1  val handsUpEventDescription = EventDescription( hands :+ up,
2    hasToContain = Time :: HANDS_UP :: Nil )
3  val localCps = Map[SVarDescription, Vec3]()
4  EntityDescription(VRPNTarget(trackerUrl, id="8")).realize(e: Entity => {
5    val positionStateVariable = e.get(Position)
6    positionStateVariable.observe(newValue => {
7      localCps += (LEFT_HAND, newValue); produceFeature()})
8  }//... RIGHT_HAND is processed analogous.
9  def produceFeature() {
10   if(localCps.contains(LEFT_HAND) && localCps.contains(RIGHT_HAND))
11     handsUpEventDescription.emit(
12       Time(System.currentTimeMillis()),
13       HANDS_UP(localCps(LEFT_HAND).y > 2. && localCps(RIGHT_HAND).y > 2.)
14 }
```

Listing 2. Native processing of multimodal input (no DSL). Relative coordinates (e.g. to the torso), or user-specific variance (e.g., the arm length), temporal variances or time series management are missing. Details see text.

Processing starts with the declaration of the resulting event's description (lines 1-2). Then, a data structure for an actor-local copy of the most recent hand positions (`localCps`) is set up. Next, an entity which wraps access to tracking data is created (`realize`) and equipped with an individual initializer (lines 5-7). This handler sets-up monitoring for the entity's representation of the hand positions. It registers the executing actor as an observer (`observe`, line 6) which buffers new positions in `localCps` (line 7). If new position data for the right and the left hand is available (line 10), a resulting event is emitted. The event contains the current timestamp (line 12) and the result of a pre-defined, hard-coded template match using absolute coordinate axis and values (line 13).

Note that neither individual variances, e.g. hand positions relative to torso and the user's actual arm length, nor temporal variances based on time series management are considered for the sake of simplicity. Still, this example illustrates typical deficiencies arising from missing decoupling. It widely uses concepts from the underlying software framework which makes it hard to understand for non-experts and which complicates portability and hence maintenance. In addition, it hard-codes aspects sensitive to variance which makes it hard to customize and reuse the developed products.

5.2 DSL-supported Processors

The first step to revise the example decomposes the overall task into (1) a layer wrapping the raw sensor data (listing 3, lines 1-4), (2) a layer which transforms absolute coordinates into relative measures (listing 3, lines 5-10), and (3) a layer performing the actual detection (listings 4 to 6). Listing 3 first describes the state wrapper which provides the position data for the user's left hand. These processors use a dedicated syntax to wrap sensor access (line 2) but already use a uniform syntax for the specification of the provided feature(s) (line 3). Wrapping of the user's torso, left and right hand, elbow, and shoulder etc. is performed likewise.

```
1  new StateWrapper {
2    Obtained from Id("8") of VRPNSource(trackerUrl)
3    Produces feature LEFT_HAND
4  }//... RIGHT_HAND, TORSO, LEFT_ELBOW, and LEFT_SHOULDER analogous.
5  new NativeDSL {
6    Requires features (LEFT_HAND, LEFT_ELBOW, LEFT_SHOULDER)
7    Produces feature LA_LEN as
8      (Length of (LEFT_ELBOW - LEFT_SHOULDER)) +
9      (Length of (LEFT_HAND - LEFT_ELBOW))
10 }
```

Listing 3. Implementation of a state wrapper for tracking data and a native DSL processor calculating the length of the user's left arm.

A specific subset of the wrapped input data is then processed and transformed into relative measures. Lines 5-10 illustrate the production of the new feature `LA_LEN`, the length of the user's left arm. The DSL inside of the productions consists of identifiers which map to concepts and relations from the ontology, or, to be more specific, to auto-generated classes and static variables establishing references to the ontology. The Scala programming language supports the DSL syntax by allowing the definition of operators and thus ease the common handling with vectors in this domain (lines 8 and 9). As can be seen, this code already increases comprehensibility significantly.

The first example to build a HANDS_UP-processor at the detection layer is illustrated in listing 4. It uses a fuzzy template match. The processor requires the hands' positions, the torso, and the length of the left arm (assuming both arms to be approximately equal in length). The required features LEFT_HAND and RIGHT_HAND are accessed at lines 4 and 5 as LH_REL and RH_REL via functors performing relative transformations.

```
1 new NativeDSL {
2   Requires features (LEFT_HAND, RIGHT_HAND, TORSO, LA_LEN)
3   Produces feature HANDS_UP as
4     ((Height of LH_REL since Time(500)) approximates LA_LEN) and
5     ((Height of RH_REL since Time(500)) approximates LA_LEN)
6 }}
```

Listing 4. A native DSL processor uses a fuzzy match to detect the hands up gesture.

The production defines a conjunction of two fuzzy matches for the left and right hand. The fuzzy matches account for temporal variance using the since operator provided by the implicit time series management available for all features. In addition, they account for individual variance using the predefined functor approximates. This functor is application dependent and auto-generated from the ontology. Finally, the DSL contains additional identifiers providing access to the the ontology, e.g. to resolve the upward direction necessary to determine the Height of a position.

The second example to build a HANDS_UP-processor is illustrated in listing 5. It uses an ATN. In addition to the already described DSL-concepts, it includes ATN-specific identifiers to set-up individual networks consisting of named states, named and directed transitions (arcs), and conditions guarding these transitions. Supplementary, some ATN-specific properties can be set, like an automatic reset of the ATN after 500 ms of receiving no new input.

```
new ATN {
  Requires features (LEFT_HAND, RIGHT_HAND, TORSO, LA_LEN)
  Create StartState 'start withArc 'lhRaised  toTargetState 'lhUp
                           andArc  'rhRaised  toTargetState 'rhUp
  Create State       'lhUp withArc 'rhRaised  toTargetState 'end
  Create State       'rhUp withArc 'lhRaised  toTargetState 'end
  Create EndState    'end
  Create Arc 'rhRaised withCondition
    {Height of RH_REL approximates LA_LEN}
  Create Arc 'lhRaised withCondition
    {Height of LH_REL approximates LA_LEN}
  Set autoReset to Time(500)
  Produces feature HANDS_UP(true)  onEntryOf 'end and
                   HANDS_UP(false) onEntryOf ('start, 'lhUp, 'rhUp)
}
```

Listing 5. A DSL-supported ATN processor detecting the hands up gesture.

The ATN-DSL extends the syntax of the productions by explicitly mapping features to be processed to potential transitions (lines 12 and 13) and hence connects the flows of control of the ATN and the processor. Like in the previous example, the concrete implementations of auto-generated ontology relations, e.g. `approximates`, or the definition of time-related ATN properties, like the auto reset, allow to cope with variance explicitly.

The third example to build a `HANDS_UP`-processor is illustrated in listing 6. It uses a supervised learning approach, i.e., a neural network. In contrast to the ATN-example, the topology of the neural network is not defined explicitly using the DSL since the variability of useful topologies do not vary as much compared to potential ATNs.

```
new SupervisedLearning {
  Requires features (LEFT_HAND, RIGHT_HAND, TORSO, LA_LEN)
  Uses NeuralNetwork "./hands-up.nn"
  Produces feature HANDS_UP
}
```

Listing 6. A DSL-supported supervised learning processor detecting the hands up gesture.

The number of input neurons is defined by the number of required features. Converters automatically map the data types of the required features to an array of floating-point numbers. The number of required features determines the number of neurons in the input layer. A default architecture with one hidden layer and a heuristic to determine the optimal number of hidden layer neurons is applied. The number of output layer neurons is implicitly defined by the number of produced features, again using converters to map an array of floating-point numbers to feature types. For many cases this specification already provides a reasonable number of suitable neural network architectures.

The necessary training and supportive tools are not discussed here, since they do not effect the actual description of the `SupervisedLearning` processor. In contrast to the latter examples, the neural network approach deals with variances implicitly, as they are an intrinsic property of the utilized training set or–later on–a part of the trained neural network parameters.

6 Conclusion

This article presented four software techniques which enhance the overall quality of multimodal processing software as motivated by [16, 12] and own preceding work. The techniques target reoccurring multimodal processing tasks, specifically taking into account variance, e.g., due to individual characteristics and cultural context.

The first technique decomposes the overall task into smaller units dedicated to well-arranged and well-defined sub-problems. While this approach is similar to common functional decomposition, the implementation of these units as actors provides concurrent execution schemes and distribution facilities adequately matching the inherent concurrency of multimodal utterances.

The second technique uses functors as a means to weaken the negative impact of often hard-coded constraints dealing with variance. Using functor-objects, the core algorithms stay plain and universal and fundamental features of the underlying programming language like type and syntax check are exploited, which raises the overall code quality.

The third technique uses domain-specific languages to reduce the diversity of user-generated algorithms, idiosyncratic identifiers, and highly individual programming styles. In combination with programming languages which support syntactic variants for method calls, e.g., like Scala, DSLs can conveniently be be expressed and tailored for the application domain.

The fourth technique uses a semantic grounding of the identifiers used for the functors and the DSL tokens in a common ontology. Auto-generated from the ontology, these identifiers match classes, constants, and variables inside of the native programming language. This supports automatic syntax checks, including proper code highlighting in the development environments. In addition, it partly ensures semantic correctness due to the reproduction constancy between the ontology and the corresponding constructs of the programming language. Finally, the ontology binding is highly beneficial during the final semantic and pragmatic interpretation of the multimodal utterances.

All four techniques together provide scalable, customizable, and reusable solutions for reoccurring multimodal processing tasks. They have been implemented inside of miPRO, a realtime-capable processing architecture for multimodal interactions used in Virtual, Augmented and Mixed Reality applications.

References

1. Böhm, K., Broll, W., Sokolewicz, M.: Dynamic gesture recognition using neural networks; a fundament for advanced interaction construction. In: Fisher, S., Merrit, J., Bolan, M. (eds.) Stereoscopic Displays and Virtual Reality Systems, SPIE Conference Electronic Imaging Science & Technology. vol. 2177. San Jose, USA (1994)
2. Bouchet, J., Nigay, L., Ganille, T.: ICARE software components for rapidly developing multimodal interfaces. In: ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces. pp. 251–258. ACM, New York, NY, USA (2004)
3. Fischbach, M., Wiebusch, D., Giebler-Schubert, A., Latoschik, M.E., Rehfeld, S., Tramberend, H.: SiXton's curse - Simulator X demonstration. In: Virtual Reality Conference VR, 2011 IEEE. pp. 255–256 (2011)
4. Fitzgerald, W., Firby, R.J., Hannemann, M.: Multimodal event parsing for intelligent user interfaces. In: Proceedings of the 2003 international conference on Intelligent user interfaces. pp. 53–60. ACM Press (2003)
5. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence. pp. 235–245. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1973)
6. Hoste, L., Dumas, B., Signer, B.: Mudra: A unified multimodal interaction framework. In: Proceedings of the 13th International Conference on Multimodal Interfaces. pp. 97–104. ICMI '11, ACM, New York, NY, USA (2011)

7. Johnston, M.: Unification-based multimodal parsing. In: Proceedings of the 17th International Conference on Computational Linguistics and the 36th Annual Meeting of the Association for Computational Linguistics COLING-ACL. pp. 624 – 630 (1998)
8. Johnston, M., Bangalore, S.: Finite-state methods for multimodal parsing and integration. In: Finite-state Methods Workshop, ESSLLI Summer School on Logic Language and Information,Helsinki, Finland. pp. 74–80 (august 2001)
9. Johnston, M., Cohen, P.R., McGee, D., Oviatt, S.L., Pittman, J.A., Smith, I.: Unification-based multimodal integration. In: 35th Annual Meeting of the Association for Computational Linguistics, Madrid. pp. 281–288 (1997)
10. Kendon, A.: Gesticulation and speech: Two aspects of the process of utterance. In: Key, M.R. (ed.) The Relation between Verbal and Non-verbal Communication (1980)
11. Koons, D.B., Sparrel, C.J., Thorisson, K.R.: Intergrating simultaneous input from speech, gaze and hand gestures. In: Intelligent Multimedia Interfaces. American Association for Artificial Intelligence (1993)
12. Lalanne, D., Nigay, L., Palanque, p., Robinson, P., Vanderdonckt, J., Ladry, J.F.: Fusion engines for multimodal input: A survey. In: ICMI-MLMI '09: Proceedings of the 2009 international conference on Multimodal interfaces. pp. 153–160. ACM, New York, NY, USA (2009)
13. Latoschik, M.E.: Designing Transition Networks for Multimodal VR-Interactions Using a Markup Language. In: Proceedings of the Fourth IEEE International Conference on Multimodal Interfaces ICMI'02, Pittsburgh, Pennsylvania. pp. 411–416. IEEE (2002)
14. Latoschik, M.E.: A user interface framework for multimodal VR interactions. In: Proceedings of the IEEE seventh International Conference on Multimodal Interfaces, ICMI 2005. pp. 76–83. Trento, Italy (October 2005)
15. Latoschik, M., Tramberend, H.: Simulator X: A scalable and concurrent architecture for intelligent realtime interactive systems. In: Virtual Reality Conference (VR), 2011 IEEE. pp. 171–174 (March 2011)
16. Nigay, L., Bouchet, J., Juras, D., Mansoux, B., Ortega, M., Serrano, M., Lawson, J.Y.L.: Software engineering for multimodal interactive systems. In: Tzovaras, D. (ed.) Multimodal User Interfaces, pp. 201–218. Signals and Commmunication Technologies, Springer-Verlag (2008)
17. Väänänen, K., Böhm, K.: Gesture-driven interaction as a human factor in virtual environments – an approach with neural networks. In: Gigante, M.A., Jones, H. (eds.) Virtual Reality Systems. Academic Press (1993)
18. Wagner, J., Lingenfelser, F., Baur, T., Damian, I., Kistler, F., André, E.: The social signal interpretation (SSI) framework: Multimodal signal processing and recognition in real-time. In: Proceedings of the 21st ACM International Conference on Multimedia. pp. 831–834. MM '13, ACM, New York, NY, USA (2013)
19. Wiebusch, D., Latoschik, M.E.: Enhanced decoupling of components in intelligent realtime interactive systems using ontologies. In: Software Engineering and Architectures for Realtime Interactive Systems SEARIS, proceedings of the IEEE Virtual Reality 2012 workshop (2012)