UTIL: Complex, Post-WIMP Human Computer Interaction with Complex Event Processing Methods

Simon Lehmann, Ralf Dörner, Ulrich Schwanecke, Nadia Haubner, Johannes Luderschmidt

RheinMain University of Applied Sciences
Dept. of Design Computer Science Media
Unter den Eichen 5, 65195 Wiesbaden
E-Mail: simon.lehmann@hs-rm.de

Abstract: Due to a rising amount of different input devices as well as multimodal interaction techniques, post-WIMP interfaces and multimodal interactive systems, such as interactive surfaces and VR systems, have seen a rapid development over the past years. However, many of those systems use rather low-level interfaces to deal with user input, leading to systems hard to extend or reuse. Where higher-level abstractions and architectural means are used to foster decoupling between applications and user input processing, completely new frameworks are commonly introduced. We present how an architecture for complex interaction event processing can be realized in large parts by an complex event processing (CEP) engine. We show how to make use of a continuous, stream-based event processing engine in this context, how the concepts of CEP can be used to realize interaction techniques and also how this approach allows for detection, aggregation, fusion and selective retrieval of input events. Also, our performance evaluation shows that the additional levels of abstraction are beneficial for interactive purposes.

Keywords: Post-WIMP; user interfaces; complex event processing; middleware

Acknowledgements: The project in which this work has been conducted was funded by the German Federal Ministry of Education and Research (BMBF), grant number 17043X10.

1 Introduction

Complex interactive systems are of increasing interest among researchers as well as practitioners. Such complex systems include most VR or AR systems, but also the more general field of post-WIMP, multimodal interaction, which is especially considered as important [TGS06]. Development and research concentrates on four different aspects: novel interaction techniques, usability evaluation, realization of hardware systems and software engineering of those systems. While the latter aspect usually has to be addressed in some way during most research projects, the first three aspects are focused on predominantly. However, to finally make proposed interaction techniques and hardware solutions available for application software development beyond prototypes and proof-of-concepts, the aspect of abstracting and factoring out input (event) processing has to be addressed.

The exact characteristics relating to the inputs and events within an interactive system certainly depend on the use cases it is made for, but there are general characteristics which can be found in most, if not all of them. Input events are short lived, happen at a single point in time (as opposed to happening over a time span), are stateful, carry positional/spatial information and might be tied to a specific user/person. Simple events are produced by input devices/subsystems (e.g., touch panels, depth cameras, speech recognizers) and typically occur at different rates, but not on isochronous time slots. A user interface layer of an application needs access to higher level information, e.g., a user tapped a finger on the screen, as well as lower level information, such as the changing position of finger while it moves across the screen.

An input event processing layer has to provide facilities for detection of interaction performed by the user, aggregation of potentially noisy or high frequent input events to more stable or low frequency input events, fusion of uni- or multi-modal input to provide additional user input events, and selective retrieval of event instances that are of interest to the application/user-interface. In addition to those functional requirements, interactive systems also impose the very important non-functional requirement of low latency. In order to provide a usable middleware which fulfills all of the above, the impact it has on the latency of a user interface may not exceed a threshold.

While there have been several very domain specific approaches to this problem in the field of HCI (e.g. [NC93, SJM+97, ERMS03, JKR09, SHSD11, HDS11]), a more general approach has been developed: complex event processing (CEP). Complex event processing means the detection, analysis, and general processing of correlated raw or simple events, which results in more abstract and meaningful complex events [Luc02]. It builds upon the concept of the event driven architecture (EDA), in which loosely coupled components communicate by emitting and consuming events [LS11]. However, the current research in the field of CEP and EDAs has not yet identified interactive systems as a potential field of application [HSB09].

There also exist several ready to use software libraries and frameworks for CEP purposes. One of them is the software library *Esper* [Esp12a], which provides a general purpose component for CEP in Java or .NET. It is based on the principle of *continuous queries* [TGNO92], where processing, filtering and dissemination of events happens by querying the engine, which continuously matches and processes events according to the active queries. Queries are formulated using the *event processing language* (EPL), which is an SQL-like declarative language supporting selection of events from streams, conditions, sliding windows (time and length based), aggregation functions, joins, pattern matching, insertion into event streams and other features known from relational databases.

This paper presents the user input middleware for interactive systems, called UTIL, that is based on the interaction event processing architecture [LDS+10] and the CEP engine *Esper*. It applies CEP to interactive systems and provides an abstraction between user interface code of an application and the input devices of a system. It offers several benefits:

- Detection, aggregation, fusion and selection of input events is based on a declarative specification (using the Event Processing Languange (EPL)). This declarative approach

allows to specify interactions like gestures more abstractly and lets the CEP engine perform the actual processing and maintenance of current and past events. This abstraction also makes it possible to benefit from optimizations and other improvements the engine can do without having to change the applications code.

- Continuous queries enable applications to precisely specify which kinds of events are expected in the same declarative manner mentioned before. This allows to reduce the amount of procedural post-processing within user interface code to a minimum, thus possibly improving the software quality.

- Allows very high event throughput and low latency, as the CEP engine is specifically designed for continuous event stream processing. It employs well tested and implemented algorithms and optimization techniques and any future improvement will be immediately available to our middleware.

- As CEP solutions are also used in non-interactive scenarios such as context event processing in ambient assisted living environments, integration of interactive systems in such scenarios is aided by a common architectural and technological base [LSDS12].

The remainder of this paper is organized as follows: First, we give a review of the related work. Then, a detailed description of the UTIL middleware is given, followed by a discussion of its properties and how actual interaction scenarios can be realized. Following this, the results of a preliminary performance evaluation are presented. Finally, we give a conclusion and outline potential future work.

2 Related Work

The general topic of our work – the application of CEP to interactive systems – has not received much attention so far. The CEP research community does not consider it as a potential application area [HSB09], while in current HCI research, some CEP-like approaches are taken, such as the Midas/Mudra system [SHSD11, HDS11]. The integration of a stream based CEP engine into the interaction event processing of complex interactive systems has not been proposed so far, but similar work and approaches can be found in the fields of multimodal user interfaces and complex event processing.

One of the earlier attempts to describe an architecture and concepts similar to CEP in the context of multimodal user interfaces can be found in [NC93]. While this work provides useful methods for classifying multimodal systems in general and also provides an architectural approach how such systems could be realized, it remains on a very conceptual level. There is no indication of how it would actually be realized.

The CAIP Center [SJM+97] is meant to provide a multimodal human-computer interaction system that focuses on several pre-defined in- and output modalities, such as tactile input with force-feedback, speech recognition, or gaze tracking. The proposed architecture

is very focused on the specific setup described in their work and is more concerned about the application specific fusion of inputs.

Flippo et al. created a framework for rapid development of multimodal interfaces [FKM03]. Within their framework, they also had to deal with processing input from different devices. They used an agent-based approach for analyzing all input modalities to find out what the user actually wants to do. While this might be a general approach (like the Open Agent Architecture [MCM99] or the Adaptive Agent Architecture [KC00]), their work focuses on speech input that is accompanied by other types of input to resolve ambiguous input situations. Even though their approach seems to be well suited for the specific setup centered at speech input, it is questionable if this can be transferred to other kinds of setups.

Architectural approaches focusing on tabletop systems are ZOIL and Squidy [JKR09]. While the former is an UI toolkit for building tabletop applications, the latter deals with the actual processing of input events from various input devices. Their approach features a graph based event filter system, which processes the incoming events. While such a system works on a lower level of event processing and can be even used for the actual input tracking, it does not address the needs of more complex event processing and it lacks an expressive, yet concise specification language.

The Midas/Mudra system [SHSD11, HDS11] is probably the most similar to the approach described in this paper. It supports the processing of multiple streams of events ranging from low-level, simple raw data streams to higher-level, complex semantic events. It employs a fact base and also a declarative language to specify how facts (representing events) should be processed. While it also aims for a working, concrete realization of a complex interaction event processing abstraction, it has several drawbacks. First, a fact base is used for maintaining information about events which have happened. Input events are continuously added to and removed from the fact base, which is not designed for this kind of processing. As input events are usually very short lived and transient in nature this approach requires additional maintenance functions which have to be added to the fact base. Second, the processing rules are realized using the inference engine CLIPS [SCR+13], an expert system tool. The inference engine is also not designed for continuous evaluation and thus has to be also modified in order to support this processing model. Third, the interface to the application layer does not allow to select events based on application defined rules (such as complex truth expressions or patterns). This is also neglected by other approaches, even though it is important in order to be able to deal with increasing amounts of events on different abstraction levels. Lastly, as the performance evaluation of those systems shows, this approach does not yield acceptable performance. All these drawbacks lead to the conclusion that, while the architectural approach is valid, the Mudra system does not provide a workable solution.

3 The UTIL Middleware

Our middleware is based on the architecture described in [LDS+10], which on a high level distinguishes between four layers: devices, event processing, interaction and the application

(see figure 1). The device layer contains all the input devices with their specific interfaces and event models. The application layer contains all application specific components, such as algorithms, business logic, storage and communication mechanisms, but also the application specific parts of the user interface. The middleware is located between those layers and consists of both the event processing as well as the interaction layer. As already noted earlier, the core functionality of the middleware is provided by the the CEP engine Esper. This also determines Java as the platform the middleware is implemented on, which we assume is well suited for implementing research projects as well as real world application software. The platform also impacts some aspects of the middleware, such as how events are represented and how modularization and extension is realized.

Events are represented as objects which have different attributes depending on the type of the event. Event types are realized as normal Java classes, which might form a hierarchy, but this is not necessary for the middleware to work (there is no common base class every other class has to be derived from). This representation is also directly usable by Esper. The benefit of using classes and instances thereof to represent events is the strong type checking the platform can provide at compile- and run-time throughout the middleware, especially to the user interface part of the application. With a dynamic representation (e.g. map-based), all type information is lost as soon as the event leaves the middleware. Application code using the object would have to rely on informal documentation to know which attributes exist and what types they have.

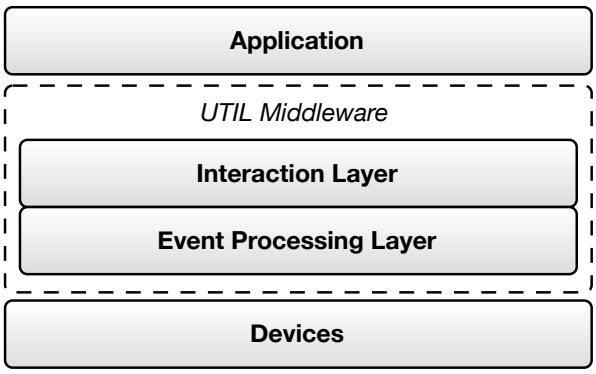

Figure 1: The general architecture of the UTIL middleware.

3.1 Event Processing Layer

The event processing layer is responsible for receiving input events produced by the input devices, applying any processing rules registered with the middleware and providing all events (raw or processed) to the interaction layer (see figure 2 for an overview).

To allow all kinds of devices to be connected to the middleware, it offers an extensible mechanism for event collection. Each input device is wrapped by a *collector* which is specific to the type of input the device provides. The collector implements any kind of native interface necessary to communicate with the device and creates the appropriate event instances (if necessary). All events coming from the independently running collectors are put into a queue, which is then processed sequentially by the CEP engine. Collectors can make new event types known to the middleware at startup or even during runtime, which allows to incorporate completely new types of input which have not been thought of yet. Of course, to allow applications to actually receive input events, the type of those events has to be known

to the application code as well. However, defining a fixed set of event types or some event hierarchy is beyond the scope of this paper.

Applying processing rules to any events that enter the processing layer is its core functionality. This is entirely realized by the CEP engine, which is based on streams and continuous queries (or more general statements). Queries are written using the EPL and at an absolute minimum specify which streams or event types they are interested in. As there are no external observers of events in this layer, it is necessary to turn a query into a rule-like construct which produces new events (otherwise, the matched events and the computed results would be lost). For this, the EPL offers an INSERT INTO clause, which allows to create new events or pass existing events back into an event stream, which results in the following structure for *rules*:

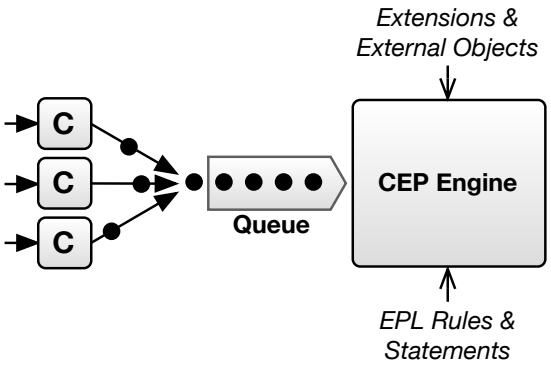

Figure 2: Structure of the event processing layer. Collectors (C) are independently receiving input from devices, the CEP engine asynchronously processes the queued events one by one.

```
1 INSERT INTO name of new event type
2 SELECT list of new event attributes
3 FROM stream specification
```

In general the EPL allows to use many SQL like constructs and expressions, but also event stream processing specific clauses and operators. It is outside the scope of this paper to explain the full EPL, but we will explain a few key aspects in order to make the examples easier to understand[1]. In general, the EPL allows to access all attributes of event objects by specifying their name (e.g., accessing the attribute 'pos' on the object 't' would be written as `t.pos`) as well as calling any methods defined on it. Within the FROM clause, it is possible to use different event stream specifications: filter-based, pattern-based or window/view-based. The first is used for directly matching events based on their type/stream name with optional filter expressions to further narrow down the amount of events it applies to. The second allows more complex matching of event patterns, which might consist of different types of events from different streams and thus also allows to express temporal relations using the *followed-by* operator ($->$). The last is used in combination with the first specification method and allows to operate on a window of events collected over time or depending on user defined conditions. Listing 1 shows an example rule.

The middleware also allows to use arbitrary external functionality from within the engine. Specifically, a query can make external static function calls, create and use arbitrary objects and supports various scripting languages to implement procedural functionality within the engine itself. This way, any kind of external processing mechanism that either already exists or cannot be easily expressed as a declarative query can be integrated. For example, a

[1]For full details about how queries can be formulated refer to the Esper reference manual [Esp12b]

```
1 INSERT INTO TouchHold
2 SELECT
3   current_timestamp(), t.pos
4 FROM PATTERN [
5    EVERY t = Touch(state = Added)
6       -> timer:interval(1 SECOND)
7  AND NOT ( Touch(pos.distance(t.pos) > thresh) OR Touch(state = Removed) )
8 ]
```

Listing 1: Example of a rule-like query, generating a TouchHold event whenever a finger touches the screen longer than 1 second in one place

probabilistic classifier based on Hidden Markov models or Naive Bayes could be implemented in an external library and then a query can call it in a filter expression or WHERE clause to discard events based on their classification.

Events enter the engine one by one and are matched against the stream specifications of all active queries. If an event matches a specification, the query is evaluated and, depending on the outcome of the evaluation, might generate a new event instance of the same or some other type. If an event is generated while evaluating a query, it is immediately processed by the engine before the engine continues to process the original event (this may happen recursively). Events are only retained as long as any query references it, e.g., by maintaining a window of events over time or within a pattern. Thus, when an event is not matched, it will not be retained and cannot be retrieved at a later time.

3.2 Interaction Layer

On top of the event processing layer, which provides the detection, aggregation and fusion of input events independently of the application, the middleware provides the interaction layer. This layer manages the access of applications to any event from the event processing layer. It also provides a general *user interface environment*, in which *interface components* can be created. The interface components represent the user interface of the application and can be extended to provide application specific behavior. They are also the only way to query and receive events from the event processing layer.

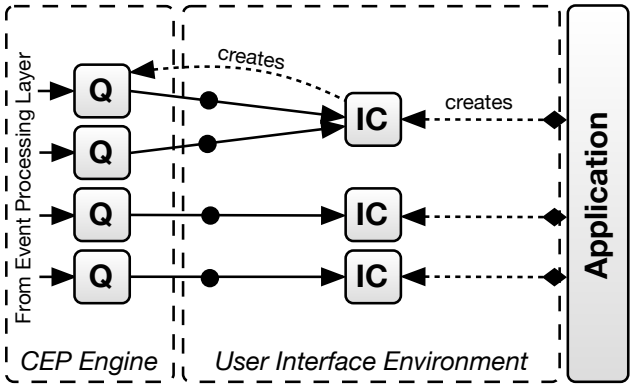

Figure 3: Structure of the interaction layer. A general user interface environment is provided, in which interface components (IC) can be created. The interface components allow to issue queries (Q) to the event processing layer and represent the user interface of the application.

The interaction layer again utilizes the Esper engine to make use of EPL queries as a flexible subscription mechanism. However, as the application does not need all the possibilities of full EPL queries (for example INSERT INTO queries do not make much sense from

the application's point of view), the main querying interface uses EPL *patterns*. When the application requires user input, it creates a new query via an interface component in the interaction layer and gets notified whenever an event matches the query. Listing 2 shows how the application could for example subscribe to all speech events containing the word "show" which is followed by a touch event. This listing also shows how strong typing of event objects is retained throughout the middleware. The event listener infrastructure is making use of generics to provide the actual type to the receiver, which eliminates the need for manual type casting in user defined code.

```
1 object.events.<Speech> on(
2   "EVERY Speech(word = 'show') -> Touch"
3 ).fire(new EventListener<Speech>() {
4     public void receiveEvent(Speech event) {
5       ...
6     }
7 });
```

Listing 2: Example of an event subscription from an application (in Java). 'object' refers to an interface component of the interaction layer.

Only being able to query for events is not sufficient though. In many user interfaces, the state of the application and the user interface itself must be taken into account in order to provide only those events that are of interest with respect to the current state. For example, when a button is shown on a screen, it should only receive those events which actually happened *on* the button. Thus, access to application components or at least interface components of the interaction layer from within a query is necessary. UTIL provides the necessary infrastructure to retrieve the interface component in custom EPL *pattern guards*. Pattern guards are one kind of pattern operators (like the EVERY or the followed-by operator), which control if the pattern expression before it will match or not. Thus, it is not possible to gain full access to the interface component in a query, but by implementing various parameterizable custom guards, it is still possible to filter out events based on the state of the interface component. Figure 4 shows how this approach can be used to filter out touches outside of a button component.

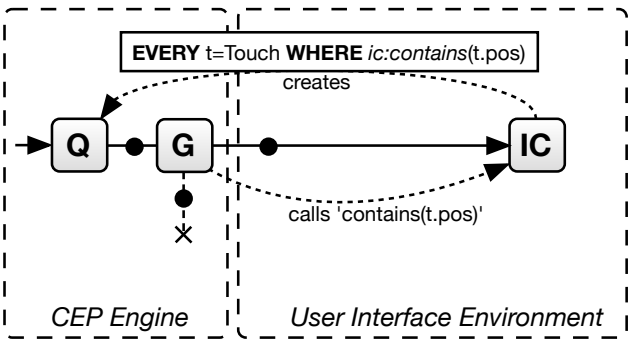

Figure 4: Filtering of events based on the state of interface components via pattern guards. The interface component (IC) issues a query (Q) containing a WHERE pattern guard. This guard calls a method on the IC for every event to determine if an event should be discarded.

Together, the user interface environment and the interface components provide the infrastructure to build user interfaces. However, the architecture and the middleware do not define how interface components are represented to the user, i.e., they are not necessarily

graphical components, which are displayed on a screen. Providing a mapping of input to output modalities or even defining which kinds of output are provided is deliberately left out of our middleware. This allows using the middleware in various scenarios with different output modalities.

4 Realizing Interaction Techniques

The middleware introduces a novel way of realizing an applications's user interface, as it centers around the use of continuous queries on event streams. In this section, we illustrate this by two examples: touch-based single tap and Bolt's seminal put-that-there [Bol80].

In the following listing, the implementation of the *single tap* gesture is shown as the first and basic example of how a specific interaction pattern can be realized.

```
1 INSERT INTO Tap
2 SELECT current_timestamp(), ts.pos
3 FROM PATTERN [
4   EVERY ts = Touch(state = Added)
5     -> betw = Touch
6   UNTIL te = Touch(state = Removed)
7 ]
8 WHERE (ts.pos).coincideAll(betw.pos) AND (ts).finishedBy(te, 500 MSEC)
```

This interaction uses touch input as its only modality and can be simply described as a quick tap with one finger on a touch sensitive surface. Even though it is a very simple gesture, it contains several useful techniques offered by the EPL. It makes use of the already introduced PATTERN matching: every touch event that signals a newly detected finger (line 5), the engine will wait for and collect all following touch events until a touch event occurs that signals that the finger has been removed (lines 6-7). Every event or group of events is assigned to an identifier (ts, betw and te), which is then used in a WHERE clause to only report a tap event if all touches occurred in the same place (line 10) and the whole gesture didn't take longer than 500 milliseconds (line 11). The former test is performed by a user defined method on the position attribute (pos), the latter makes use of the built-in time interval algebra methods.

The second example implements Bolt's put-that-there interaction. We chose this example because it is a well-known multimodal interaction technique and involves more complex user interaction. However, as the following listing shows, the implementation is quite similar and not much more complex than what was needed for the single tap gesture.

```
1 INSERT INTO BoltInteraction
2 SELECT current_timestamp(), thatp, therep
3 FROM PATTERN [
4   EVERY put = Speech(word = 'put')
5     -> (that = Speech(word = 'that') AND thatp = Point)
6     -> (there = Speech(word = 'there') AND therep = Point)
7 ]
8 WHERE (thatp).coincides(that) AND (therep).coincides(there)
```

It assumes there are already event streams for speech and pointing events. The pattern is describing the general structure of the interaction expected from the user: first, the engine

waits for the a speech event with the word 'put' to occur. After that, it expects both a speech event with the word 'that' as well as a pointing gesture to occur (note that this does not yet constrain these two events to happen at the same time). Finally, the same combination of speech and pointing events is expected again, only with the word 'there'. If this course of interaction takes place, the whole pattern matches and the WHERE clause is evaluated. There, the temporal constraints are specified: it ensures that the pointing gestures happened at the same time as the corresponding speech events (lines 12-13).

5 Performance Evaluation

As interactive systems have to respond to user input as quickly as possible, the impact on latency of any processing between the input devices and the application has to be considered. As our middleware makes extensive use of the CEP engine Esper, its performance characteristics will depend mostly on those of Esper. From the Esper documentation on performance [Esp12b] it becomes clear that the algorithms and optimization techniques used in the engine do allow for very high rates of events with only minimal impact on latency. Esper is able to process over 500 000 event/s in those benchmarks, with latency below 3 µs on average.

In order to verify that these characteristics are also applicable to the UTIL middleware, we designed and ran our own performance evaluation. In order to get close to a 'typical' usage scenario, we used a recording of 4384 raw input events from a multi-touch enabled interactive tabletop system and configured the middleware with 27 rules. Those rules were actual implementations of various interaction techniques, among them also those found in this paper. As we only wanted to measure the impact of the middleware itself, but still wanted to include the impact of the interaction layer event delivery, we added 1000 unspecific queries for all events on the interaction layer and used one empty event listener for each. The whole setup was automated and repeatedly executed. For each raw event passed into the middleware, the time was measured it took the middleware to accept the next event, which we call the *roundtrip time*. This time includes all query evaluations as well as calling all registered event listeners. As this does not capture the time it takes until the *first* listener is notified (which is closer to what would be expected in real-world usage), this measurement results in an upper bound to what processing times can be expected.

The tests were ran on a notebook computer, precisely a MacBook Pro with an 2.66 GHz Intel Core 2 Duo with 4 GB of RAM. The measurements resulted in an average roundtrip time of 12.35 µs per event and event listener with a standard deviation of 4.8 µs. This results in an average event throughput of 80 973 event/s. These numbers are certainly lower than what is found in the quoted performance evaluation above. However, we have to take into account that our middleware cannot make use of concurrent processing. Still, the latency and event throughput can be considered sufficient for interactive systems, as input event rates are typically ranging from hundreds to a few thousand events per second.

6 Conclusion & Future Work

We presented UTIL, a middleware based on an interaction event processing architecture. It introduces the established principles and methods of complex event processing into the domain of interactive systems. The middleware makes use of the existing tools of the CEP domain and we illustrated, how the middleware enables interactive applications to leverage the declarative query language to implement existing or new interaction techniques in an application independent form. We also shown, how the same query language provides a flexible means to subscribe to interaction events from an application.

As future work we see essentially two areas that could be explored further. First, the uncommon way of implementing interaction using the EPL certainly requires additional tooling specifically tailored to the needs of interaction designers, as well as finding common design patterns that aid the development of interaction techniques. Second, the middleware does not specifically handle uncertainty and probabilistic event types. While it is possible to make use of probabilistic classifiers via external objects and add attributes to events that store probabilities, a native support of those concepts could have some advantages.

References

[Bol80] R. A. Bolt. "Put-that-there". *ACM SIGGRAPH Computer Graphics*, 14(3):262–270, July 1980.

[ERMS03] C. Elting, S. Rapp, G. Möhler, and M. Strube. Architecture and Implementation of Multimodal Plug and Play. In *Proc. of the 5th intern. conf. on Multimodal interfaces*, pages 93–100, 2003.

[Esp12a] EsperTech Inc. ESPER - An Event Stream Processing and Event Correlation Engine (Version 4.7.0). `http://esper.codehaus.org`, October 2012.

[Esp12b] EsperTech Inc. Esper Reference (Version 4.7.0). `http://esper.codehaus.org/esper-4.7.0/doc/reference/en-US/html/`, October 2012.

[FKM03] F. Flippo, A. Krebs, and I. Marsic. A Framework for Rapid Development of Multimodal Interfaces. In *Proc. of the 5th intern. conf. on Multimodal interfaces*, pages 109–116, 2003.

[HDS11] L. Hoste, B. Dumas, and B. Signer. Mudra: A Unified Multimodal Interaction Framework. In *Proc. of the 13th intern. conf. on multimodal interfaces - ICMI '11*, page 97, New York, New York, USA, November 2011. ACM Press.

[HSB09] A. Hinze, K. Sachs, and A. Buchmann. Event-based applications and enabling technologies. In *DEBS '09: Proc. of the Third ACM International Conference on Distributed Event-Based Systems*, pages 1–15, 2009.

[JKR09] H. C. Jetter, W. A. König, and H. Reiterer. Understanding and designing surface

computing with zoil and squidy. In *CHI'09 Workshop-Multitouch and Surface Computing*, pages 1–5, 2009.

[KC00] S. Kumar and P. R. Cohen. Towards a Fault-Tolerant Multi-Agent System Architecture. In *Proc. of the fourth intern. conf. on Autonomous agents*, pages 459–466. ACM New York, NY, USA, 2000.

[LDS+10] S. Lehmann, R. Dörner, U. Schwanecke, J. Luderschmidt, and N. Haubner. An Architecture for Interaction Event Processing in Tabletop Systems. In *Self Integrating Systems for Better Living Environments: First Workshop, Sensyble 2010*, pages 15–19. Shaker Aachen, November 2010.

[LS11] D. Luckham and R. Schulte, editors. *Event Processing Glossary - Version 2.0.* Event Processing Technical Society, July 2011.

[LSDS12] S. Lehmann, J. Schäfer, R. Dörner, and U. Schwanecke. Towards integration of user interaction and context event processing in intelligent living environments. In *ARCS Workshops*, volume 200 of *LNI*, pages 111–122. GI, 2012.

[Luc02] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman, 2002.

[MCM99] D. Martin, A. Cheyer, and D. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1):91–128, January 1999.

[NC93] L. Nigay and J. Coutaz. A design space for multimodal systems: concurrent processing and data fusion. In *Proc. of the INTERACT '93 and CHI '93 conf. on Human factors in computing systems*, pages 172–178, 1993.

[SCR+13] R. Savely, C. Culbert, G. Riley, B. Dantes, B. Ly, C. Ortiz, J. Giarratano, and F. Lopez. CLIPS: A Tool for Building Expert Systems. `http://clipsrules.sourceforge.net/`, January 2013.

[SHSD11] C. Scholliers, L. Hoste, B. Signer, and W. De Meuter. Midas: a declarative multi-touch interaction framework. In *Proc. of the fifth intern. conf. on Tangible, embedded, and embodied interaction - TEI '11*, pages 49–56, New York, New York, USA, January 2011. ACM Press.

[SJM+97] A. Shaikh, S. Juth, A. Medl, I. Marsic, C. Kulikowski, and J. L. Flanagan. An Architecture for Multimodal Information Fusion. In *Proc. of the Workshop on Perceptual User Interfaces (PUI'97)*, pages 91–93, 1997.

[TGNO92] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. *ACM SIGMOD Record*, 21(2):321–330, 1992.

[TGS06] E. Tse, S. Greenberg, and C. Shen. Motivating Multimodal Interaction around Digital Tabletops. In *Video Proc. ACM CSCW Conf, Computer Supported Cooperative Work*, pages 6–7, 2006.