

A Declarative Approach for Engineering Multimodal Interaction

Lode Hoste

26th June 2015

Jury Members:

- Prof. Dr. Theo D'Hondt (chair)
SOFT DINF, Vrije Universiteit Brussel
- Prof. Dr. Olga De Troyer (secretary)
WISE DINF, Vrije Universiteit Brussel
- Prof. Dr. Beat Signer (promoter)
WISE DINF, Vrije Universiteit Brussel
- Prof. Dr. Wolfgang De Meuter (promoter)
SOFT DINF, Vrije Universiteit Brussel
- Prof. Dr. Bart De Boer
AI DINF, Vrije Universiteit Brussel
- Prof. Dr. Bart Jansen
ETRO, Vrije Universiteit Brussel
- Prof. Dr. Jean Vanderdonckt
Louvain Interaction Laboratory (LiLab), Louvain-la-Neuve, Belgium
- Prof. Dr. Judith Bishop
Director of Computer Science, Microsoft Research, Redmond, USA

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel / Fax : +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN 9789492312006
NUR 989

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, print, photoprint, microfilm, electronic, mechanical, recording, or otherwise, without the prior permission from the author.

Alle Rechten voorbehouden. Niets van deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook, zonder voorafgaande schriftelijke toestemming van de auteur.

Copyright © 2015 Lode Hoste

Abstract

The communication between human and machine is rapidly changing with the introduction of new commodity hardware, such as Apple's iPad, HP's Sprout, Microsoft's PixelSense and Kinect. This hardware embeds novel input sensors to facilitate a more natural user interaction (NUI) paradigm. The development of NUI applications, where the machine tries to understand and anticipate the user's interaction, typically relies on a continuous monitoring of multiple input channels. The collection of events, the detection of relevant patterns and the embedding of these concerns into the application engenders significant challenges because relevant information is hidden in a continuous stream of events. Moreover, the implementation of the detection process in imperative programming languages is excessively difficult.

In this dissertation we present novel programming abstractions to describe multimodal interaction patterns. Our approach consists of two major efforts: a programming language and a compatible runtime platform with an extensible architecture. The first effort consists of a domain-specific language, called Midas, which allows developers to express their multimodal tasks in a declarative manner. A declarative programming style allows the programmer to think about *what* the fundamental conditions are, instead of analysing *how* to process input events one by one, as would be necessary with an imperative language. Midas uses declarative rules to express multimodal interaction patterns. These conditions rely on the existence and the spatio-temporal relation of input events that were obtained from various input modalities. Midas provides adequate programming abstractions to help developers express these conditions in a modular and composable manner.

Midas programs are interpreted by Mudra, an efficient multimodal interaction architecture and processing engine. Mudra is centred on a global information storage, called the fact base, which is populated by multimodal input events from various devices. As these events arrive in a

continuous manner, rules and other processes actively react to changes in the fact base. In order to do this efficiently, Mudra progressively filters and combines facts in order to derive a conclusion. Our high-level Midas programming language and its efficient Mudra runtime platform allows developers to fuse information across the data-level, feature-level and decision-level.

We have successfully deployed our solution in the real world, including live programming sessions and live music performances. Using the programming abstractions presented in this dissertation, we foresee the rapid prototyping of a whole range of novel natural user interfaces in a modular and composable manner.

“Knowledge not shared, is wasted.” - Clan Jacobs.

Samenvatting

Innovatieve apparaten zoals Apple's iPad, HP's Sprout, Microsofts PixelSense en Kinect bieden nieuwe manieren om te interageren met computers ten opzichte van de klassieke toetsenbord- en muisopstelling. Deze hardware bevat een groot assortiment van sensoren om een meer natuurlijke interactie (NI) met de gebruiker te bekomen. NI toepassingen worden gekenmerkt door het begrijpen van en anticiperen op de interactie van de gebruiker en vertrouwt typisch op een voortdurende analyse van meerdere invoermodaliteiten. Het verzamelen van gebeurtenissen, de detectie van relevante patronen en de inbedding van deze interpretatie in toepassingen brengt echter grote uitdagingen met zich mee, omdat de relevante informatie verborgen zit in een continue stroom van gebeurtenissen. Bovendien is het programmeren van deze analyse in imperatieve programmeertalen buitengewoon complex.

In dit proefschrift introduceren we nieuwe programmeer abstracties om multimodale interactiepatronen eenvoudig te beschrijven. Onze aanpak bestaat uit twee grote inspanningen: een programmeertaal en een bijhorend uitvoerplatform met een uitbreidbare architectuur. Ten eerste introduceren we een nieuwe domein-specifieke taal, genaamd Midas, waarmee ontwikkelaars multimodale interactiepatronen kunnen uitdrukken op een declaratieve manier. Het gebruik van een declaratieve programmeerstijl laat de programmeur toe om te denken over *wat* basisvoorwaarden van de interactie zijn, in plaats van *hoe* gebeurtenissen één voor één moeten worden afgehandeld, zoals noodzakelijk is met imperatieve programmeertalen. Midas maakt gebruik van declaratieve regels om multimodale interactiepatronen uit te drukken. Regels bevatten voorwaarden die uitgedrukt worden aan de hand van tijd en ruimtelijke relaties tussen gebeurtenissen die verkregen werden uit verschillende invoer modaliteiten. Midas biedt de nodige programmeer abstracties om deze voorwaarden uit te drukken op een modulaire en compositionele manier.

Midas programma's worden geïnterpreteerd door Mudra, een efficiënte multimodale interactie architectuur en uitvoermachine. Centraal in Mudra staat een feitenbank die multimodale invoergebeurtenissen bevat. Bij elke wijziging in deze feitenbank zal de uitvoermachine elke regel aftoetsen. Om de nodige efficiëntie te bekomen, gebruikt Mudra een incrementeel algoritme waarbij de combinatie tussen feiten in een tijdelijke opslag wordt bijgehouden. De combinatie van onze hoog-niveau Midas programmeertaal en de efficiënte Mudra architectuur stelt ontwikkelaars in staat om informatie over data-, feature- en besluitvormingsniveau te fusioneren.

We hebben met succes onze oplossing toegepast in de echte wereld, onder andere voor live programmeer sessies en live muziek optredens. Met behulp van de programmeer abstracties gepresenteerd in dit proefschrift, kan men snel prototypes ontwikkelen die modulair kunnen worden opgebouwd waardoor er een hele reeks van nieuwe applicaties met natuurlijke interactie mogelijk wordt.

Table of Contents

1	Introduction	1
1.1	Research Context	3
1.2	Research Goals	5
1.3	Methodology	7
1.3.1	Language-oriented Approach	7
1.3.2	Architecture and Execution Engine	8
1.3.3	Towards a Solution	9
1.4	Contributions	10
1.5	Supporting Publications and Demonstrators	11
1.5.1	Publications	11
1.5.2	Demonstrators	15
1.6	Dissertation Outline	16
2	Multimodal Interaction	19
2.1	Multimodal Concerns	19
2.2	Multimodal Fusion Levels	21
2.2.1	Data-Level Fusion	21
2.2.2	Feature-Level Fusion	22
2.2.3	Decision-Level Fusion	22
2.3	Criteria for Expressing Multi-Level Multimodal Fusion	23
2.3.1	Language Features	24
2.3.2	Multimodal Processing Concerns	26
2.3.3	Multimodal Disambiguation	32
2.3.4	Accessibility and Tooling	35

2.4	Conclusion	36
3	Related Work	39
3.1	Data Streams and Semantic Inferencing	39
3.1.1	Data Stream-oriented Solutions	40
3.1.2	Semantic Inferencing Solutions	40
3.1.3	Irreconcilable Approaches?	43
3.2	Positioning of Related Work	43
3.3	Multimodal Languages	47
3.3.1	Data-level Gesture Languages	47
3.3.2	Gesture Authoring	59
3.3.3	Template Matching and Machine Learning	60
3.3.4	Decision-level Multimodal Languages	62
3.4	Multimodal Architectures	67
3.4.1	Data Stream-oriented Architectures	67
3.4.2	Semantic Inferencing Architectures	70
3.5	Conclusion	73
4	Midas: A Programming Language for Multimodal Inter- action	75
4.1	A Declarative Language	76
4.1.1	Formal Grammar of Midas	76
4.2	Interpreting Midas	76
4.2.1	Templates, Modules, Facts and Events	78
4.2.2	Rules with Conditional Elements, Tests, Attempts and Functions	79
4.2.3	Rules with Modifiers	82
4.2.4	A Midas Implementation of the Hold-and-Rotate Gesture	83
4.3	Multimodal Language Features	84
4.3.1	Modularisation and Abstraction	84
4.3.2	Inheritance as Composition of Modules	88
4.3.3	Customisation and Extensibility	89
4.3.4	Negation	92

4.3.5	Application Symbiosis	92
4.3.6	Unbound Variables and Unification	94
4.3.7	Event Expiration	95
4.4	Data-level Fusion	96
4.4.1	Spatial Specification	97
4.4.2	Temporal Specification	100
4.4.3	Spatio-Temporal Specification	100
4.4.4	User-defined Attempts and Functions	100
4.4.5	Identification and Grouping	101
4.4.6	Segmentation and Control Points	103
4.5	Feature-level Fusion	107
4.5.1	Synchronising Streams	108
4.5.2	Dynamic Service Instantiation	109
4.5.3	Asynchronous Tests	111
4.5.4	Verification	112
4.5.5	Cross-level Fusion	113
4.6	Decision-level Fusion	114
4.6.1	Shadow Facts	114
4.6.2	Alternating Between Conditions and Modifiers	116
4.6.3	Conflict Resolution	117
4.7	Multimodal Language Patterns	121
4.8	Developer Feedback	124
4.9	Conclusion	125
5	Mudra: A Unified Multimodal Interaction Architecture	127
5.1	Conceptual Architecture of Mudra	129
5.1.1	Motivating Examples	129
5.2	Mudra's Unified Fusion Architecture	133
5.2.1	The Infrastructure Layer	134
5.2.2	The Distribution Layer	138
5.2.3	The Core Layer	142
5.2.4	The Service Layer	145
5.2.5	The Application Layer	153

5.3	Multimodal Processing Concerns	154
5.3.1	Online Processing	154
5.3.2	Offline Processing	155
5.3.3	Partially Overlapping Matches	155
5.3.4	Segmentation	157
5.3.5	Long Term Reasoning	159
5.3.6	Concurrent Interaction	161
5.3.7	Portability, Serialisation and Embeddability . . .	162
5.3.8	Runtime Definitions and Device Instantiation . .	162
5.3.9	Reliability and Scalability	163
5.4	Authoring Tools	164
5.4.1	Inferencing and Refining Control Points	164
5.4.2	A Graphical Full-Body Development Environment	166
5.4.3	Summary	167
5.5	Compilation and Runtime Model	167
5.5.1	Compilation Flow	167
5.5.2	Midas 2.0 ANTLR Compiler	169
5.5.3	Midas 1.9 Ruby Compiler	169
5.5.4	Midas 1.0 Core Engine	172
5.6	Conclusion	173
6	Midas & Mudra at Work	175
6.1	Midas and Mudra: A Qualitative Evaluation	175
6.1.1	Language Features	176
6.1.2	Multimodal Processing	178
6.1.3	Multimodal Disambiguation	180
6.1.4	Accessibility and Tooling	182
6.1.5	Conclusion	182
6.2	Comparing Software Engineering Abstractions for Mul- timodal Interaction	183
6.2.1	Comparing the Data-Level Language Abstractions of Midas and Proton	183

6.2.2	Comparing the Decision-Level Language Abstractions of Midas and SMUIML	187
6.3	Case Study #1: The Kinect Presenter	190
6.4	Case Study #2: Live Gesture Programming Session	191
6.5	Case Study #3: Declarative Gesture Spotting	194
6.6	Case Study #4: Augmented Live Music Performance	196
6.6.1	Constraints	198
6.6.2	Expressive Control	200
6.6.3	Discussion and Conclusion	205
6.7	Case Study #5: Hand Grip Assessment for Effort Discounting Tasks	205
6.8	Case Study #6: Water Ball Z	206
6.8.1	Electronic Schema	208
6.8.2	Solenoid Valves and Nozzles	209
6.9	Conclusion	209
7	Conclusions	211
7.1	Summary and Contributions	212
7.1.1	Analysis of Criteria, Challenges and Open Issues in Multimodal Fusion Frameworks	214
7.1.2	Midas	214
7.1.3	Mudra	215
7.1.4	Shadow Facts	215
7.1.5	Control Point-based Gesture Spotting	216
7.2	Shortcomings and Future Work	216
7.2.1	Forgiving Interfaces	218
7.3	Overall Conclusion	218
	Appendices	223
A	Terminology in Multimodal Interaction	225
B	Transcript of the Formal Grammar of Midas	229
C	Positioning and Discussion of Related Work	231

D ANTLR Specification of Midas	251
E Reused Attempts and Functions	263
F Built-in Mudra Templates	265
G Compatibility of Criteria Defined by Cirelli et al.	267
H SMUIML XPaint Implementation	271

1

Introduction

Starting in the late sixties, human-computer interaction has shifted from command line interfaces (CLI) to graphical user interfaces (GUI). In recent years this trend is taken one step further by expanding human-computer interaction beyond the typical keyboard and mouse setup in a trend called natural user interfaces (NUI) [8]. A NUI is an interaction methodology which incorporates human skills such as *touch*, *sight* and *body movement* to enable human-computer interaction. Many NUIs rely on interaction patterns that are also used in everyday life. For example, a virtual deck of cards can be dealt by swiping towards players, as if performed with real cards on a table¹. In a similar way, a baseball game with a NUI interface enables players to hit the ball by swinging their arms².

New commodity hardware facilitates the expansion of NUI applications. Devices such as Apple's iPad³, HP's Sprout⁴ and Microsoft's PixelSense⁵ add a new dimension to human-computer interaction because one can touch, move and manipulate virtual digital objects in a natural way. Moreover, physical objects, in the form of tangibles, can be placed on a multi-touch table to initiate interaction. A nice application of such

¹wePoker: <http://wepoker.info>

²Kinect Sports Season Two: <https://marketplace.xbox.com/Product/66acd000-77fe-1000-9115-d8024d5309d6>

³Apple iPad: <https://www.apple.com/ipad>

⁴HP Sprout: <https://sprout.hp.com>

⁵Microsoft PixelSense: <https://www.microsoft.com/en-us/pixelsense>

tangibles is Reactable, which is used to compose music by carefully positioning multiple physical cubes representing musical effects, filters and generators [79].

Input sources, such as the Wii Remote⁶ or Samsung's Smart Remote⁷, allow users to interact with computers by performing arm movement in the air. These remotes embed accelerometers accompanied by optical sensors to capture their movement and direction. Air gestures can also be captured by depth sensors, as popularised by SoftKinetic's DepthSense⁸ and Microsoft's Kinect⁹ in Smart TVs¹⁰ and Xbox consoles¹¹.

The development of NUI applications is hindered by the fact that sensors used for NUIs are typically much noisier than traditional input devices. This means that we need to invest in systems that can receive different streams of information, such as touch, speech, pen and visual signal, in order to obtain effective interpretation of the human's interaction [41, 100].

Besides noise, NUI sensors often provide a continuous stream of information, which is in big contrast to discrete input found in traditional applications. In traditional applications, a single key or mouse press typically results in a single directly connected command. Keyboard shortcuts such as copy  +  and paste  +  are amongst the most complex operations humans can perform using a keyboard, but are still relatively easy to interpret in computer code. Likewise, movement of the mouse is characterised by its sensitivity and enables pointing with good accuracy and very little noise, causing few apparent issues in today's input handling code [111]. In extreme contrast, novel input modalities that are based on accelerometers, touch-sensitive capacitive sensors, depth sensors or microphones generate an abundance of information. Such sensors are active all the time and generate a continuous stream of data, with an inverted signal-to-noise ratio compared to traditional input sensors. Therefore, to extract meaningful information from this sensor technology, we need developer support to *combine* input sources, to *segment* continuous streams and to reduce *noise* [27, 97, 142].

The implementation of traditional applications relies on stateless event handlers to process input one by one. This process is already complex and

⁶Wii Remote: <http://www.nintendo.com/wiiu/accessories>

⁷Samsung Smart Remote: <https://www.samsung.com/us/video/tvs-accessories/SEK-1000/ZA>

⁸SoftKinetic DepthSense: <http://www.softkinetic.com/Products/DepthSenseCameras>

⁹Microsoft Kinect: <https://www.microsoft.com/en-us/kinectforwindows>

¹⁰Samsung Smart TV: https://www.samsung.com/global/microsite/tv/2013_vi

¹¹Microsoft Xbox: <https://www.xbox.com>

error prone [111] and will be further complicated with the introduction of additional input device types. Existing applications for instance rely on a mouse cursor that can manipulate a single object at once. Multi-touch technology already allows the use of multiple fingers to manipulate multiple elements of a graphical user interface in parallel.

These observations lead us to the conclusion that although the industry has made important steps in terms of hardware technology, the development of sophisticated human-computer interaction is currently hampered by the lack of adequate programming abstractions. We argue that developers use inadequate programming abstractions to process this multitude of sensor input data.

It is essential to support developers with adequate programming abstractions to solve these problems. In the following sections we discuss existing approaches and summarise the need for additional means to properly address the challenges of today and tomorrow's human-computer interaction.

1.1 Research Context

This dissertation is positioned at the crossroad of two major computer science domains, namely Human-Computer Interaction and Programming Language Design. This is due to the fact that the complexity to extract information coming from input devices is increasingly higher. Today's software-related limitations severely hinder the ability to experiment with novel interaction techniques. Additionally, the robustness of every day's use of these devices compared to traditional input methods decreases due to the inability to program user interface code at an adequate abstraction level.

To extract meaningful information, such as gestures¹², from multiple sensors, the literature uses the term *multimodal fusion*. Sharma et al. [144] distinguish three levels of multimodal fusion, namely data-level fusion, feature-level fusion and decision-level fusion. In data-level fusion tasks, developers focus on the fusion of identical or tightly linked types of multimodal data. The goal is to (1) remove the excess of noise and to (2) provide feature candidates in (3) a real-time manner [41]. Feature-level fusion tasks rely on derived information from the data-level to fuse closely coupled modalities. A classical feature-level example is the integration

¹²A gesture in the context of this dissertation is based on the definition of Rhyne et al, namely a configuration of strokes, including handwritten text, pointing, and others [133].

of speech (captured by a microphone) and lip movement (captured by a camera) to improve speech recognition results [128]. Finally, decision-level fusion is the highest abstraction level of multimodal fusion, as it focuses on correlating information coming from loosely coupled modalities, such as speech and gestures. For example, the well-known *put that there* example by Bolt [11] fuses speech input such as “that” and “there” with pointing information to identify an object and a new location.

Unfortunately, mainstream programming languages do not align very well with the continuous event-driven nature of modern sensors, such as multi-touch surfaces and accelerometers. Imperative languages are designed with the assumption that the control flow is decided by the programmer and the state of the computer, and not driven by external events. The complexity to collect events, detect relevant patterns and to embed these into the application therefore engenders enormous challenges. The situation is further aggravated when more than one sensor is involved.

Various multimodal solutions have been proposed in literature. However, these research approaches are narrowly focused on a single fusion level, resulting in incompatibilities with one another. On the one hand, data-level fusion solutions are characterised by a focus on performance, noise filtering and typically provide abstraction in the form of composition boxes that need to be chained together [5, 97, 142]. This means data-level solutions lack high-level language abstractions and need to implement the logic inside composition boxes [41, 102]. On the other hand, decision-level fusion solutions focus on bridging the gap between input data and the application layer by providing abstractions in the form of dialogue management and high-level programming languages. Unfortunately, these high-level abstractions cannot cope with the vast amount of input data.

Due to the inability to properly describe multimodal interaction patterns, many researchers resort to machine learning solutions. Therefore, in general, many advantages of programming the interaction are lost, including the ability to control the result of the model (i.e. which conditions are crucial), to verify the model (i.e. is there a risk of accidental activations), to comprehend the model (i.e. will this work for other users) and to manually manipulate its preciseness (although there exist machine learning algorithms that partially allow this). Similarly, we argue that there needs to be a way to describe multimodal fusion concerns without having to resort to mainstream imperative programming languages. The use of an imperative programming style to implement event-driven fusion results in the inversion of control (i.e. the user, and not the code, dictates

the program flow) [73], requires a complex manual state management and lacks modularisation and composition abstractions.

The lack of adequate high-level language abstractions to properly describe multimodal fusion processes has been highlighted in recent surveys [41, 100]. Lalanne et al. explicitly state that *engineering aspects of fusion engines must be further studied*. As stated by Cuenca et al. [31] the ultimate goal of multimodal frameworks is to minimise the programming effort and allow for a faster creation of prototypes. To the best of our knowledge, this concern has not been adequately addressed. These surveys further argue that there is a need to efficiently manage the large amount of input data, to provide abstraction and composition of gestural interaction implementations, and to allow for a proper symbiosis between the application logic and the inherent continuous processing of multimodal events.

1.2 Research Goals

The extraction of meaningful information from multiple continuous input streams is a challenging task. In this work, we focus on two basic principles of software engineering, namely the separation of concerns and the reduction of accidental complexity by providing adequate abstractions.

Implementing multimodal interaction patterns is complex because it requires the concurrent processing of multiple event streams, the segmentation of endless input information and dealing with an abundance of noise. Even the recognition of a simple multimodal interaction demands for a major amount of work when using traditional programming languages. Furthermore, the reasoning over interaction from multiple users and devices significantly increases the complexity. Therefore, we need a clear separation of concerns between the multimodal application developer and the designer of new multimodal interactions to be used within these applications. The user experience designer must be supported by a set of programming abstractions that go beyond simple low-level input device event handling. Application developers on the other hand should be able to properly integrate these designed building blocks into their applications. Therefore, the direct beneficiaries of our work are developers who wish to rapidly prototype novel multimodal interaction patterns. Additionally, we consider that our approach should satisfy requirements to deploy these prototypes in the real world by gradually refining the multimodal interaction patterns. Finally, we wish to improve

the programming code of existing multimodal applications by exploiting the novel programming abstractions our approach offers. Indirectly, we target an improved human machine interaction by leveraging these new input sensors to drive numerous applications.

In software engineering, a problem can be divided into its *essential* and *accidental complexity* [18]. Accidental complexity relates to the difficulties a programmer faces due to the choice of software engineering and problem modelling tools. Other research fields, for example mathematics and natural sciences, have made great strides by designing simplified models of complex phenomena. These models are then verified with the help of experiments. This paradigm works because the complexities ignored in the model were not the essential characteristics of the phenomena. The modelling does not work if the complexities are of essence.

Essential complexity is caused by the inherent characteristics of the problem to be solved and cannot be reduced. Selecting or developing better tools can reduce the accidental complexity because the view and implementation of the model can be simplified. While the accidental complexity of today's traditional keyboard and mouse applications is partially addressed by the use of high-level programming languages such as Java, C# or XAML, we have not witnessed the same software engineering support for the development of multimodal applications. Furthermore, the few existing high-level multimodal languages cannot cope with the vast amount low-level input events, making existing multimodal languages not suitable for processing low-level input events.

To summarise, the practical problem this dissertation wants to address is the following:

How much can the accidental complexity of engineering multimodal interactions be reduced by using a non-imperative approach?

In this dissertation, we aim to provide a high-level programming language to describe multimodal fusion with the ability to process a vast amount of incoming information in real time. Concretely, the *research goals* for this approach are:

- To reduce the accidental complexity, the approach must facilitate the implementation of multimodal interaction patterns through **high-level programming abstractions**.

- To cope with the vast amount of low-level input events in soft real-time, the approach must facilitate an **execution engine** and react accordingly to the given multimodal descriptions. This includes dealing with *segmentation* (i.e. the process of extracting meaningful bits from continuous streams), and supporting *overlapping matches* (i.e. where input data can be shared between multiple multimodal descriptions).
- To fuse low-level data with high-level data, the approach must facilitate **cross-level multimodal fusion**. This is challenging due to the fact that low-level data and high-level data operate at different frequency rates. Additionally, the approach must integrate with existing fusion processes, such as feature extractors, in order to reuse existing specialised methods with a small amount of development effort.
- The **applicability** of this approach must be demonstrated in real world settings. This verifies the ability to describe the functionality defined by customers, as well as the real-time processing properties of the engine and its robustness against noise in real world settings.

1.3 Methodology

To address the aforementioned challenges, we rely on the design science research methodology defined by Pepper et al. [127]. This methodology is based on six steps: problem identification and motivation (Chapter 2), definition of the objectives for a solution (Chapter 2 and 3), design and development (Chapter 4 and 5), demonstration (Chapter 6), evaluation and communication (Chapter 6 and 7). The challenges to express multimodal interaction patterns transcend the traditional focus of the Human-Computer Interaction domain and involve advanced computational resources. Therefore, in order to pursue our research goals, the approach consists out of two major design and development artefacts: a programming language and a compatible runtime platform with a unified architecture.

1.3.1 Language-oriented Approach

In order to ease the application development process, we need tools that let developers focus on the essential complexity of the multimodal fusion

problem. Existing frameworks mostly rely on imperative programming paradigms that do not align well with the event-driven nature of HCI. This complexity translates into poor abstraction levels, inadequate recognition rates, and ad-hoc solutions. Therefore, we propose a language-oriented approach that allows developers to express their multimodal tasks in a **declarative** manner. A declarative programming style allows the programmer to think about *what* the fundamental conditions are, instead of analysing *how* to process input events one by one as necessary in an imperative approach. Our research shows that our declarative language approach offers a number of important benefits, including the reuse of existing code through modularisation and composition. This reuse of existing code is not limited to linking components as is commonly done in pure data stream approaches, but actually allows case-specific customisation without the need to modify existing code. Our declarative approach corresponds to an implicit programming flow [139], where the execution engine translates the descriptions into a Rete network [54]. This execution engine and its overarching architecture form the second part of our work.

1.3.2 Architecture and Execution Engine

An important aspect of our work is to apply the proposed solution in real world scenarios. This requires (1) an **efficient processing** engine and (2) an **extensible architecture** to incorporate existing work (e.g. practical solutions that have been derived with machine learning). Additionally, many other multimodal concerns, including cross-level fusion, overlapping matches or event expiration, form part of our focus. We describe a **unified fusion** architecture that interprets our declarative high-level language and enables the processing of real-time sensor data. Such a unified architecture should support multi-level fusion across low- and high-level data and be extensible to enable the incorporation of existing feature processes. This integration is based on existing techniques, including the publish/subscribe [49] model.

The core idea of our unified architecture is the use of a central fact base. The fact base, in combination with a declarative language and the Rete algorithm, allows developers to easily share information between various fusion processes. A main characteristic of the Rete network is the ability to efficiently cache intermediate results. This means that if an input event satisfies one condition of a multimodal description consisting out of two conditions (i.e. event *a* and event *b* need to happen), an

intermediate representation of the result is temporarily kept in memory. This information is maintained for some time until the event expires to free memory for newer input events.

The central fact base also allows for extensibility and sharing of information with other processes. Each process can access all available information, including intermediate results, application information and share its derived knowledge with all other processes.

Finally, we provide novel mechanisms to incorporate application information. For instance, the x, y position of particular GUI components or the current interaction state provide precious information for the multi-touch gesture recognition processes. Context, such as application information, is of utmost importance to properly process multimodal input data.

1.3.3 Towards a Solution

Listing 1.1 provides a preview of our language abstractions. It describes that a multi-touch pinch gesture (**Pinch**, line 2) should be interpreted as a basic zoom operation (**scale**, line 5) when performed on top of a digital image (**inside**, line 4, a **Image**, line 3). The runtime interpretation of this declarative description is illustrated in Figure 1.1. Our approach transforms declarative conditions into a directed acyclic graph using the Rete algorithm [54]. Input events are progressively filtered and joined with other events in order to derive a conclusion. This process is reactive because the input drives the computation.

Listing 1.1: Shrink an image

```

1 rule shrinkImage
2   p = Pinch
3   i = Image
4   p ← inside i
5   call i.scale(p.difference)
6 end

```

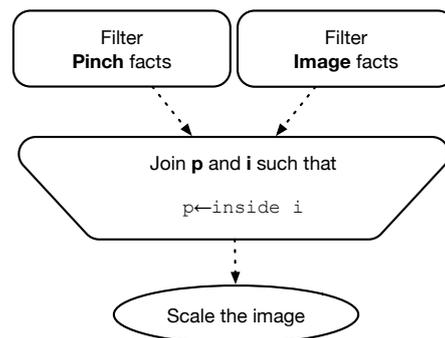


Figure 1.1: Rete graph

1.4 Contributions

In the following section we would like to highlight the main contributions of this dissertation.

Analysis of Criteria, Challenges and Open Issues in Multimodal Fusion Frameworks

Based on a literature study of the broad domain of multimodal fusion and our expertise, we propose a set of 30 criteria. These criteria unify a number of concepts from existing work and expose a large number of issues that received little attention. A first analysis was discussed at the first international workshop on Engineering Gestures for Multimodal Interfaces (EMGI 2014) [46, 69]. In this dissertation, we provide a second iteration of these criteria which is used as a guideline throughout the text.

A Multimodal Programming Language

We define a new multimodal programming language, called Midas. Midas aims to reduce the accidental complexity of developing multimodal interaction and therefore enables developers to focus on the essential complexity. Midas is a declarative programming language providing a number of multimodal-specific constructs to ease the modularisation and composition of fusion processes. Furthermore, it supports customisation, negation and an application symbiosis which remained rather primitive in existing approaches. Our programming language therefore significantly increases the expressiveness in contrast to existing solutions.

A Multimodal Fusion Engine

Mudra is a unified and extensible multimodal architecture focusing on the real-time processing of input events. Mudra reconciles data stream and semantic inferencing approaches by relying on a central information storage, in the form of a fact base and an efficient Rete network algorithm to process the raw data. This approach offers inherent support for fusion across low-level data and high-level semantic information. Our architectural design further enables the incorporation of existing solutions through the use of a publish/subscribe and actor model.

Declarative Description of 2D and 3D Gestures

We introduce a novel method to declaratively describe complex 2D and

3D gestures based on *control points*. With this method, gesture trajectories are split into multiple points that need to be traversed. Control points enable the automated segmentation of continuous input data and unyieldingly deal with noise by ignoring irrelevant events. Automated gesture segmentation is a valuable asset for many cases where begin and end points of gesture input cannot be clearly defined. Segmentation problems are fundamental when processing continuous input streams and are prevalent in novel, always-on sensors.

Real World Deployment of the Presented Abstractions

We performed a real world deployment of our proposed solution in multiple scenarios and discuss the results. Firstly, we performed a live gesture programming session as a demo during the TEI 2011 conference. At the same conference, we controlled the interaction of our presentation via a Kinect sensor, only a few weeks after it was released. Secondly, we deployed our multi-touch abstractions in a NoiseTube demonstrator showcased at the “Brussels Innovates!” exhibit. Thirdly, we provided expressive control of indirect augmented reality during live music performances. Finally, we designed and tested an augmented fighting game using water as tactile feedback. A more in-depth description of our demonstrators can be found in Section 1.5.2.

1.5 Supporting Publications and Demonstrators

1.5.1 Publications

We disseminated our work in one journal, 7 conferences and 5 workshops.

Software Engineering Abstractions for the Multi-Touch Revolution *Proceedings of ICSE 2010, 32nd ACM/IEEE International Conference on Software Engineering, Microsoft Student Research Competition*. Lode Hoste [63]. This short paper describes the first approach to a declarative language to express multi-touch gestures. It uses declarative rules to modularise and compose gesture implementations. The goal is to disentangle code found in existing imperative approaches that rely on event callbacks.

Midas: A Declarative Multi-Touch Interaction Framework *Proceedings of TEI 2011, 5th International Conference on Tangible, Embedded and Embodied Interaction*. Christophe Scholliers, Lode Hoste, Beat Signer and Wolfgang De Meuter [141]. The work in this paper extends the previous paper and proposes additional spatial and temporal operators to describe multi-touch gestures. Furthermore, shadow facts are introduced that allow gestures to be linked to GUI components. We need to clarify that the language presented in this paper is named Midas but refers to an older incarnation. In this dissertation we present a second iteration of the Midas programming language.

Mudra: A Unified Multimodal Interaction Framework *Proceedings of ICMI 2011, 13th International Conference on Multimodal Interaction*. Lode Hoste, Bruno Dumas and Beat Signer [64]. In this paper we describe our unified architecture to perform multimodal fusion across low-level data and high-level semantic information.

SpeeG: A Multimodal Speech- and Gesture-based Text Input Solution *Proceedings of AVI 2012, 11th International Working Conference on Advanced Visual Interfaces*. Lode Hoste, Bruno Dumas and Beat Signer [65]. SpeeG is a multimodal speech- and body gesture-based text input system targeting media centres, set-top boxes and game consoles. It provides a controller-free zoomable user interface that combines speech input with a gesture-based real-time correction of the recognised voice input.

Parallel Gesture Recognition with Soft Real-Time Guarantees *Proceedings of the compilation SPLASH 2012 workshops, AGERE! workshop, 2nd International Workshop on Programming based on Actors, Agents, and Decentralized Control*. Thierry Renaux, Lode Hoste, Stefan Marr and Wolfgang De Meuter [131]. In collaboration with co-authors, we designed a parallel and scalable variant of Mudra, called PARTE. PARTE is a complex event-processing engine and is compatible with the Midas Language. It detects event patterns and provides soft real-time guarantees for the computational processes.

Declarative Gesture Spotting Using Inferred and Refined Control Points *Proceedings of ICPRAM 2013, 2nd International Conference on Pattern Recognition Applications and Methods*. Lode Hoste,

Brecht De Rooms and Beat Signer [66]. In this work we propose a novel gesture spotting approach for processing continuous streams of two- or three-dimensional Cartesian coordinates. This approach translates into declarative Midas code and offers fine-grained control over the gesture trajectory.

Expressive Control of Indirect Augmented Reality During Live Music Performances *Proceedings of NIME 2013, 13th International Conference on New Interfaces for Musical Expression*. Lode Hoste and Beat Signer [67]. In this paper we present a real world application of Midas and Mudra that uses explicit gestures and implicit dance moves to control the visual augmentation of a live music performance. The focus of this work is to evaluate our abstractions in a challenging environment. Firstly, only a single sample is available for each of the five 3D gestures. Secondly, there was no ‘noise’ data available that contains other movement of the artists during the song. Thirdly, the 3D input data needs to be processed in real-time and finally there was little room for recognition errors.

Cloud PARTE: Elastic Complex Event Processing based on Mobile Actors *Proceedings of AGERE! 2013, 3rd International Workshop on Programming based on Actors, Agents, and Decentralized Control*. Janwillem Swalens, Thierry Renaux, Lode Hoste, Stefan Marr and Wolfgang De Meuter [151]. In collaboration with co-authors, we extended PARTE to dynamically distribute the processing load on multiple machines. It involves data and code mobility of rete networks [54] and automated load-balancing mechanisms.

SpeeG2: A Speech- and Gesture-based Interface for Efficient Controller-free Text Input *Proceedings of ICMI 2013, 15th International Conference on Multimodal Interaction*. Lode Hoste and Beat Signer [68]. In this paper we present a second version of SpeeG, a multimodal text entry solution combining speech recognition with gesture-based error correction. Four innovative prototypes for the efficient controller-free text entry have been developed and evaluated. A quantitative evaluation of our SpeeG2 text entry solution revealed that the best of our four prototypes achieves an average input rate of 21.04 WPM (without errors), outperforming current state-of-the-art solutions for controller-free text input.

Water Ball Z: An Augmented Fighting Game Using Water as Tactile Feedback *Proceedings of TEI 2014, 8th International Conference on Tangible, Embedded and Embodied Interaction*. Lode Hoste and Beat Signer [70]. In this paper we present a second real world application of Midas and Mudra in the form of a game. Water Ball Z is a novel interactive two-player game that allows kids and young adults to “fight” in a virtual world with water-based physical feedback. The focus lies on the online processing capabilities of Mudra and the integration of Midas with the application layer to deliver incremental feedback.

Parallel Gesture Recognition with Soft Real-Time Guarantees *Science of Computer Programming*. Stefan Marr, Thierry Renaux, Lode Hoste and Wolfgang De Meuter [113]. In collaboration with co-authors, we evaluated the scalability of PARTE on machines with up to 64 cores. The presented evaluation indicates that gesture recognition can benefit from the exposed parallelism with superlinear speedups. The paper demonstrates the scalability of a declarative approach for gesture recognition and multimodal fusion processes.

Criteria, Challenges and Opportunities for Gesture Programming Languages *Proceedings of EGMI 2014, 1st International Workshop on Engineering Gestures for Multimodal Interfaces*. Lode Hoste and Beat Signer [69]. The work described in this paper summarises a large number of criteria, challenges and opportunities for gesture programming languages. In this dissertation, we have elaborated on these concerns and extended them for multimodal fusion solutions.

Software Engineering Principles in the Midas Gesture Specification Language *Proceedings of PProMoTo 2014, 2nd International Workshop on Programming for Mobile and Touch*. Thierry Renaux, Lode Hoste, Christophe Scholliers and Wolfgang De Meuter [132]. In this paper we present our second and latest iteration of the Midas programming language. In this dissertation, we have elaborated on these concerns and extended them for multimodal fusion solutions.

Next to these publications, I co-organised the first international workshop on Engineering Gestures for Multimodal Interaction (EGMI 2014) [46]. Additionally, I have been invited to review papers for several journals and conferences, including Science of Computer Programming, International Conference on Tangible, Embedded and Embodied Interac-

tion, European Conference on Object-Oriented Programming, International Conference on Distributed Applications and Interoperable Systems, Journal on Software: Practice and Experience, Computers in Biology and Medicine.

1.5.2 Demonstrators

Our software engineering abstractions for expressing multimodal interaction have been used in real world setups. The following demonstrators form part of our validation of the applicability and robustness of our approach.

Live Gesture Programming Session. In this session, participants of the TEI 2011 conference were able to spontaneously propose novel multi-touch gestures. We then implemented these gestures within a few minutes using initial incarnations of Midas and Mudra. Challenging gestures during this session have been a driving force for vast improvements and numerous extension of the multimodal framework presented in this dissertation.

Multi-Touch-enabled NoiseTube. A multi-touch enabled demonstrator was showcased at the exhibit “Brussels Innovates!” which ran from October 17 to 29 (2011) in the Woluwe Shopping Centre in Brussels. We presented this exhibition along with innovative actors in Brussels such as BMW Motors, Solar Impulse, PlantDesign, and others.

Expressive Control of Indirect Augmented Reality During Live Music Performances. The Midas and Mudra abstractions have been used to program and recognise complex 3D gestures. During multiple live music performances at the ArtCube¹³ and the International Convention Center Ghent¹⁴, our expressive control of indirect augmented reality was received with great enthusiasm by the audience. The artist was able to control the visualisations via 3D gestures as part of a dance act. Four live performances took place in 2012 and 2013 with an audience of about 1800 people.

¹³ArtCube: <http://artcube.be>

¹⁴International Convention Center Ghent (ICC): <http://www.iccghent.com>

1.6 Dissertation Outline

The remainder of this dissertation is structured as follows:

Chapter 2. Multimodal Interaction. In this chapter we discuss and analyse the three multimodal fusion levels [144]: data-level fusion, feature-level fusion and decision-level fusion. As the multimodal domain is broad, we also define the terminology used throughout this dissertation. We further enumerate and discuss criteria for expressing multi-level multimodal fusion.

Chapter 3. Related Work We provide an overview of the landscape of existing multimodal frameworks. Existing work is categorised into two main strands, namely data stream and semantic inferencing solutions. We then focus on the expressiveness of multimodal and gesture programming language support.

Chapter 4. Midas: A Programming Language For Expressing Multimodal Interaction. This chapter defines the Midas programming language and discusses its features on the various multimodal fusion levels. We analyse our language using the language feature criteria expressed in Chapter 2.

Chapter 5. Mudra: A Unified Multimodal Interaction Framework We present the architecture of our proposed framework. It demonstrates how events drive the computation expressed in the Midas language with various examples. We discuss the capabilities of the framework over the multimodal fusion criteria and show which abstractions are available on different fusion levels. Finally we conclude with the implementation details of the proposed framework.

Chapter 6. Midas & Mudra at Work In this chapter we analyse the applicability of our approach with a number of use cases. These use cases serve as the validation of our work with respect to real-time execution properties.

Chapter 7. Conclusion. The last chapter revisits the research goals to argue that the combination of Midas and Mudra forms an adequate solution to solve a number of important challenges in the multimodal

fusion domain. This chapter further discusses a number of limitations and proposes an interesting body of future work. For example, the notion of transactional user interfaces, originating from limitations in our experiments is discussed.

2

Multimodal Interaction

In this chapter we discuss the features of multimodal interaction and multimodal fusion processes. We present three multimodal fusion levels and a list of 30 criteria for multimodal frameworks we observed during the literature study during the last four years. The goal of this chapter is twofold: (1) we contextualise our research in the vast amount of multimodal literature and (2) the definition of distinguishing criteria in order to identify missing abstractions and compare related work.

2.1 Multimodal Concerns

In their survey, Lalanne et al. [100] express four important concerns a fusion engine needs to deal with: (1) probabilistic input, (2) multiple and temporal combinations, (3) adaptation to context, tasks and users, and (4) error handling. In the following, we elaborate on these four concerns.

Probabilistic input Traditional applications rely on deterministic events such as keystrokes or mouse clicks. However, in most multimodal solutions developers rely on sensor information that should be interpreted first. This interpretation needs to deal with a high degree of uncertainty due to sensor noise, such as background noise or convoluted image frames. Multimodal frameworks aim to reduce this uncertainty by ignoring noise, allowing relaxation parameters (such as spatial approximation via inter-

vals) or by embedding probabilistic information within the events. Dealing with probabilistic input is therefore a key concern for multimodal fusion engines.

Multiple and temporal combinations The CASE model is used to classify multiple and temporal combinations from a machine point of view. The CASE model [123] categorises different usage of modalities in concurrent, alternate, synergistic and exclusive. The concurrent category is defined as the parallel use of modalities without the need for time synchronisation between the two modalities. If this information is combined, such as the fusion of audio input with lip information to increase recognition rates [17, 128], the fusion process is synergistic. In the alternate category, modalities are used in a sequential manner and information will be fused when the interaction is completed. An example is shown by Kuijpers et al. [98]: the user can enter text in natural language with word references (for instance the user types “what is the size of this disk?”) and then clicks with the mouse on the graphical object representing a physical disk. Exclusivity can be enforced to reduce accidental activations. For example users cannot turn the page via a swipe gesture while actively writing notes with the pen. Without exclusiveness, a moving palm on the touch interface might accidentally trigger the swipe gesture.

From a human perspective, the CARE properties [29] characterise and assess the usability and fusion in multimodal interaction. Complementary modalities are used when an interaction is best achieved with two or more combined modalities. This allows for a more natural interaction such as using speech and pointing at the same time. The assignment property indicates the absence of a choice. The user is forced to use a single modality to reach their goal. On the other hand, when two modalities with the same expressiveness are required to be used to achieve one goal, we talk about redundancy. This can be used to limit unintentional actions, for example to deleting a movie on a smart TV requires a voice command and the confirmation of the head nodding. Finally, equivalence defines that users can choose between modalities to express their intention.

Fusion engines should support the CASE and CARE properties to deal with various multimodal scenarios, however this requires the ability to express advanced temporal relations between multiple input sources.

Adaptation to context, tasks and users Adaptation to context can be an important factor to increase recognition rates. The interpretation of

a gesture might be different, different clues might be used (such as the gaze direction) or a lower threshold can be used to activate commands when the user is increasingly nervous. In order to deal with these adaptations, fusion engines require access to application-level information and need to react to inferred cues from other modality input.

Error handling Multimodal frameworks aid developers to handle input containing noise and missing information. However, obtaining error-free results will be very hard. This is also the case for human-to-human interaction. Therefore multimodal solutions should provide mechanisms to correct mistakes and learn from them.

2.2 Multimodal Fusion Levels

The previously mentioned multimodal concerns span the entire fusion process. However, the transformation of low-level input data to high-level semantic information is complex and typically happens in multiple stages. Sharma et al. [144] distinguish three levels of abstraction to characterise multimodal input data fusion: data-level fusion, feature-level fusion and decision-level fusion. In this section, we present these different levels with some classic use cases.

2.2.1 Data-Level Fusion

Data-level fusion focuses on the fusion of identical or tightly linked types of multimodal data. The goal is to (1) remove the excess of noise to (2) provide feature candidates in (3) a real-time manner [41]. This is challenging due to the fact that information continuously arrives at a high frequency. The classic illustration of data-level fusion is the fusion of two video streams coming from two cameras filming the same scene at different angles in order to extract the depth map of the scene. Data-level fusion rarely deals with the semantics of the data but tries to enrich or correlate data that is potentially going to be processed by higher-level fusion processes. As data-level fusion works on the raw data, it has access to the detailed information but is also highly sensitive to noise or failures. Data-level fusion frequently entails some initial processing of raw data including noise filtering or very basic recognition.

2.2.2 Feature-Level Fusion

Feature-level fusion is one step higher in abstraction than data-level fusion. Feature-level fusion of modalities is typically applied to closely coupled modalities with possibly different representations. A classic example is speech and lip movement integration [128], where data comes from a microphone that is recording speech as well as from a camera filming the lip movements. The two data streams are synchronised and in this case the goal of the feature fusion is to improve speech recognition by combining information from the two different modalities. Feature-level fusion is less sensitive to noise or failures than data-level fusion and conveys a moderate level of information granularity. Typical feature-level fusion algorithms include statistical analysis tools such as Hidden Markov Models (HMM), Neural Networks (NN) or Dynamic Time Warping (DTW), which translate a sequence of events into feature-level information based on previously annotated data (i.e. training data). For example, DTW returns the classification label (i.e. name) of a sample that corresponds most closely to a given sequence of events.

2.2.3 Decision-Level Fusion

Decision-level fusion focuses on deriving interpretations based on semantic information. It is the most versatile kind of multimodal fusion, as it can correlate information coming from loosely coupled modalities, such as speech and gestures. The well-known *put that there* example by Bolt [11] fuses speech input such as “that” and “there” with pointing information to identify an object and a new location. Decision-level multimodal fusion includes merging high-level information obtained by data- and feature-level fusion as well as modelling human-computer dialogues. Decision-level fusion is assumed to be highly resistant to noise and failures by relying on the quality of previous processing steps. Therefore, the information that is available for decision-level fusion algorithms may be incomplete or distorted. Typical classes of decision-level fusion algorithms are meaning frames [158], unification-based algorithms [78], finite-state machines [76] or symbolic-statistical algorithms [22].

Surprisingly, existing multimodal interaction frameworks often excel at one specific fusion level but encounter major difficulties at other levels. We argue that the reason for these limitations lies on the architectural level and in particular how the initial data from different modalities is handled. In particular, each fusion level comes with its own requirements. Data-

level fusion focuses on performance in order to keep pace with the vast amount of input information. Feature-level fusion requires robust feature extraction methods that filter out the final noise. Finally, decision level requires contextual information to properly manipulate application-level information.

We argue that a unified fusion framework is required to enable fusion of information at all levels, while retaining high-level software engineering abstractions. In the next section we provide an extensive list of criteria which multimodal fusion engines need to cope with.

2.3 Criteria for Expressing Multi-Level Multimodal Fusion

In this dissertation we focus on a unified fusion framework with both high-level programming language and architectural support. Multimodal programming languages are designed to support developers in specifying their multimodal gesture interaction requirements more easily than with general purpose programming languages [40, 47, 97]. General purpose programming languages such as Java often require an excessive amount of code to express a developer’s intention which makes them hard to read and maintain. A domain-specific language (DSL) might help to reduce the repetitive boilerplate that needs to be written in existing languages as described by Van Cutsem [154]. Van Cutsem argues that languages can *shape the thought* (earlier attributed to “The limits of my language means the limits of my world”, Ludwig Wittgenstein). For instance, interaction patterns can be declaratively described by its requirements versus an imperative implementation with manual state management. This impacts the way of thinking during design and implementation. A programming language can also be seen as a *simplifier* that omits complex features, such as memory pointers, which are not helpful to describe gestural interaction. Finally, domain-specific languages can be used as a *law enforcer*. For example, the Proton [95] gesture language analyses gesture descriptions and informs the developer about unintentional gesture overlaps at compile time. Law enforcement further enables the inference of properties that help domain-specific algorithms to obtain better classification results or reduced execution time. Next to these concerns related to the programming language, the execution model and architecture also require deep thought as they need to coordinate the vast amount of input events and keep up the computational effort to perform fusion.

Criteria We define a number of criteria that characterise (1) the choice of a particular framework, (2) the implementation of the multimodal interaction and (3) the open issues in the multimodal engineering domain. These criteria combine features proposed by different approaches, including domains such as machine learning, multimodal architectures, multimodal languages and template matching. They are compatible but defined on a more fine-grained level, than with previously established fusion criteria [41, 116]. Together with our experiments and the reuse of core criteria of existing work, these requirements offer a refined view of multimodal fusion frameworks. We split the criteria up in four main categories: language features, multimodal processing, multimodal specification and accessibility as well as tooling.

2.3.1 Language Features

The following criteria have an effect on the software engineering aspects of the multimodal interaction specification. We argue that they might require corresponding features to be implemented in the processing engine.

Modularisation

By modularising multimodal definitions we can reduce the effort to gradually increase the number of interaction scenarios. Modularisation is based on the separation of concerns principle, one of the main principles in software engineering, which dictates that different modules of code should have as little overlapping functionality as possible. Therefore, in modular approaches, each multimodal specification is written with its own separate linguistic component, such as a separate function, rule or definition.

Composition

Composition allows programmers to tame complexity by building complex interactions from simpler building blocks. For instance, a tap gesture is defined by a touch down event followed shortly by a touch up event and with limited spatial movement in between. A double tap gesture can then be defined by the composition of two tap gestures with a defined maximum time and space interval between them. Composition thus allows developers to abstract and reuse multiple modular specifications to define more complex interaction.

Customisation

Customisation is concerned with the effort a developer faces to modify a multimodal interaction gesture specification for use in a different context. How easy is it, for example, to adapt the existing definition of an interaction when an extra condition is required or the order of events should be changed? For graphical programming toolkits, the customisation aspect is broadened to how easy it is to modify the automatically generated code (if possible at all). Note that in many machine learning approaches customisation is limited due to the lack of a decent external representation [82].

Negation

Negation is a feature that allows developers to express a context that should *not* be true in the definition of a particular interaction. Many approaches partially support this feature by requiring a strict sequence of events, thereby implying that no other events should happen in between. However, it is still crucial to be able to describe explicit negation for some scenarios. Suppose one wants to express that there should be no other user in the proximity in order to preclude unintended actions or that the dominant hand should not be located near the waist before the beginning of a gesture.

Application Symbiosis

The integration of application information in the fusion process allows developers to inspect the application state before execute certain functionality. For instance, a multi-touch scroll gesture can only happen when both fingers are inside the GUI region that supports scrolling [47]. This means application-specific information such as the location of graphical components is required to precisely constrain the multi-touch scroll gesture. This further aids the disambiguation process and thus increases the recognition quality. Another example is when a GUI widget needs to be rescaled or rotated. The gesture can be defined by requiring that one finger should be on top of the GUI component while the other two fingers are executing a pinch or rotate gesture in the neighbourhood.

Activation Policy

Whenever a multimodal interaction is recognised, an action is typically executed. In some cases the developer may want to provide a more detailed activation policy such as *trigger only once* or *trigger when entering and leaving a particular state*. Another example is the sticky bit [47] option that activates the gesture for a particular GUI object. A shoot-and-continue policy [70] denotes the execution of a complete gesture followed by online gesture activation. The latter can for example be used to define a lasso gesture where at least 360° circular movement is required and afterwards each incremental part (such as every 90°) causes a gesture activation.

Unbound Variables and Unification

An unbound variable allows developers to define a variable which is instantiated by a value from an input event at runtime. For example, an unbound variable `1` can be used to capture the x value of an event if it is included in the spatial interval (i.e. the x coordinate should be between 10 and 50, and an example runtime value of the coordinate bound to `1` is 12). An unbound variable can also be used to describe dependencies to other values (i.e. bind `1` to the average of the x coordinates of two events). This is particularly useful to correlate different events if the specification of concrete values is not feasible and when the runtime value is meaningful for the rest of the program. Unification, in the context of this dissertation is used to enforce equality between two unbound variables.

2.3.2 Multimodal Processing Concerns

Multimodal processing concerns impact the design of a multimodal fusion engine. For example, does the engine need to be reactive such that it executes code for every input event or can it wait for certain characteristic events (i.e. as done in offline processing tasks)? Can the engine adapt to runtime changes, such as (dis)connecting devices or additional code provided by the user? A combination of the concerns targeted by the engine therefore decides its main architecture and implementation.

Online Processing

Interactions such as a multi-touch pinch gesture for zooming require feedback *while* the gesture is being performed. These so-called online in-

teractions can be supported in a framework by allowing concise definitions (i.e. interactions defined by hard-coded raw values) or by providing constructs (i.e. high-level features) that offer advanced callback mechanisms to monitor the progress of a larger interaction.

Offline Processing

Offline definitions are executed when an interaction is completely finished and typically represent a single command. This type of interaction is easier to support in multimodal programming languages, as they need to pass the result to the application only once after it has been detected. This is in contrast to online processing which typically require an incremental evaluation for performance reasons. Offline definitions increase the robustness due to the ability to validate the entire setting. A complete collection of events increases the correctness of the detection when compared to online processing where decisions are made when large parts are yet unknown.

Partially Overlapping Matches

Several conditions of a multimodal interaction definition can be partially or fully contained in another definition. The manifestation of this concern is shown in two ways by the well-known *put that there* example by Bolt [11]. On the one hand, in the data-level fusion, the speech recogniser component recognising the *put* word should be capable of dealing with other word candidates that begin with the same pronunciation. This is the case when other words such as *push* and *pull* are alternative commands. These overlapping initial conditions increase the complexity in the processing engine, as it has to keep track of multiple candidate solutions from the same input events. Furthermore, the initial sounds of a word might not be distinctive enough and will need to be reconsidered when the word has been fully pronounced or the context became clear. On the other hand, at a higher-level fusion layer, keeping track of partially overlapping matches is required to deal with user stuttering or background noise (i.e. multiple events satisfy the same constraint). If the speech recogniser provides the word *that* multiple times, the most interesting one related to the pointing gesture should be chosen. However one cannot know this as soon as the first *that* candidate arrives and therefore it is crucial for the processing engine to be able to keep track of the partial matches until they are either rejected or accepted. Keeping track of multiple partial

matches is a complex mechanism that is supported by several approaches and intentionally blocked or ignored by others.

Segmentation

Typically, streams of sensor input events do not contain explicit hints about the beginning and the end of a gesture. This is known as *segmentation* or *gesture spotting*. Segmentation gains importance given the trend towards the continuous capturing and free-air interaction in which a single event stream can contain many potential begin events. The difficulty of gesture segmentation manifests itself when one cannot know beforehand which potential “begin” events should be used until a “middle” or even an “end” candidate event is found to form decisive gesture trajectory. It is possible that potential begin and end events can still be replaced by better future events. For instance, how does one decide when a free-air *swipe right* gesture begins or ends without reasoning about past (i.e. starting poses) or the future (i.e. trigger a narrow swipe right or wait for a wider swipe right gesture)? This lack of explicit begin- and end-points generates a lot of gesture candidates and increases the computational complexity. Several approaches tackle this issue by means of a velocity heuristic with a slack variable (i.e. a global constant defined by the developer) or by programming incremental processing code. Many solutions make use of a garbage gesture model to increase the accuracy of the gesture segmentation process. Other segmentation problems include the distinction of background noise versus conversations to invoke the speech recogniser at the correct time.

Figure 2.1 illustrates a continuous event stream from a multi-touch device with two segmentation candidates. The first candidate begins after a couple of events and is identified by a heuristic detection for characteristic turning points. The extraction of this candidate from the continuous stream is crucial for higher level processing components to classify the gesture. In this case a gesture classifier later rejects the first candidate. Unfortunately, initial phase of the second candidate happens before the final phase of the previous candidate. In this example, sophisticated engine support is required that supports both segmentation and overlapping matches.

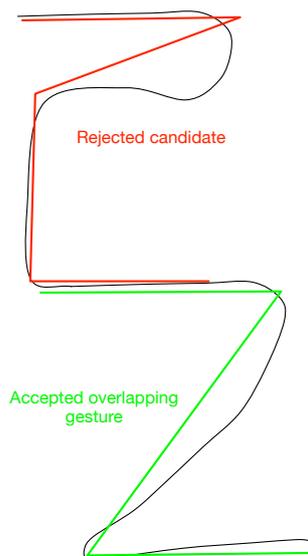


Figure 2.1: Segmenting and overlapping matches in a Z gesture

Synchronising Streams

The time at which events are received by the fusion engine frequently differs from the time they are produced. This happens due to physical properties, input frequency, sensor latency or an interpretation delay. For example, a stream containing camera-based lip tracking will be ahead in time compared to a stream with audio cues such as phonemes because speech recognisers have an inherently high latency. In order to fuse these two streams, the programmer needs to resynchronise these input streams such that events that were produced at the same time are processed together. The resulting synchronisation process is illustrated in Figure 2.2. In this example, events from streams e and f should be combined based on their timestamp (denoted as a number in subscript). However more complex configurations are possible when dealing with additional streams and sequences of events.

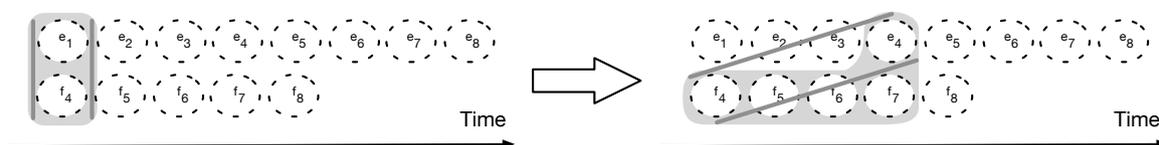


Figure 2.2: Synchronising streams

Event Expiration

The expiration of input events is required to keep the memory requirements and processing complexity within limits. The manual maintenance of events is a complex task and most frameworks offer at least a simple heuristic to automatically expire old events. In multi-touch frameworks, a frequently used approach is to keep track of events from the first touch down event to the last touch up of any finger. This history tracking method introduces problems when dealing with multiple users as there might always be at least one active finger touching the table and therefore no events will be removed. Another approach is to use a timeout parameter, thus creating a sliding window solution. An advantage of this approach is that the maximum memory usage is predefined, but a slack value is required.

Long-Term Reasoning

When events are managed by a sliding time window technique, the reasoning capabilities of rules become limited to events during that period. In order to retain valuable information for future extraction, a long-term reasoning alternative must be provided. Long-term reasoning can be used to store characteristic events (e.g. the user entered a room at time t), discover frequently occurring patterns (e.g. an employee makes tea every morning) or persist information for future profiling (e.g. a gesture from user a is frequently unintentionally activated).

Concurrent Interaction

In order to allow concurrent interaction, one has to keep track of multiple partial instances of a multimodal recognition process. For instance, multiple fingers, hands, limbs or users can perform the same gesture at the same time. To separate these instances, the framework can offer constructs or native support for concurrent multimodal processing. In some scenarios, it is hard to decide which touch events belong to which hand or user. For example, Proton splits the gesture execution surface in half to support two player games. Each player is therefore limited to execute gestures on their private zone, otherwise gestures will be invalidated by the touches from the other player. This STATIC split of the surface only works for a limited number of use cases. A better method is to set a maximum bounding box of the gesture [47] or to

define the spatial properties of each gesture condition. The use of GUI-specific contextual information can also serve as a separation mechanism. However, for some cases it is not possible to know beforehand which combination of fingers will form a gesture, leading to similar challenges as discussed in the *partially overlapping matches* criterion where multiple candidates need to be tracked at the same time.

Portability, Serialisation and Embeddability

Concerns such as portability, serialisation and embeddability reflect the platform independence of an approach. Portability is defined by how easy it is to run the framework on different platforms. Some approaches are tightly interwoven with the host language, which limits portability and the transfer of multimodal interaction definitions over the network. Such code mobility could be used to exchange multimodal definition sets between users or even to offload the recognition process to a dedicated server with more processing power. The exchange requires a form of serialisation of the interaction definitions that is usually already present in some domain-specific languages. Embeddability deals with technical challenges to embed a multimodal framework in existing applications. For instance, a framework can be delivered as a daemon process or a library. Embeddability also involves an assessment on how compatible the abstractions are with existing programming languages.

Runtime Definitions and Device Instantiation

Refining multimodal parameters or outsourcing definitions to multimodal services requires runtime modification support by the framework. The refinement can be instantiated by an automated algorithm (such as an optimisation heuristic) by the developer (during a debugging session) or by the user to provide their preferences. Multimodal scenarios that occur in dynamic environments where devices become available and disconnected at any time require a dynamic device instantiation mechanism.

Reliability and Scalability

The dynamic nature of user input streams implies the possibility of an abundance of information in a short period of time. In order to be reliable, a framework might offer a maximum computational boundary for a given setting, such as a maximum amount of events per second [113]. Without such a boundary, users might flood the system with input events,

thereby introducing latency. Reliability also involves security measures, such as the encapsulation of low-level functionality to reduce attack vectors. Scalability deals with the ability to increase the computational boundary of a framework to support additional sensors and users using multiple cores and distribution across machines. In recent years, access to many machines and whole data centres became a commodity and enables frameworks to perform more complex fusion by tapping into this vastly increased processing power. However, a scalable approach must still offer some reliability guarantees to deal with latency.

2.3.3 Multimodal Disambiguation

Developers need a way to disambiguate noise from multimodal interaction patterns. Often, expressing such conditions is difficult as a single event does not convey the necessary information. Therefore, disambiguation criteria focus on spatial and temporal relations between events from (potentially) different input sources. Furthermore, multimodal frameworks need to deal with decision making processes, which includes criteria such as prioritisation and verification. These criteria have an impact on the language and the architecture of an approach.

Spatial Specification

The role of spatial specification is to describe spatial relations between events. A key ingredient in the context of multimodal fusion is the ability to specify spatial approximation. Spatial approximation is required to support the variability of a gesture execution, to deal with noise or to accommodate imprecise sensor measurements. For instance, spatial specifications can be used to describe a virtual line between the pointing gesture of person to a target person. When this specification becomes true based on the input events, the system can conclude that one person is indicating another person. The movement of a finger or body part can also be described using spatial relations. These relations consist of the travelled path to which a gesture has to adhere. The path can be formed by constraining events in a spatial dimension such as $10 < event1.x < 50$. The use of relative spatial specifications, such as $event1.x > event2.x$, are also useful to describe fusion. Other spatial relations such as scale invariance deal with the recognition of a single gesture trajectory regardless of its scale. Similarly, rotation invariance is concerned with the rotation.

Spatial operators are domain specific and therefore need to be extendible by developers.

Temporal Specification

The fusion of events from multiple sources can depend on their temporal relation. An example of such a temporal relation is that two events should (or should not) happen within a certain time period. They are required in nearly all stream input such as audio, video and accelerometers [116]. For example, to select a person, one can confirm a pointing gesture with a speech utterance of “him” (or “her”) in parallel.

Spatio-temporal Specification

An arbitrary mix of spatial and temporal operators allows for an advanced spatio-temporal specification. Such a spatio-temporal specification allows one, for example, to distinguish between slow and fast walking [116]. Another example offering a closed loop spatio-temporal feature to describe that the beginning and end event of a gesture should be approximately at the same location, is provided by Khandkar et al. [91]. This spatio-temporal feature can be used to describe a recurring movement such as a circle dance. Other spatio-temporal fusion processes exist to analyse accelerometer data (i.e. velocity) and other input modalities [86, 116].

Identification and Grouping

The identification problem is related to the fact that sensor input does not always provide enough details to disambiguate a scenario. For example, Echtler et al. [45] demonstrate that two fingers from different hands cannot be distinguished from two fingers of the same hand on a multi-touch table due to the lack of shadowing information. Furthermore, when a finger is lifted from the table and put down again, there is no easy way to verify whether it is the same finger. Therefore, a double tap gesture cannot easily be distinguished from a two-finger roll. Similar problem emerge with other types of sensors such as when a user leaves the viewing angle of a sensor and comes back later. In these cases, the fusion of redundant input sensors (Section 2.1) can be used to address the identification problem.

The grouping problem can be considered as the simultaneous identification of two or more events. For instance, when multiple people are dancing in pairs, it is sometimes hard to see who is dancing with whom.

Therefore the system needs to keep track of alternative combinations for a longer time period to group the individuals.

Prioritisation and Toggling

Whenever two multimodal descriptions overlap, developers want to prioritise one above the other. The annotation of multimodal definitions with various priority levels is a first form of prioritisation. However, this requires knowledge about existing definitions. In case there are many, it might not be possible to maintain a one-dimensional priority schema. Nacenta et al. [119] for instance demonstrate that we should not distinguish between a scale and rotate multi-touch gesture on the frame-by-frame level but by using specialised prioritisation rules such as magnitude filtering or visual handles. As an alternative to prioritisation, the developer can decide to enable or disable (i.e. toggle) certain interactions based on the application context or input data.

Prediction

One of the major issues with multimodal disambiguation is that information in the near future can lead to a completely different interpretation of the interaction. An interaction definition can, for instance, fully overlap with a larger, higher prioritised definition. At a given point in time, it is difficult to decide whether the application should be informed that a particular interaction has been detected or that it is better to wait a bit longer. If future events show that the long-term interaction does not match, users might perceive the execution of the short-term interaction as unresponsive. Late contextual information might also influence the fusion process of primitive events that are still in the running to form part of more complex interactions.

Verification

A second pass can be applied to the combination of events which match a multimodal description. For example, an efficient segmentation approach may be combined with statistical analysis tools to verify whether the segmentation is accurate. Verification can also be used to further separate critical interactions. For example, deleting a movie from your hard drive is more critical than scaling a picture. Note that the concept of bundling classifiers (also known as ensembles) is already frequently used

in the machine learning domain, but it is underexplored in multimodal programming frameworks.

Uncertainty

Uncertainty can be explicitly supported in a framework by annotating all events with a confidence value. For instance sensors could provide the confidence level of multi-touch locations or limb positions. The fusion of those events could then combine these uncertainties to form a proper threshold. Confidence values can also trickle through higher levels. Unfortunately, this increases the complexity for developers.

User Profiling

To ease decision making, a multimodal framework should keep track of previous results in order to provide user profiling. User profiling is beneficial for multimodal descriptions which are known to cause confusion. For example, a multimodal description can be specified too precisely (i.e. when using exact temporal relations or without room for spatial approximation), causing poor recognition rates when used by many different users. The description can also be specified too loosely, causing accidental activations. Therefore, the profiling of users, such as tracking undo operations, could lead to an adaptation of the multimodal definition for that particular user.

2.3.4 Accessibility and Tooling

The final category deals with the accessibility and tooling support of an approach. A programming API can be provided in the form of a textual or graphical programming language. A programming language provides abstraction to a particular level of complexity. For example, does the language aim to support a wide variety of multimodal fusion tasks or is it a minimal language targeted for end users? In this category, we also assess the support for adequate tooling in order to debug and refine the multimodal processing tasks.

Readability

Kammer et al. [84] identified that multimodal and gesture definitions are more readable when concise terms are used (i.e. D instead of Down). They present a statistical evaluation of the readability of various gesture

languages that has been conducted with a number of students in a class setting. In contrast to the readability, they define complexity as the number of syntactic rules that need to be followed for a correct gesture description, including the number of brackets, colons or semicolons. Languages with a larger number of syntactic rules are perceived to be more complex.

Debuggability

In order to debug multimodal descriptions, developers apply pre-recorded positive and negative sample sets to see whether the given definition is compatible with the recorded data. This kind of simulation is rather primitive and lacks information to properly debug multimodal descriptions as it merely returns accuracy numbers. In order to increase debuggability, it might be interesting to explore more advanced debugging support such as notifying the developer of closely related (or overlapping) gesture trajectories. In this case developers could be informed that a particular gesture was rejected because a particular value (i.e. the x coordinate or timestamp) was off by a few units [107].

Authoring Support

Graphical authoring frameworks for defining multimodal interaction patterns provide tooling for both expert and non-developers. On the one hand expert developers use editor support to maintain an overview of the descriptions or use them to refine descriptions of audio processing or 3D gesture extraction tasks. Non-developers on the hand use graphical tools to express their multimodal interaction patterns without having to resort to powerful programming constructs. Authoring tools can provide an external representation [82], such as provided when using tablatures [95] or hurdles [92]. This external representation allows developers to further refine graphically defined constraints in programming code. When dealing with 3D input events from a Kinect, a graphical representation is valuable to get to the correct spatial coordinates, while temporal relations are easier to express in text.

2.4 Conclusion

In this chapter, we defined the terminology used throughout this dissertation. Furthermore, we contextualised our work and introduced three

levels of multimodal fusion, namely data-, feature-, and decision-level fusion, based on existing literature. Finally, we defined 30 criteria in four categories that characterise (1) the choice of a particular framework, (2) the implementation of the multimodal interaction and (3) novel approaches to solve open issues in gesture programming languages. These criteria will be used to evaluate the related work in the next chapter, as well as to reflect on the proposed solution in this dissertation.

3

Related Work

The goal of this dissertation is to design and implement a unified multimodal framework that enables fusion across the data, feature and decision levels. The contribution lies in the combination of a novel multimodal architecture, a high-level multimodal specification language and novel concepts to define interaction. In this chapter, we categorise two main strands of multimodal frameworks: data stream-oriented and semantic inference-based approaches. The former approach is better suited to continuously process a large number of input events. The latter provides abstractions for decision-level fusion and application integration. We then zoom into gesture specification languages as they expose many issues related to multimodal fusion. Afterwards, we generalise these gesture languages concerns to multimodal fusion descriptors. We conclude with a discussion on the different kinds of architectures used for multimodal interaction and show what challenges are still subject to research.

3.1 Data Streams and Semantic Inferencing

In this section we discuss two strands of multimodal frameworks, namely data stream-oriented frameworks and semantic inference-based frameworks. We show that existing fusion engines tend to rely on an ad

hoc approach when confronted with pieces of information coming from different abstraction levels, thereby losing their generality.

3.1.1 Data Stream-oriented Solutions

One approach to build multimodal interaction architectures is to assume a continuous stream of information coming from different modalities and to process them via a number of chained filters, as illustrated in Figure 3.1. This is typically done to efficiently process streams of high frequency data and to perform fusion on the data and/or feature level. Typical representatives of this strategy are OpenInterface [103, 142] and Squidy [97]. Each data-source is filtered by a data stream-pipeline and the resulting information is fused with other sources on an event-per-event basis.

Data stream approaches advocate the use of *composition boxes* in the form of a pipeline architecture [122]. Although this provides a form of composition, data stream approaches do not provide a fundamental solution to interlink temporal relations between multiple input sources. All incoming events are handled one by one and the programmer manually needs to take care of the intermediate results. This leads to difficulties in the management of complex semantic interpretations. Data stream-oriented architectures show their limits when high-throughput information such as accelerometer data (typically more than 25 events per second) should be linked with low-throughput semantic-level information such as speech (usually less than one event per second).

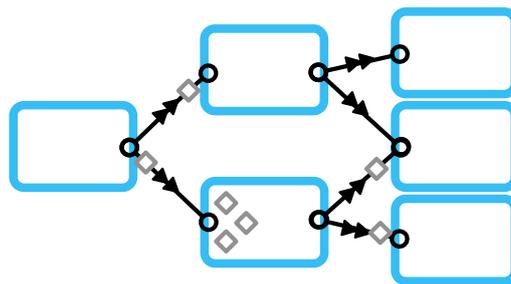


Figure 3.1: A pipeline architecture for data stream-oriented fusion

3.1.2 Semantic Inferencing Solutions

A second type of framework for multimodal interaction focuses on supporting fusion of high-level information at the decision level. These approaches

offer constructs to specify sets of information that is required before an action is triggered. Information gathered from the different input modalities is assumed to be classified correctly beforehand. These approaches work best with relatively low data frequency and highly semantic data.

Four classes of fusion algorithms are used to perform fusion at the decision level:

- **Meaning frame**-based fusion [158] uses data structures called frames for representing semantic-level data coming from various sources or modalities. In these structures, objects are represented as attribute value pairs.
- **Unification**-based fusion [78] is based on recursively merging attribute value structures to infer a high-level interpretation of user input.
- **Finite-state machine**-based approaches [76] model the flow of input and output through a number of *states*, resulting in a better integration with strongly temporal modalities, such as speech, because of their sequential flow. However, describing a flow of events with a non-trivial temporal ordering is much more difficult using state machines.
- **Symbolic/statistical fusion**, such as the Member-Team-Committee (MTC) algorithm used in Quickset [164] or the probabilistic approach of Chai et al. [22], is an evolution of standard symbolic unification-based approaches, which adds statistical processing techniques to the fusion techniques described above. These kinds of *hybrid fusion techniques* have been demonstrated to achieve robust and reliable results. However, the lack of support for segmentation is problematic to process information from novel sensors as they provide a continuous input stream.

These approaches have been shown to work well for fusing semantic-level events [39]. However, when confronted with lower level data, such as streams of 2D or 3D coordinates or data coming from accelerometers, semantic inference-based approaches encounter difficulties in managing the high frequency of input data [43]. In order to show their potential, these approaches assume that the different modalities have already been processed at the data level and therefore can reason over semantic-level information.

However, even when confronted with semantic-level data, several shortcomings can arise with existing approaches. First, they rely on the results of the modality-level recognisers without having the possibility to exploit the actual raw data. This can lead to problems in interpretation. For example, consider continuous gestures such as pointing in thin air. On the one hand, low level recognisers need to decide on a single *pointing* event, without contextual information or hints on the timing. This unlikely result in an appropriate event which can be used at a higher level. On the other hand, when continuous data is provided at a high frequency, decision-level fusion algorithms cannot handle the computational load. Therefore, existing decision-level algorithms only keep track of the latest events, which is problematic when fusing input from speech recognisers, which are known to introduce a lot of latency.

Second, as decision-level fusion engines assume that the creation of semantic events happens at a lower level, there is no (or very limited) control on what the refresh rate of these continuous gestures is. The typical solution for this scenario is to create a single pointing event every time the hand is detected to be in a steady state. Unfortunately this considerably slows down the interaction and introduces usability issues.

A third issue that arises when employing meaning frames or similar fusion algorithms is related to the previously discussed problem of overlapping matches. When a user aborts and restarts their interaction, thereby reissuing their command, the slot in the meaning frame is already occupied. For example, when a speech utterance of “hello” is repeated, the second event is refused by the algorithm. This results in a false negative as the time gap between the first speech utterance and newer pointing gesture becomes too large. The second speech utterance could also overwrite the first one, however, this means that another valid combination will be rejected without verification to the context.

Next to these three generic issues, we observe that finite-state machine-based approaches such as presented by Johnston et al. [76], typically lack the constructs to express advanced temporal conditions. The reason for this is that a finite-state machine (FSM) enforces the input of events in predefined steps (i.e. event x triggers a transition from state a to b). When fusing concurrent input, all possible combinations need to be expressed manually.

3.1.3 Irreconcilable Approaches?

In conclusion, data stream-oriented architectures are very efficient when handling data streams and semantic inference-based approaches process semantic-level information with ease. However, none of the presented approaches is efficient in handling both high-frequency data streams at a low level of abstraction and low-frequency semantic pieces of information at a high abstraction level. Moreover, the possibility to use data-level, feature-level and decision-level information on same stream of data at *the same time* is hindered by today's frameworks. By designing a unified fusion framework, the development of fusion across levels is encouraged, which allows the influence of decision-level information to make decisions at the lower-level. For example, derived user identification at the decision level can improve gesture or speech input. A second example is the analysis of target objects during free-air pointing to influence skeleton tracking. In the next section, we position the related work according to the criteria devised in Chapter 2. Afterwards, we describe a number of gesture and multimodal languages and emphasise their position in a few criteria. We then continue with a discussion on the architecture of multimodal fusion engines.

3.2 Positioning of Related Work

In this section we evaluate the related work to our criteria defined in Section 2.3. We performed a best-effort evaluation for the most relevant frameworks to our work. All results use the latest information available for each approach albeit some of them bundled (for example GDL (Echtler) and GiSpL are bundled as they are based on the same approach from the same authors). The goal of these figures is to identify the focus of existing work and to visualise open issues in multimodal framework design. As mentioned in Section 2.3, we devised 30 criteria by keeping track of features proposed by different approaches, including domains such as machine learning and template matching.

In Figures 3.2(a) to 3.2(h) we provide an indicative classification of existing gesture language-oriented frameworks, including GDL (Khandkar) [91], GeForMT [84–86], GDL (Echtler) [44,47], Proton [94,95], GestureAgents [80], EventHurdle [92,93], GestIT [148] and ICO [61]. In the context of this dissertation, we consider gesture frameworks as data-level frameworks.

Figures 3.3(a) to 3.3(g) demonstrate an indicative classification of generic multimodal frameworks, including QuickSet [27, 74, 164], MIML [102], PATE [129], OpenInterface [103, 142], Squidy [97], Hep-haisTK [39, 40, 42, 43] and DynaMo [5, 6].

The frameworks are sorted chronologically according to the date of their first publication. For each approach, we provide a score ranging from 0 (no support) to 5 (fully supported by a high level of abstraction) for every individual criteria, together with a short explanation (found in the dataset). The dataset and an up-to-date discussion of the axes is available online¹. A snapshot of the dataset is provided in Appendix C. It should be noted that most of our criteria can only be evaluated subjectively. However, results for the most recent approaches, including GeForMT, GDL (Echtler), GestIT, ICO, were peer reviewed and discussed with the corresponding authors during the workshop on Engineering Gestures for Multimodal User Interfaces (EGMI) [46]. An interactive visualisation of the criteria for each of the approaches can be accessed online².

¹Dataset for the evaluation of the criteria: <http://soft.vub.ac.be/~lhoste/research/multimodal/criteria/radarchart-mm/data.js>

²Interactive visualisation of the dataset: <http://soft.vub.ac.be/~lhoste/research/multimodal/criteria>

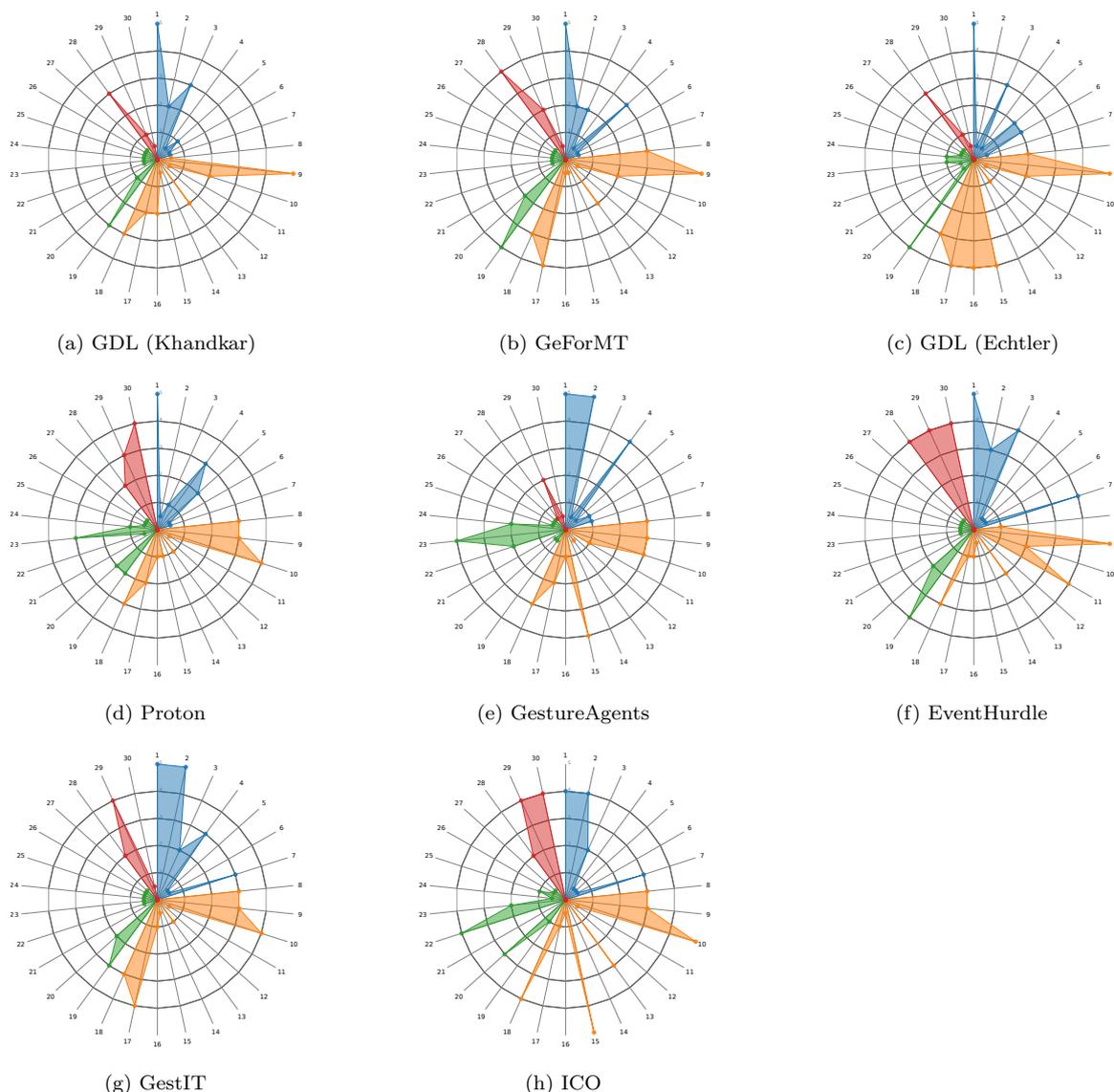


Figure 3.2: Indicative classification of gesture frameworks. The labels are defined as follows: (1) modularisation, (2) composition, (3) customisation and extensibility, (4) negation, (5) application symbiosis, (6) activation policy, (7) unbound variables and unification, (8) online processing, (9) off-line processing, (10) partial overlapping matches, (11) segmentation, (12) synchronising streams, (13) event expiration, (14) long-term reasoning, (15) concurrent interaction, (16) portability, serialisation and embeddability, (17) runtime definitions and device instantiation, (18) reliability and scalability, (19) spatial specification, (20) temporal specification, (21) spatio-temporal specification, (22) identification and grouping, (23) prioritisation and toggling, (24) prediction, (25) verification, (26) uncertainty, (27) user profiling, (28) readability, (29) debuggability, (30) authoring support.

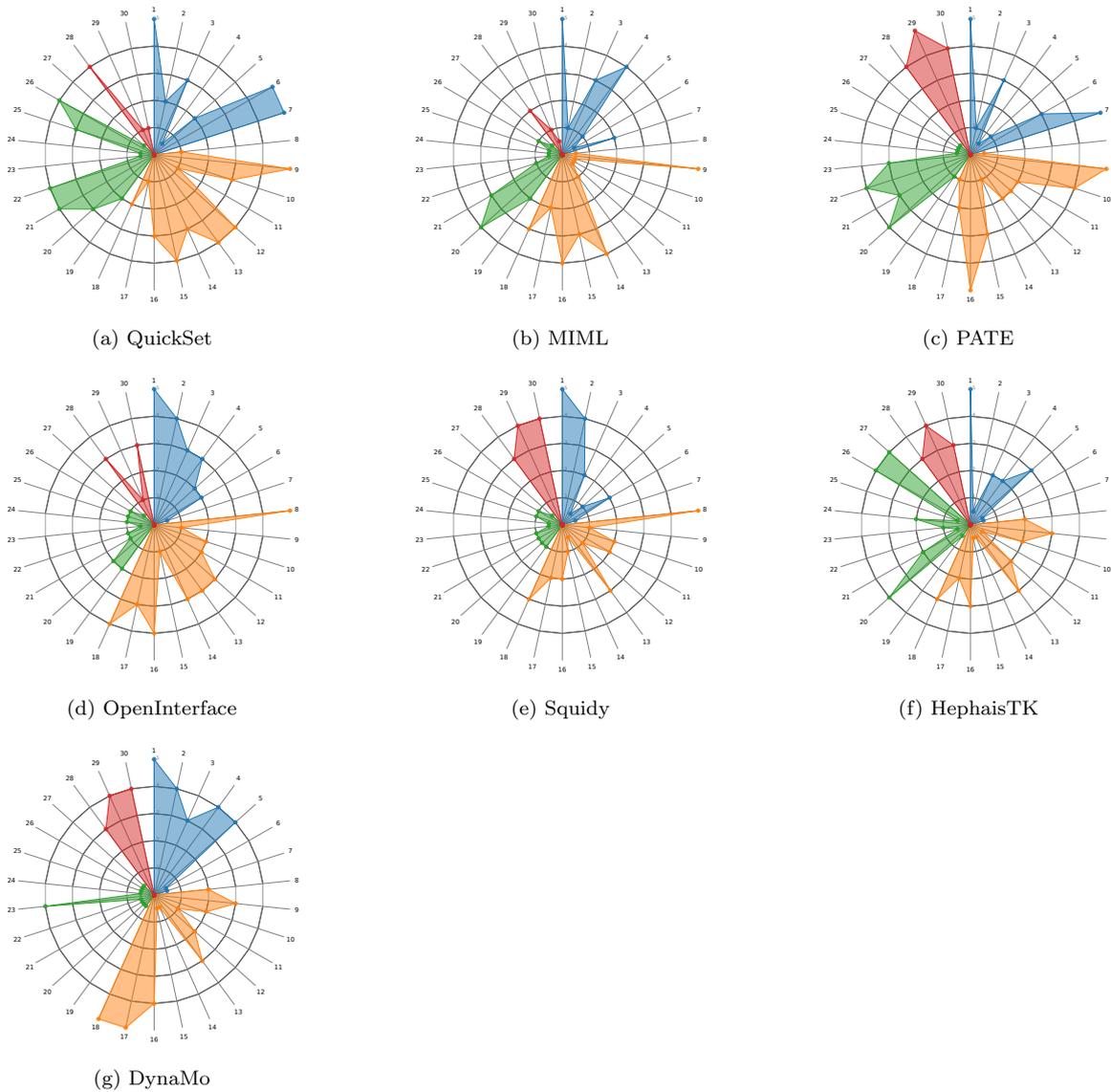


Figure 3.3: Indicative classification of multimodal frameworks. The labels are defined as follows: (1) modularisation, (2) composition, (3) customisation and extensibility, (4) negation, (5) application symbiosis, (6) activation policy, (7) unbound variables and unification, (8) online processing, (9) offline processing, (10) partial overlapping matches, (11) segmentation, (12) synchronising streams, (13) event expiration, (14) long-term reasoning, (15) concurrent interaction, (16) portability, serialisation and embeddability, (17) runtime definitions and device instantiation, (18) reliability and scalability, (19) spatial specification, (20) temporal specification, (21) spatio-temporal specification, (22) identification and grouping, (23) prioritisation and toggling, (24) prediction, (25) verification, (26) uncertainty, (27) user profiling, (28) readability, (29) debuggability, (30) authoring support.

3.3 Multimodal Languages

The goal of multimodal frameworks is to abstract the complexity of extracting meaningful information from multiple input event streams. These abstractions are found in various flavours, such as programming languages, application programming interfaces (APIs) and visual editors. In this section we first discuss a number of data-level fusion languages, in particular DSLs for gesture specification. Generic data-level multimodal frameworks focus on architectural abstractions instead of novel programming abstractions (Section 3.4). We then proceed with a discussion of feature- and decision-level programming abstractions.

3.3.1 Data-level Gesture Languages

Expressing data-level fusion is hard due to the low-level information and high-frequency of the data throughput. We further observe that programming languages for data-level fusion are underrepresented in existing multimodal solutions. Therefore this section discusses more general related work for fusing multiple raw touch streams, accelerometer data and full-body streams. We focus on these sensor inputs because their complex multi-stream behaviour (i.e. multiple fingers and limbs) resembles multimodal processing [61]. Similar to multimodal input, multi-touch events can vary according to the context, task, user and time [100].

Running Example

We present an example of a simple *hold-and-rotate* multi-touch gesture which we will use throughout this discussion in order to assess various aspects of existing gesture languages. The example is visualised in Figure 3.4 and is defined by the following low-level specification in plain English:

Find three events of three different fingers touching the surface in any order at about the same time. All these fingers should be located near each other (in a bounding circle). Two fingers should be approximately vertically aligned and the third finger should be on the left of the others. Find follow-up events of the two vertically aligned fingers such that it corresponds to a movement to the right (i.e. with a minimum and maximum x and y displacement). The third finger should remain approximately stationary, without lifting up during the time interval defined by the movement of the two other fingers.

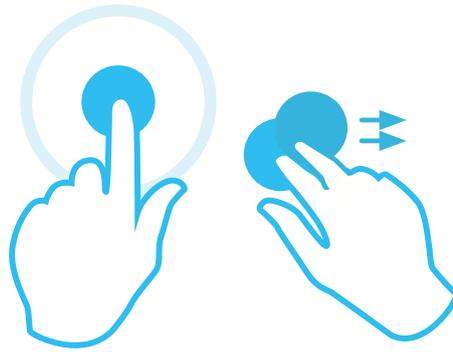


Figure 3.4: The hold-and-rotate multi-touch gesture

The hold-and-rotate gesture can, for instance, be used in 3D drawing applications such as AutoDesk³ to rotate objects along a fixed centre. The following paragraphs discuss existing gesture languages with respect to this running example.

Gesture Definition Language The Gesture Definition Language of Khandkar et al. (GDL-K) [91] focuses on four main areas: modularisation, flexibility, extensibility and hardware independence (the Gesture Definition Language was originally abbreviated to GDL, but we use GDL-K to differentiate between another approach called Gesture Description Language by Echtler et al. [47] later described in this section). The authors propose a DSL that allows developers to describe multi-touch gestures as pattern definitions rather than implementing them manually. Unfortunately we have been unable to draft an implementation for the running example due to the lack of information about the built-in functionality and lack of support for user-defined operators. An example lasso gesture is shown in Listing 3.1. It specifies a name (line 1) enabling the modularisation of gestures, a `validate` block (lines 2 to 8) which contains the logic for detecting the multi-touch gesture from raw data and a `return` command (lines 9 and 10) to express the result. Multiple primitive conditions are connected via an implicit logical AND operation, allowing easy customisation and refinement. We further observe the use of built-in functionality such as `Closed loop`, `length`, and `Enclosed area` to reduce the complexity for the programmer. However, this implies that many gestures require extensions (with a complex imperative implementation) to the interpreter. A single gesture can be composed of multiple `validate` blocks, which is useful to describe multi-stroke

³AutoDesk: <http://www.autodesk.com>

gestures. However, multiple gesture definitions cannot be combined, thus severely limiting the composition of complex multi-touch gestures. Khandkar et al. acknowledge that the order of conditions can significantly affect the overall performance of the system, as it evaluates the conditions in order in a lazy manner. A short discussion of GDL-K's position within the criteria outlined in Chapter 2, can be found in Appendix C.

Listing 3.1: Lasso GDL-K implementation

```

1 name: Lasso
2 validate
3   Touch state: TouchUp
4   Touch limit: 1
5   Closed loop and
6   Touch path bounding box: 200x200..1000x1000 and
7   Touch path length: 600..100000 and
8   Enclosed area: 5000x1000000
9 return
10  Touch points

```

GeForMT GeForMT [84–86] positions itself as a high level and accessible language for defining multi-touch gestures. The language offers a number of atomic building blocks such as shapes (for example a circle or line) and directions (such as north and southeast) that can be combined with relational operators (such as cross or spread). Custom shapes are supported by relying on the \$1 dollar recogniser [162] which is a template matcher (Section 3.3.3). However specific constraints such as the maximum horizontal displacement of a *swipe right* gesture or the vertical alignment of two fingers is not possible. Listing 3.2 demonstrates that the GeForMT language is a readable language [84] providing concise definitions of multi-touch gestures. Unfortunately, its simplicity allows too much variation on the gesture execution, which lowers the precision. For example, one cannot specify most spatio-temporal relations in the hold-and-rotate gesture such as the spatial neighbourhood and vertical alignment of the fingers or time intervals. Furthermore, the composition of gestures, segmentation and dealing with overlapping matches is currently not supported [84]. It should also be noted that the identification and grouping problem is inherently problematic due to the lack of references. For example, expressing a *line up* by finger *a*, followed by a *line up* by finger *b*, followed by a *line up* by finger *a* is not possible.

Listing 3.2: Hold-and-rotate GeForMT implementation

```

1 1F(HOLD(o)) + 1F(LINE(a)) * 1F(LINE(a))

```

GDL-E The Gesture Description Language (GDL-E, originally abbreviated as GDL but we suffix it due to the overlap with a previously discussed approach) by Echtler et al [44, 47] is another early approach that allows the decoupling of low-level touch gestures from the user interface code. A gesture definition consists of three main elements: regions, gestures and features. Rectangles or arbitrary shapes filter touch events based on its spatial coordinates. A region can then enable local gestures that in their term are defined by a number of features. For example, a two-finger horizontal swipe right gesture can be defined by means of a combination of the *ObjectCount* feature (expressing the need for two fingers) and a motion feature (to define a movement to the right). The implementation of the running example is finalised by describing the stationary finger using the *ObjectDelay* feature, as shown in Listing 3.3 (lines 4 to 6). Unfortunately, specifying spatio-temporal relations or dealing with finger identification is not supported.

Listing 3.3: Hold-and-rotate GDL-E implementation

```

1 hold-and-rotate
2   Flag OneShot
3   Region 0 0 100 20
4   ObjectDelay 255 1 1 0
5     Filter 131072 // object type
6     1000
7   ObjectCount 255 2 2 0
8   Motion 255
9     100 0 0 // lower boundaries
10    1000 10 10 // upper boundaries
11    0 0 0 // result (empty)

```

GDL-E offers a number of built-in features such as the path feature. It can be used to specify a gesture trajectory in greater detail. The patch feature enables a declarative specification of the trajectory in a number of points and calculates an overall distance score on how well a given gesture matches the definition. However, there is no refined control over minimum and maximum thresholds for each individual point. Furthermore, GDL-E does not support segmentation. GDL-E can process concurrent gestures and has a tight GUI integration with the application level with dynamically changing region specifications at runtime.

The Gestural Interface Specification Language (GISpL)⁴ [44] is a second iteration of GDL-E that focuses on providing the unambiguous description of gestures used in different multimodal interfaces. This includes multi-touch, digital pens, hand-held controllers or free air gestures.

⁴Gestural Interface Specification Language (GISpL): <http://www.gispl.org/>

GISpL provides additional features, such as an advanced delay operator, access to historical data and novel activation policies. The language is based on JSON to allow an easier integration with existing applications. However, it still lacks spatio-temporal operators between multiple features, which are required to implement the running example. In Listing 3.4 we show our best effort with GISpL. Unfortunately, due to the use of preprocessed features the `Scale` feature on line 15 will never reach the intended behaviour since one cannot express that it should exclude the hold finger to calculate the relative position between the two moving fingers. Without this exclusion the scale feature will erroneously result in a match if a finger moves towards the holding finger. We further elaborate on the necessity of user-defined spatio-temporal operators in the next approach on our list, namely Proton.

Listing 3.4: Hold-and-rotate GISpL implementation

```

1 { "name": "hold_and_rotate",
2   "flags": "oneshot",
3   "region":{
4     "id": "max_region",
5     "flags": "poly",
6     "filters": 2,
7     "points": [[0, 0], [0.2, 0.2]] // an array of x, y coordinates specify a region
8   },
9   "features": [
10    { "type": "Count",
11      "filters": 3,           // 3 fingers required
12      "constraints": [2,2,2], // 3 times bitmask '2' to represent touch events
13      "result": []
14    }, {
15      "type": "Scale",
16      "filters": 2,
17      "constraints": [-5,5], // two fingers should move without scaling
18      "result": []
19    }, {
20      "type": "Delay",
21      "filters": 1,
22      "constraints": [10,50], // temporal interval
23      "result": []
24    }
  ]
}
```

Proton Regular expressions are a common way to match text patterns. Proton [94, 95] extends the regular expressions semantics to multi-touch gestures. Developers can use Kleene operators to specify multiple occurrences and vertical bars to express the logical OR. The most important aspect of Proton is the detection of gesture conflicts at compile time. This vastly improves debugging capabilities and removes dealing with uncertainty at runtime. However, it also restricts a lot of potential gesture

definitions that could coexist without problems. Proton further assumes that only one gesture can be active at a particular time, limiting the runtime interaction. When a gesture cannot be expressed in a linear temporal relation, regular expressions quickly reach their limitations. For example, developers cannot express that the movement of a finger should overlap in time with another finger movement. Additionally, when defining gestures where multiple conditions are allowed to be executed first (a before b or b before a), Proton definitions suffer from a combinatorial explosion. This does not scale well for multi-stream processing where more than three fingers can move at the same time. Therefore the definition of the running example would require a major engineering effort and lose its readability.

To clarify the need for *unbound variables* and spatial and temporal operators, we analyse a simplified gesture shown by the Kin et al. [94] in Listing 3.5. In this example, two fingers should move towards each other. In Proton this can be denoted by concatenating different states of touch events, namely D when touching down on the surface with the finger, M for an arbitrary movement and U when lifting the finger up. These states can be superscripted and subscripted with attributes such as cardinal directions (i.e. north, east, south, west). Therefore, M^N expresses a movement to the north, M^E to the east, M^S to the south, M^W to the west, and no movement is defined as M^O . A second attribute is an incremental number for each finger on the display. For example, the expression $M_1^E M_2^W$ means that finger 1 should move east followed by the movement of finger 2 to the west. When either finger can move first, the developer needs to enrich the definition using the OR ($|$) operator: $(M_1^E | M_2^W)(M_1^E | M_2^W)$. However this allows the first finger to move twice to the east and thus does not define that both fingers should be moving towards each other. In order to specify that both fingers should move, we need to introduce an explicit M_1^E and M_2^W between the code resulting in a manual expansion of the definition (line 2 in Listing 3.5). The Kleene star operator ($*$) in the code allows capturing multiple move events from the same finger. The additional prefix attributes in Listing 3.5, such as $M^{L:}$ and $(M^{R:})$ are used to split the screen in a left and right part.

Listing 3.5: A Proton implementation of two fingers moving towards each other

```

1  $D_1^{L:O} M_1^{L:O} * D_2^{R:O} (M_1^{L:O} | M_2^{R:O}) *$ 
2  $(M_1^{L:E} (M_1^{L:E} | M_2^{R:O}) * M_2^{R:W} M_2^{R:W} (M_1^{L:O} | M_2^{R:W}) * M_1^{L:E})$ 
3  $(M_1^{L:O|E} | M_2^{R:O|W}) * (U_1^{L:O|E} M_2^{R:O|W} * U_2^{R:O|W} | U_2^{R:O|W} M_1^{L:O|E} * U_1^{L:O|E})$ 

```

We can observe multiple facts from the declarative Proton code in Listing 3.5:

1. The developer needs to express that a finger needs to move to the east. Unfortunately, Proton does not provide scoping mechanisms to express a spatial constraint about the movement of a particular finger. Therefore, the first character in the superscript (E) and the number in the subscript attribute (1) are repeated for every state (this is similar for the other finger, namely W and 2). This is error prone, redundant and makes the code hard to read. This example shows why custom expressions are necessary: developers want to express that a certain finger identifier should be linked to a certain attribute for the whole gesture (or a part of it).
2. Identifying fingers by an explicit number leads to limitations on concurrent gesture execution. When a second user puts down extra fingers on the touch screen at the wrong time, the combination of fingers number 1 and 2 can map to different users. At another time, fingers of the second user get assigned the numbers 3 and 4, which are not supported by the definition. The presence of finger numbers in Proton code is caused by a leaky hardware abstraction. In terms of Chapter 2, this problem is attributed to the lack of *unbound variables* and the possibility to easily *identify and group* events because the numbers have to be explicitly written down in the code.
3. Line 1 is incomplete because $D_1^{L:O}$ specifies that the left finger should touch the surface first. Ideally, the statement should be replaced with $(D_1^{L:O} | D_2^{R:O})$ to also allow the touch down of the right finger before the left. However, this requires an update the second and third statement ($M_1^{L:O}$) with a combinatorial expansion. This problem is attributed to the lack of *temporal operators*. Whenever a combination of events can happen at the same time, a combinatorial expansion of the current and potentially following code is required. Therefore, the accidental complexity of Proton increases when the execution order of a gesture needs to be flexible.
4. The use of cardinal directions is suboptimal. It forces developers to use explicit labels (i.e. south or west), abstracts too much information (i.e. the distance and speed is not accessible for the developer) and requires a system-wide slack variable to segment the event

stream into pieces. This limits the ability to specify crucial properties such as the minimum and maximum distances. Furthermore, cardinal directions are not sufficient to express curvy gestures. This is a common problem with approaches that rely on quantisation processes. Quantisation processes transform a set of input values into a particular value, thereby losing potentially valuable information, especially when performed system-wide without support for customisation.

5. In Proton, cardinal direction attributes are calculated between two frames causing sensitive sensors to invalidate the definition due to jittering (i.e. a single noise event such as M_1^N or M_1^S invalidates the entire gesture presented in Listing 3.5). Many sensors are susceptible to this behaviour and gesture languages should therefore deal with this form of *uncertainty*. Proton does not.
6. To support a pinch gesture, Proton provides additional superscript attributes (i.e. M^P for pinch and M^S for spreading). However these attributes are preprocessed on a global level without context, meaning that they will not work when three or more fingers are on the screen since most movement with three fingers results in both pinching and spreading. This issue can be solved by providing spatio-temporal operators that can be defined by the developer, rather than providing a limited set of quantisation features⁵.
7. Extending Proton with additional attributes is done in the host language. However, implementing these attributes is hard in imperative languages, as discussed in Chapter 2, as they require to manually manage intermediate states [141, 148].
8. The left ($M^{L:}$) and right ($M^{R:}$) attributes are used when splitting the screen in two. This is Proton's primitive manner of grouping. In this case it supports the *left* and *right* finger. However, statically splitting the screen is not flexible enough to support multiple users in many other scenarios.
9. A final observation from this code is about the typability of Proton. Most programming languages require ASCII- or Unicode-based code. Therefore the integration of these definitions in existing code bases

⁵Furthermore, the attribute S for spreading is overloaded in the existing Proton examples, as the S symbol was already used to denote the south direction.

reduces the readability of Proton code. Listing 3.6 demonstrates the \LaTeX variant of the programming code.

Listing 3.6: Demonstrating the typeability of a Proton implementation of two fingers moving towards each other in \LaTeX

```

1 $D_1^{\{L:O\}} M_1^{\{L:O\}} * D_2^{\{R:O\}} (M_1^{\{L:O\}} | M_2^{\{R:O\}}) * ... $
2 $(M_1^{\{L:E\}} | M_1^{\{L:E\}} | M_2^{\{R:O\}}) * M_2^{\{R:W\}} M_2^{\{R:W\}}
3 (M_1^{\{L:O\}} | M_2^{\{R:W\}}) * M_1^{\{L:E\}} ... $
4 $(M_1^{\{L:O|E\}} | M_2^{\{R:O|W\}}) * (U_1^{\{L:O|E\}} M_2^{\{R:O|W\}}
5 * U_2^{\{R:O|W\}} | U_2^{\{R:O|W\}} M_1^{\{L:O|E\}} * U_1^{\{L:O|E\}}) $

```

These nine observations give a detailed insight on some of the remaining challenges of gesture language research. In particular, readability [84], temporal expressions and concurrent interaction are poorly supported in Proton. Therefore Kin et al. [94] additionally present tablatures, a graphical interface to define gestures, as further discussed in Section 3.3.2.

GestureAgents GestureAgents [80] is a framework that primarily focuses on gesture disambiguation on a single multi-touch table. Developers can coordinate input events to obtain concurrent multi-user multi-tasking interaction. It is one of the few frameworks providing programming abstractions for gesture disambiguation as observed in the comparison figures (Figure 3.2(e)). The basic concept consists of of an *agent* and a *recogniser*. Every agent emits events to a recogniser and is responsible for maintaining the axiom that one input event can only be part of one gesture. Agents can be composed of other agents to support *modularisation* and *composition*. The authors assume the existence of the implementation of a gestures and therefore do not provide any abstractions to specify gestures. Their focus lies on the disambiguation between gestures by proposing four recognition states, as illustrated in Figure 3.5⁶.

1. The *initial state* is used when the recogniser is waiting for new agents. Agents control the event sources and make it possible to define that an input event can be part of only one gesture.
2. When an agent is acquired and provides an event source, the recogniser starts matching potential gestures. This is called the *evaluation state*. This state should be as short as possible because it blocks the activation of other recognisers that are eager to transition to the recognition state using the same events. Therefore the evaluation state can be seen as an early spotting process.

⁶Figure 3.5 reproduced with permission of Carles Fernández Julià

3. The *recognition state* is used when a recogniser is confident that the input events match its gesture definition. The engine sticks to the decision and waits for final events or activates the gestures callbacks in an online manner.
4. Recognition *fails or finishes* when incompatible events are captured or when a timeout expires.

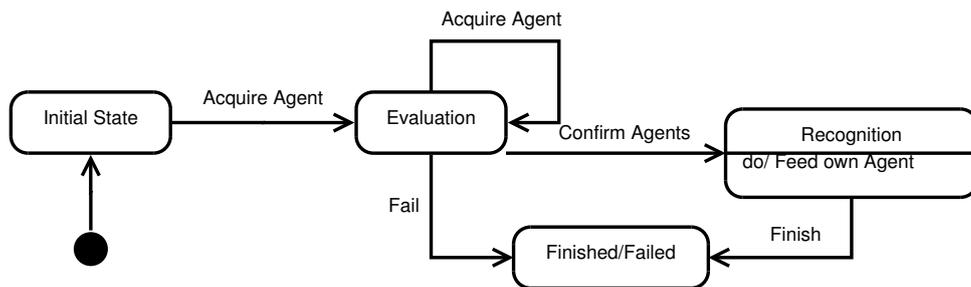


Figure 3.5: State diagram of GestureAgents recognisers

With these abstractions is it possible to disambiguate between a tap and double tap gesture since the double tap agent will not release the input events to trigger a tap as long as the double tap is a potential gesture candidate. Therefore the timeout slack variable should be as low as possible. To detect a double tap in a multi-user scenario, additional abstractions are required. The tap of a second user right after the tap of the first user might incorrectly invalidate the double tap recogniser because the spatial distance between the taps is too large. Therefore GestureAgents provides a technique of competition called Agent Exclusivity. When considering alternative gestures duplicates of the recogniser instances are taken and each of these instances must compete for exclusivity. Acquiring contextual information can also be used to discriminate between gestures. Exclusivity abstractions and duplication features allow developers to deal with *future events*, *prioritisation* and offers a primitive form of *grouping*. However, it requires quite complex code logic in order to know whether a given event is a potential candidate for grouping. A lot of combinations are possible, potentially affecting the performance. Furthermore implementing and evaluating all these combinations at design time is a challenging task. Unfortunately there are no abstractions for gesture *specification* and therefore we are unable to provide an implementation for the running example without major engineering effort. Developers are required to implement their gestures in an imperative programming language (Python) without additional language or API support.

GestIT Proton detects conflicting gestures at compile time by scanning for identical prefixes. However when conflicts are detected the developer is required to adapt (and typically unify) their implementation to remove the ambiguity. This ruins composition as argued by Spano et al. [148] as multiple gesture definitions are merged. GestIT uses a Proton-compatible declarative language with the following operators: iterative, sequence, parallel, choice, disabling and order independence. The disabling operator can be used to stop iteration and the order independence operator partially solves the manual combinatorial expansion problem Proton suffers from. However, expressing temporal relations such as defining a minimum or maximum time range between constraints is still not supported. Furthermore, GestIT proposes a different solution than Proton in order to deal with partial overlapping gesture definitions. The overlapping gesture definitions are executed in parallel and as soon as a gesture activates it will inform the application. When the slower gesture also activates using similar input events, it will activate with an extra parameter to list the gestures it conflicts with. This solves (1) the problem of implementing partial overlapping gestures; (2) latency problems from the user perspective; and (3) the context-less execution of two different gesture activations at the application side. This implies that developers may have to revert state at the application level. This results in non-trivial code and might not be possible for some cases.

In Listing 3.7 an implementation of the hold-and-rotate gesture is presented. Line 1 uses the order independence operator to allow every finger to touch first. However it should be noted that the numbering of the fingers directly maps to the identification schema of the underlying multi-touch sensor. The authors propose to extend the library to verify all combinations to resolve the *identification* and *unbound variable* problem. Custom spatial operators such as `bbox` (a rectangular bounding box boolean function), `vc` (checking the vertical alignment) and others are used to express the movement of the fingers. These boolean functions are implemented in the host language and have access to historical data within a predefined time window.

Listing 3.8 shows the integration of GestIT into a XAML user interface application⁷. Line 6 registers a callback function that will be triggered as soon as the first part of the gesture gets detected. In this way fine-grained incremental feedback can be sent to the application level. However it

⁷Extensible Application Markup Language (XAML): <http://msdn.microsoft.com/en-us/library/ms752059.aspx>

does reduce the reusability of the gesture definition because it is highly case specific.

Recently, the authors of GestIT validated their abstractions with the CARE [147] properties. They model complementary, assignment, redundancy and equivalence using the available operators in GestIT, paving the way to incorporate additional input modalities.

Listing 3.7: Two fingers moving towards each other GestIT implementation

```

1 (Start1[bbox] |= Start2[bbox ∧ vc] |= Start3[bbox ∧ vc]) >>
2 (Move2*[rx ∧ dist] || Move3*[rx ∧ dist])[>
3 (End1 |= End2 |= End3)

```

Listing 3.8: Integrating GestIT code into a XAML application

```

1 <g:GestureDefinition x:Name="moveSelection">
2   <g:Sequence Iterative="True">
3     <!-- turn (front of the screen) -->
4     <g:Change Feature="ShoulderLeft" Accepts="screenFront">
5       <g:Change.Completed>
6         <g:Handler method="screenFront.Completed"/>
7       </g:Change.Completed>
8     </g:Change>
9     <g:Disabling>
10    <!-- grab gesture -->
11    <g:Disabling Iterative="True">
12      <g:Change Feature="HandRight" Iterative="True">
13        <g:Change.Completed>
14          <g:Handler method="moveHand.Completed"/>
15        </g:Change.Completed>
16      </g:Change>
17      <g:Change Feature="OpenRightHand" Accepts="rightHandClosed">
18        <g:Change.Completed>
19          <g:Handler method="rightHandClosed.Completed"/>
20        </g:Change.Completed>
21      </g:Change>
22    </g:Disabling>
23    <!-- turn (not in front of the screen) -->
24    <g:Change Feature="ShoulderLeft" Accepts="notScreenFront">
25      <g:Change.Completed>
26        <g:Handler method="notScreenFront.Completed"/>
27      </g:Change.Completed>
28    </g:Change>
29  </g:Disabling>
30 </g:Sequence>
31 </g:GestureDefinition>

```

ICO Interactive Cooperative Object (ICO) is a formal description technique based on Petri nets to model multi-touch and multimodal interaction. ICO offers reachability graphs, invariants, liveness, consistency and precedence properties to assist the developer in *debugging*. It also increases

the *reliability* of the system. By offering explicit fork and join operations, ICO supports *overlapping* gestures and *concurrent interaction*. A final interesting property of ICO is the use of dynamic finger clustering to deal with the *grouping* problem. Unfortunately due to the use of Petri nets, the modelling of all possible transitions has to be complete. Therefore ICO is labour intensive and requires a visual editor to keep track of all cases. On the data-level, ICO does not offer spatial abstractions to ease the development of gestures. The ICO editor is not freely available. Therefore we cannot provide an implementation of our hold-and-rotate example.

3.3.2 Gesture Authoring

When input events can be visualised in an accessible manner, *visual* programming abstractions can be used to define patterns. Such tools allow developers to visually define and modify gesture definitions, as discussed in the following paragraphs.

Event Hurdle With Event Hurdle [92], programmers use a visual markup language to draw directed line segments (i.e. a so-called hurdle is represented by a blue line in Figure 3.6) that the sensor trajectory should pass. This is useful for quick prototyping as it requires no programming knowledge and the concept is easily understood [92]. An example of the coding process is shown in Figure 3.6. Additional compositions such as serial, parallel and recursive are supported. Furthermore, the use of *false hurdles* allows developers to express that certain trajectories should be avoided (i.e. *negated*). The recognition engine is based on finite-state machines but it is unclear how accessible and modifiable the generated code is. Unfortunately the support for multi-touch and multi-stream gestures is very limited and our running example could not be implemented due to the lack of temporal abstractions to link multiple finger strokes. This is caused by the fact that hurdles have a strict ordering.

Tablatures As mentioned before, gestures implemented in Proton can be challenging to read. Therefore the authors proposed a visual alternative called tablatures [94]. A tablature is the visual representation of a Proton implementation and helps developers in identifying temporal patterns in a gesture. An example is shown in Figure 3.7. In this gesture, the first finger touches a delete icon (represented by label *d*) while the second finger

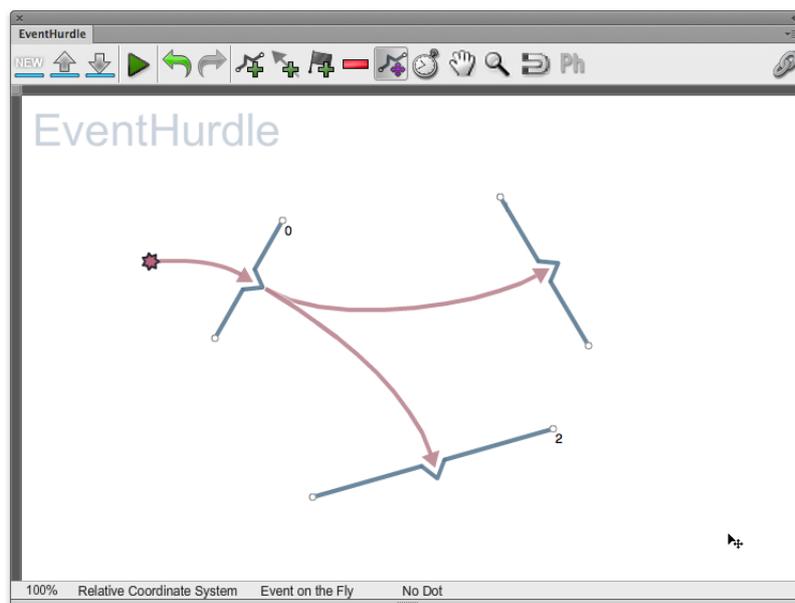


Figure 3.6: A gesture implementation based on two event hurdles

selects (*s*) one or multiple items (*) that should be removed. In their evaluation Kin et al. [94] show that tablatures are easier to understand than the regular expressions of Proton and its Objective-C variants.

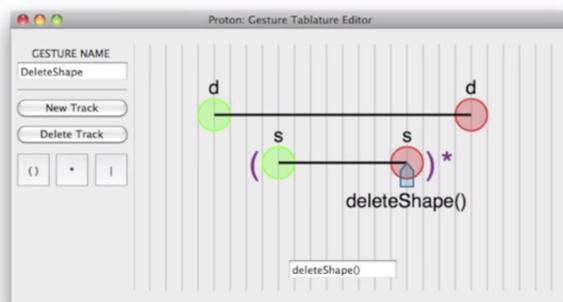


Figure 3.7: Implementing a delete gesture using tablatures

3.3.3 Template Matching and Machine Learning

Instead of relying on a language to describe gestures, algorithms such as Rubine [137], Dynamic Time Warping [32], the \$1 recogniser [162] and Protractor [106] recognise gestures by calculating a distance score between a set of input events and one or more pre-recorded samples. These samples form the *template* of a gesture. A benefit of template

matching is that no programming knowledge is required. The user can register a new gesture by executing it multiple times and labelling it with a name. A user then needs to set a threshold that will trigger the action of the gesture if the distance between the input and the sample is lower than that threshold. However, the recognition process slows down with an increasing number of samples. Template matching is frequently used for single or multi-stroke [4] but does not perform well for multi-touch gestures [109].

In contrast to template matchers, machine learning solutions distil a single model from the set of samples. A training procedure tries to extract the most characteristic features for each gesture. Neural Networks (NN), Hidden Markov Models (HMM) and Support Vector Machines (SVM) are typical examples of such learners. These approaches are very popular due to their ability to learn complex patterns. Furthermore, they perform well if many samples are available since a single model of the gesture is distilled which avoids comparing the given input events to all templates at runtime. However, learning approaches require an extensive amount of training data [48, 104].

In terms of multimodal interaction, Van Seghbroeck et al. [155] target the entire chain, from the creation of profiles for various input modalities to classification using HMM. A major disadvantage of their approach is that they do not allow human developers to verify nor modify the results produced by classification algorithms. This is due to the lack of an *external representation* [82]. Furthermore example-based approaches provide little control to developers [108]. Lü et al. [108] argue that developers should be able to provide more information about the gestures. A first example is given in Figure 3.8 where a triangle and a sector can be disambiguated by describing that they exist out of 3 lines versus two lines and an arc. Lü continues with a second example about a spring gesture that may contain a varying number of zigzags. In order to recognise this gesture, all variations must be part of the sample set, which requires a lot of effort and often results in poor definition. Finally, some gestures require that some attributes such as the direction of an arrow need to be part of the recognition result. Template matchers and machine learning-based approaches often fail to provide this kind of valuable information, as they only return a score.

A number of frameworks, such as iGesture [145] and Weka [59] provide a uniform API for a number of template matchers and machine learning solutions. In particular, the iGesture framework bundles classification algorithms such as DTW, Rubine [137], SiGeR [152] and Hidden Markov

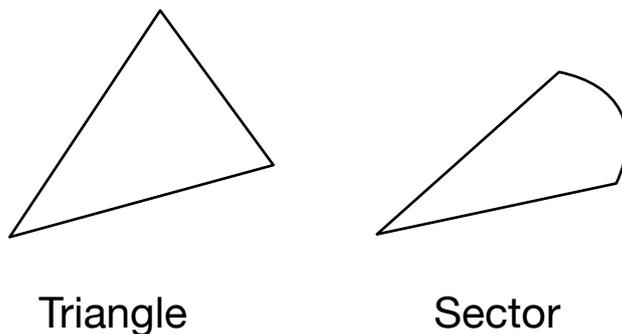


Figure 3.8: A triangle and a sector gesture

Models. This enables developers to experiment with multiple algorithms to obtain the best classification results.

3.3.4 Decision-level Multimodal Languages

In this section we broaden the scope to decision-level multimodal fusion language abstractions. In a recent survey, Dumas et al. [41] demonstrate the lack of higher-level programming tools for multi-level multimodal interaction. The most notable decision-level approaches are QuickSet [27], MIML [102], XISL [88,89] Emma [75], HephaisTK [39] and Ensemble [15].

QuickSet QuickSet [27,78,164] offers unification operators to denote *unbound variables*. These unification operators allow developers to fuse partial information from multiple input modalities and provide four main benefits: *partiality*, *mutual compensation*, *structure sharing*, and *multimodal discourse*. A *partial* description can be used when certain aspects are underspecified such as the location of a pointing gesture when pronouncing *here*. In this case both the pointing gesture and the speech command are required but the location is not specified. *Mutual compensation* is active when certain aspects of the combination of gesture and speech are meaningful or used in a redundant manner. For instance it is common for speech recognition to falsely recognise words, but these can be rejected if there is no corresponding pointing gesture. The authors describe *structure sharing* as an advantage when additional information can be extracted from the unification. For instance when a user utters *facing this way* while gesturing an arrow, the orientation feature of the gesture is automatically instantiated. Finally, Quickset offers a conflict resolution schema called *Members-Teams-Committee* that uses multiple

agents to calculate different recognition candidates in order to obtain a weighted decision. A similar approach is used by Flippo et al. [53].

Listing 3.9 shows an example of a QuickSet rule. In this example, we observe limited expressiveness on the *spatial*, *temporal* and *spatio-temporal* criteria, as operators (such as `overlap` and `follow`) are built-in instead of user-defined. Additionally, *unbound variables* are offered via numbers, which makes the code prone to errors and hard to understand. The authors demonstrated that unification is a fundamentally powerful construct in multimodal interaction, but major improvements on the language expressivity are still open for future research.

Listing 3.9: QuickSet rule

$$\left[\begin{array}{l}
 \text{lhs :} \\
 \\
 \text{rhs :} \\
 \\
 \text{constraints :}
 \end{array} \right. \left. \begin{array}{l}
 \left[\begin{array}{l}
 \text{cat : command} \\
 \text{content : [1]} \\
 \text{modality : [2]} \\
 \text{time : [3]} \\
 \text{prob : [4]}
 \end{array} \right] \\
 \\
 \left[\begin{array}{l}
 \text{dtr1 :} \\
 \left[\begin{array}{l}
 \text{cat : located_command} \\
 \text{content : [1] [location : [5]]} \\
 \text{modality : [6]} \\
 \text{time : [7]} \\
 \text{prob : [8]}
 \end{array} \right] \\
 \\
 \text{dtr2 :} \\
 \left[\begin{array}{l}
 \text{cat : spatial_gesture} \\
 \text{content : [5]} \\
 \text{modality : [9]} \\
 \text{time : [10]} \\
 \text{prob : [11]}
 \end{array} \right]
 \end{array} \right] \\
 \\
 \left. \begin{array}{l}
 \text{overlap}([7], [10]) \vee \text{follow}([7], [10], [4]) \\
 \text{total_time}([7], [10], [3]) \\
 \text{combine_prob}([8], [11], [4]) \\
 \text{assign_modality}([6], [9], [2])
 \end{array} \right\}
 \end{array} \right]$$

MIML Latoschik [102] demonstrates the use of temporal Augmented Transition Networks (tATNs) for multimodal interaction. These temporal extensions support complex *temporal* relations in multimodal descriptions, as Vanderdonckt et al. [100] argued: “*finite-state automata or Augmented Transition Network do not allow for the representation of concurrent behaviour and thus do not make it possible to express constraints such as ev1 and ev2 can occur at the same time*”. This is related to the combinatorial expansion observed in approaches as Proton (Section 3.3.1).

However, manually constructing and maintaining tATNs turned out to be difficult and was limiting the expressiveness of the interaction. Therefore, a multimodal markup language (MIML) was developed to *significantly expand the vocabulary as well as the grammar* [102].

An example of a multimodal description in MIML is shown in Listing 3.10. The temporal relations can be found on lines 2 and 15. However, nesting temporal expressions quickly becomes complex for nontrivial examples. This is similar as the GestIT approach explained before. Furthermore, when *customising* interaction patterns, the order of nesting conditions becomes inherently difficult as a deep understanding of all sub-components is required. Other abstractions to deal with spatial relations or user-defined constraints need to be implemented in the host language. However, such a host language does not provide adequate abstractions to minimise the accidental complexity. As shown on line 4, developers need to call functionality of the host language to describe that both the ‘rotate’ and ‘turn’ utterances can be used as a valid speech input. For the same reason we argue that the specification of *spatial* relations is not properly supported in MIML.

The implementation engine of MIML supports properties such as *negation*, access to historical information for *long-term reasoning* and the processing of *concurrent interaction*. These elements ease the fusion of decision-level multimodal input. However, adequate programming abstractions and an architecture to support fusion on data- and feature-levels are still lacking, as mentioned in the paper:

(...) Whether such a (multimodal) grammar exists is still an active research topic (...) The problem of finding a clean structure in the gesture stream⁸ might be one of the reasons for the sometimes vague or even contradictory results reported in the context of cross-modal temporal relations. Nevertheless there are some relations that hold and that should be considered when developing tools for the design of multimodal interfaces. This includes support for temporal constraints⁹ between input streams of varying granularity, incorporation of integration methods based on the inputs semantic content, and — regarding the HCI utilization — access to information from the application context level¹⁰.

⁸Also known as segmentation

⁹Corresponds to the temporal specification criterion defined in Chapter 2.

¹⁰Corresponds to the application symbiosis criterion defined in Chapter 2.

(Latoschik, 2002)

Listing 3.10: MIML

```

1 <description>
2   <temporalrelation type="sequential">
3     <speech>
4       <function name="rotateAction"/> <!-- This function returns true if --!>
5     </speech>                               <!-- event.equals("rotate") || event.equals("turn") --!>
6     <requires>
7       <function name="objectDescription"/>
8       <fill-slot source="identifier" target="object"/>
9     </requires>
10    <select>
11      <choice>
12        ...
13      </choice>
14      <choice>
15        <temporalrelation type="overlap">
16          <speech>
17            <function name="modalAdverb"/>
18          </speech>
19          <gesture>
20            <function name="rotating"/>
21            <exec-on-start>
22              <apiCommand name="rotateObjectByHand-On"/>
23            </exec-on-start>
24            <exec-on-end>
25              <apiCommand name="rotateObjectByHand-Off"/>
26            </exec-on-end>
27          </gesture>
28        </temporalrelation>
29      </choice>
30    </select>
31  </temporalrelation>
32 </description>

```

PATE PATE [129] is a goal-oriented rule-based approach with support for conflict resolution. Rules are written in XML form. They contain conditional elements and built-in temporal functions. The conditional elements are satisfied when a corresponding event is found in the working memory. Temporal functions such as **before** and **sameTime** are provided by the engine. They cannot be extended. An example is shown in Listing 3.11 where the content of a speech utterance is unified with a gesture. It is interesting to note that working elements can be contained within each other. For instance the type ‘Speech’ may contain an element of the type ‘Word’ in the slot ‘content’ which provides a string of the uttered word. PATE further provides an *authoring* tool to inspect the working memory and which rules are activated. This is useful for *debugging* and fine-tuning. Each rule has a weight parameter to define its *priority*.

When multiple rules are satisfied, only the one with the highest score will be activated. When two rules have an equal score, the behaviour is randomized but consistent until the system restarts. Unfortunately, due to the missing functionality such as *composition*, *spatio-temporal* and *user-defined operators*, PATE fails to provide abstractions that transcends the decision-level fusion.

Listing 3.11: PATE

```

1 <rule name="redundantInput">
2   <comments>identifies redundant information</comments>
3   <activation>0.8</activation>
4   <conditions>
5     <condition name="goal">
6       <object type="Speech">
7         <slot name="content">
8           <variable name="content"/>
9         </slot>
10      </object>
11    </condition>
12    <condition name="wme1">
13      <object type="Gesture">
14        <slot name="content">
15          <variable name="content"/>
16        </slot>
17      </object>
18    </condition>
19  </conditions>
20  <actions>
21    <action name="pop"/>
22    <action type="output" name="content"/>
23  </actions>
24 </rule>

```

HephaistTK Dumas et al. [39] present an XML-based language (Synchronized Multimodal User Interfaces Markup Language: SMUIML) to describe advanced dialogue interaction. In their work, the focus lies on describing qualitative temporal relations and state transmissions for multimodal decision fusion. Recently, the authors have extended this approach with quantitative temporal relations that can be trained using HMMs [42]. Furthermore, a graphical user interface was built to overcome the verbosity of the XML descriptions. However, this approach is focused on decision-level fusion and does not provide abstractions for data- or feature-level fusion. Required characteristics such as overlapping support, or stream-oriented algorithms are not provided. In general, we observe that existing decision-level frameworks fall short on the ability to ignore events that at first sight fit the description but can later be replaced by an event even better suited for fusion.

Multimodal Authoring Tools

As an alternative to programming interaction patterns, visual authoring tools can be used. Prime examples are IMBuilder/MEngine [13] ICARE [12], OpenInterface/Skemmi [103, 116, 142] and HephaisTK [43]. As argued by Dumas et al. [43] *‘graphical tools for designing multimodal interfaces can be broadly separated in two families: stream-based approaches and event-driven approaches’*. The former approaches typically use processing blocks to filter information, which can be graphically linked in multiple ways as shown in Figure 3.9. The latter approaches focus on low-frequency decision-level data and are represented by state machines or meaning frames (see Figure 3.10). Dumas et al. further note that graphical editors are built without an underlying formal language except for Petshop for ICO [120], MultimodalXML [149] and HephaisTK. For instance the XML format of NiMMiT is defined based on the functionality offered by the authoring tool [33]. The importance of an external representation of authoring tools is confirmed by a recent survey on graphical toolkits for multimodal systems by Cuenca et al. [31]. Cuenca et al. further conclude that the ease of use of authoring tools has not been extensively studied. Additionally, we are not aware of the existence of authoring tools for a particular input modality based on multimodal abstractions. Either authoring tools are modality-specific without the ability to fuse information from other input devices (such as EventHurdle [92]) or tooling is provided to build an entire multimodal architecture without a focus on processing single modalities (for instance Squidy and HephaisTK).

3.4 Multimodal Architectures

As mentioned in Section 3.1, multimodal frameworks can be classified in two main strands: data stream-oriented architectures and semantic inferencing methods. In this section we focus on the architecture and implementation of these two variations.

3.4.1 Data Stream-oriented Architectures

Data stream solutions rely on a pipeline architecture whereby composition boxes are chained together. The focus lies on processing input events as efficiently as possible. Therefore, these approaches are well suited for data-level fusion. In this section we discuss OpenInterface and Squidy, two prime examples of data-level fusion frameworks.

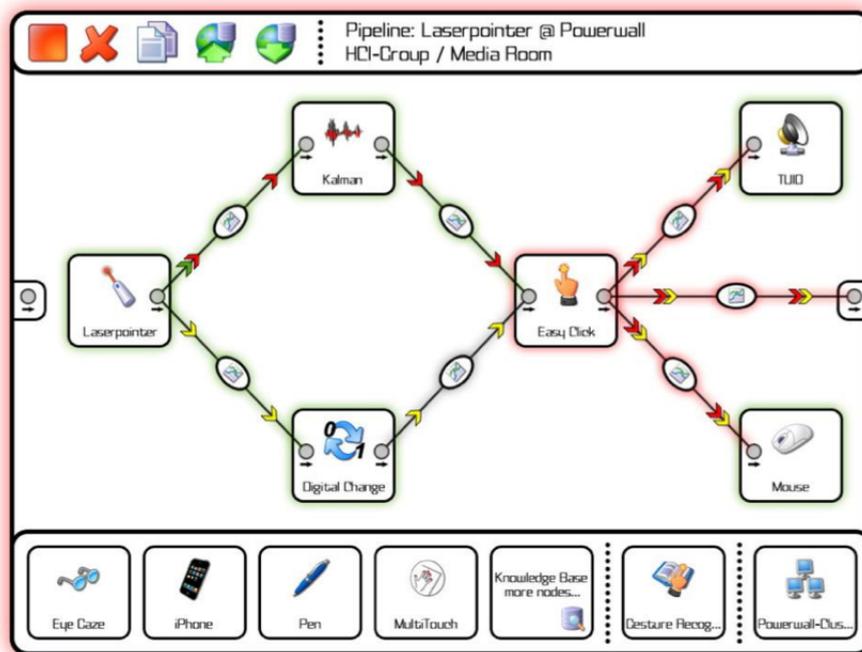


Figure 3.9: Filtering Wii remote data to simulate mouse input using Squidy

OpenInterface OpenInterface [142] interconnects “components” (i.e. a composition box) via pipes. Components such as built-in filters (e.g. thresholds and averages), libraries and external TCP/IP services are supported. This allows the framework to reuse the existing functionality of a large *kernel*. This has been validated by integrating various input modalities such as sketch recognition, finger tracking, stylus input and speech recognition.

Unfortunately, the internal implementation of composition boxes can become highly complex. Existing data-level fusion frameworks do not provide adequate programming APIs to implement these composition boxes and therefore developers need to rely on the host language to implement a complex fusion process. Firstly, such a composition box relies on a single event callback method in the host language. This callback method is reactive since it is called when new information is available. However, the development complexity originates from the state management where developers need to keep track of previous events from single or multiple sources. Secondly, clearing older events is another task developers need to perform as few approaches provide an automatically maintained list of historical data. Finally, resetting current states during the cleanup task might lead to false negatives.



Figure 3.10: A meaning frame which requires the ‘play’ command via speech and RFID using HephaisTK

Another challenge when implementing such composition boxes is dealing with *out-of-order event input*. Assume one needs to detect the pattern sequence ‘ A, B, C ’. After event A and event B have been received it is expecting the event C . However, due to external causes, such as unreliable network connectivity or failing sensor nodes [105], event A_2 arrives before C . In this case many implementations will reset the state back to the beginning. Therefore it will ignore event C as it is not part of the correct sequence, even if information provided by event C indicates that it happened before event A_2 . The cost of manually maintaining the potential combinations of fusion candidates introduces a lot of *accidental* complexity for the developer. Additionally, they need to be implemented in an incremental state in order to deliver real-time efficiency. Additionally, when taking into consideration that some patterns are not sequential, for instance when using a complex *temporal specification*, or when streams need to be *synchronised*, this challenge can clearly benefit from architectural abstractions to reduce the accidental implementation complexity.

Squidy In a similar way, Squidy [97] provides nodes with sources and sinks which can be connected. In contrast to OpenInterface, which allows the integration of external components, Squidy is limited to a predefined set of nodes. To communicate the processed information to the outside world the TUIO protocol [83] is used. TUIO relies on the Open Sound Control (OSC) [163] which in its turn is based UDP. UDP-based communication implies that messages might get lost and the protocol needs to accommodate this issue. TUIO is an efficient way to transmit continuous data such as multi-touch, tangible and accelerometer data.

The TUIO protocol is defined such that the difference between the current and last received value can be calculated. However it should not be used to transmit single events such as speech utterances or actions the application should perform. The architecture also limits the ability to support concurrent interaction. Whenever an additional sensor input is required, the developer has to replicate the entire pipe, causing code duplication. Without this separation, input from two devices would sink into a single node that would mix the content and confuse the fusion process.

Finally, most data stream-oriented architectures cannot fuse across levels. For instance, it is difficult to use results from feature-level or decision-level fusion, such as *user identification*, to influence data-level fusion. When fusion across levels is orchestrated using pipes, it means a further increase in complexity inside composition boxes, as the developer need to process input events in combination with higher level events through the same event handler. Unfortunately these streams are not *synchronised*, which means introducing manual delay operations if necessary. In general, data-stream approaches do not cope well with high-level data and decision making. In the next section we analyse a second strand of architectures, namely semantic inferencing architectures, which are more suitable to process high-level data and dialogue management.

3.4.2 Semantic Inferencing Architectures

Semantic inferencing architectures focus on aggregating a limited number of events by means of high-level programming abstractions. As shown in Section 3.1.2, multiple approaches have been explored in the literature, such as meaning frames, unification-based fusion, finite-state machines and symbolic/statistical fusion. In this section we discuss the overall architecture of these approaches.

A Central Hub Architecture A prime example of a decision-level fusion architecture is described by Dumas et al. [39]. As illustrated in Figure 3.11, the architecture is built around a centralised component which receives data from many input sources, including lower-level recognition engines. It is assumed that these lower-level recognisers deliver high-level, mostly unambiguous information, which can be directly used to make informed decisions at the application side. This assumption is not always valid, hence problems arise when conditions are partially satisfied or state transitions have to be undone. Furthermore, existing central hub

architectures do not provide access to historical data as the structures (such as meaning frames or state machines) are only interested in the latest data. With an ATN architecture [102], tokens are the result of a partial fusion and subsequent nodes need to reason over their own constraints.

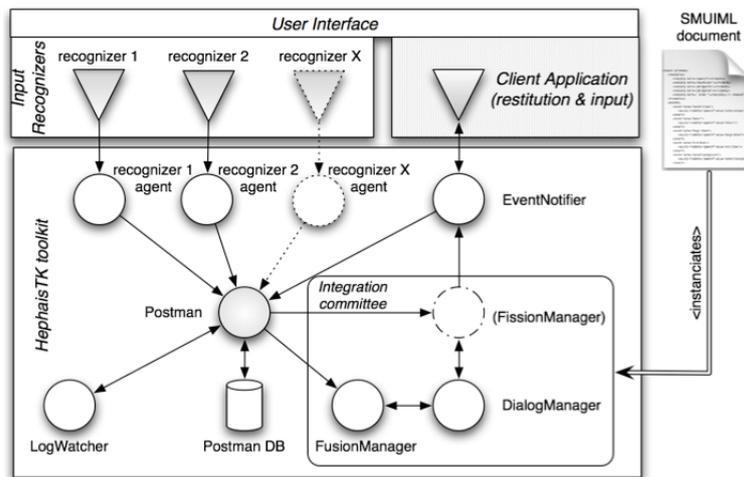


Figure 3.11: HephaisTK architecture

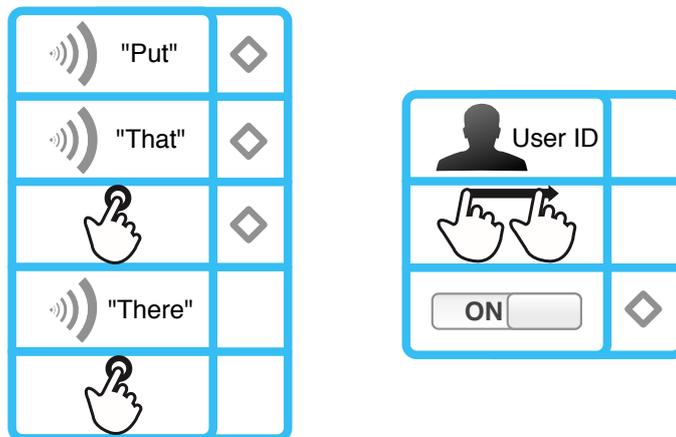


Figure 3.12: A semantic inferencing structure

PATE [129] offers a global working memory structure with a few built-in temporal operators that allow more complex dialogue decisions. However the developer is entirely responsible for clearing event data. Furthermore, each event requires a complete reevaluation of the goal and its conditions. Without an incremental evaluation mechanism, the system is not designed to cope with a high load of event information.

Existing central hub architectures assume the existence of high-level events, delivered by data- and feature-level fusion processes. Unfortunately, for continuous interaction such as pointing, it is hard to control the

frequency to make it compatible with semantic inferencing architectures. The continuous pointing in Bolt's *put that there* scenario [11] is a simple example to illustrate this issue. The refresh rate of the pointing gesture is declared at the data-level processing. A high refresh rate introduces the problem of occupied slots in meaning frames, while a low refresh rate can lead to skipped decision-level integration since they are invalidated by the temporal constraints. Therefore, ad hoc solutions are required to deal with this issue. However, for more complex scenarios such as a multi-user environment, current semantic inferencing architectures do not scale well. Figure 3.12 illustrates this problem by showing a partial match (i.e. occupied slots) for a number of constraints (i.e. `put`, `that`, `point` and a `toggle`).

Service-oriented Architectures Other approaches such as DynaMo [5] rely on a service-oriented architecture to help developers with modularisation, composition and self-management. The latter feature forms the core contribution of DynaMo as it tries to offer self-repair abstractions when services (for example due to network communication issues) go down. In this case, the 'autonomic manager' as depicted in Figure 3.13 will actively search for a compatible replacement or wait for the service to come back. For instance, when detecting pointing gestures, an unstable bluetooth connection of the Wiimote can be replaced with a less precise RGB-D camera. However, the framework does not provide any decision-level fusion abstractions. Furthermore, existing work shows that centralised information storage is required to perform informed decision-level fusion.

In most frameworks such as HephaisTK, Quickset and Match [77], application information can be used to enhance the fusion process. In HephaisTK's architecture, which is shown in Figure 3.11, this is enabled by the `EventNotifier` link. However no programming abstractions are provided to enable a two-way synchronisation between the fusion engine and the application state. A deeper *application symbiosis* is still subject to research. Recent work on the plasticity of multimodal applications, such as Sottet et al. [146] and Stanciulescu et al. [150], provide abstractions to ease the development of user interfaces. However, these approaches focus more on multimodal fission and less on the fusion process. In general, we draw similar conclusions as stated by Lalanne et al. [100], namely that "(...) *the dynamic adaptation of fusion engines to usage patterns and preferences should be further studied.*". This requires multimodal architectures which support fusion across the three fusion levels.

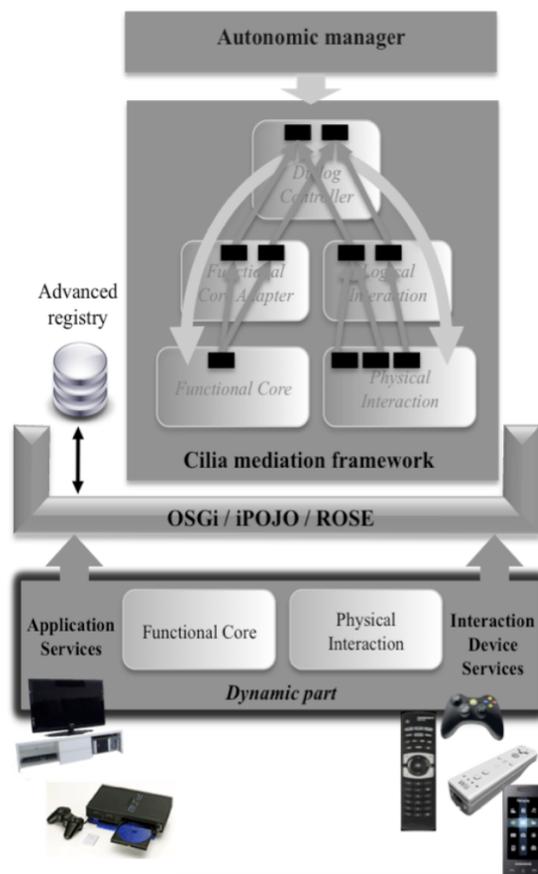


Figure 3.13: Service-oriented architecture of DynaMo

3.5 Conclusion

On the one hand, frameworks positioned at the data level, such as OpenInterface or Squidy rely on a linear chaining of composition boxes. Although these composition boxes encapsulate the implementation complexity, the internal implementation of such a box is far from trivial. As illustrated in Section 3.2, data-stream approaches score poorly on language features and multimodal disambiguation criteria.

On the other hand, semantic approaches provide higher-level language features but lack adequate multimodal processing capabilities. Semantic inferencing approaches rely on late fusion, where all high level information is assumed to be gathered and correlated. Despite the introduced robustness, typical decision-level frameworks cannot recover from the loss of information that might occur at lower levels. A secondary limitation of these frameworks is the lack of support for overlapping matches, which is present when using continuous sensor input, complex definitions or multiple users.

We argue that in order to scale high-level languages to fuse high-frequency data-level information, an adequate language in combination with a well-founded architecture is required. Furthermore, the symbiosis of application-level information and the fusion process is of high importance when developing tools for multimodal applications [102] (Section 3.3.4). Finally, we observe that visual authoring tools are either device specific without multimodal integration capabilities (Section 3.3.2) or allow developers to define a multimodal pipeline but lack device specific authoring abstractions (Section 3.3.4).

The indicative classification of related work in Section 3.2 shows that many important aspects such as segmentation, negation, event expiration, grouping, scalability in terms of performance and others are still challenging to implement in existing multimodal frameworks. In the remainder of this dissertation we argue that these concerns can be tackled by focusing on two main domains: software language engineering abstractions and a performant architecture to process events across data, feature and decision levels.

4

Midas: A Programming Language for Multimodal Interaction

In this chapter we present Midas, a high-level programming language designed to express multimodal fusion based on well-founded software engineering abstractions. We start by providing a formal grammar of a core version of the Midas language, we introduce some terminology and we gradually refine a running example. We then discuss our generic multimodal language features and continue with support for data-, feature- and decision-level fusion processes. We demonstrate data-level fusion constructs that are particularly interesting for processing low-level events, including the discovery of patterns based on spatio-temporal relations. Then we demonstrate feature-level fusion abstractions such as the alignment of input streams, dynamic service instantiation for feature extraction or verification, and asynchronous conditions. Afterwards, we elaborate on language abstractions for decision level fusion such as shadow facts, activation policies. We conclude with a discussion on abstractions for commonly observed multimodal patterns.

4.1 A Declarative Language

As described in the previous chapter, various frameworks and architectural abstractions have been proposed. Only a few of these provide adequate software engineering abstractions and most of them lack composition and negation. Furthermore, existing high-level languages, such as GeForMT [86] and SMUIML [40], offer only a narrow vocabulary for touch or dialogue management. In contrast to approaches providing a simpler, but highly constrained domain-specific programming language, we do provide a high-level declarative programming language capable of processing various input streams for data-, feature- and decision-level fusion. Emphasis is put on providing software engineering abstractions such as modularisation, composition, customisation and the ability to express complex patterns. Developers should be able to describe their requirements declaratively and let the engine perform the state management and computational complexity. Furthermore we encourage the development of novel primitives such that the developers can share code and extend the scope of this work.

4.1.1 Formal Grammar of Midas

A formal grammar of the basic Midas programming language is presented in Figure 4.1. Syntactic text is represented in bold or by symbols. A pipe (`|`) denotes choice while parentheses indicate optional usage. Overlined entities implicate a repetition of zero, one or more. We provide a minimal Midas specification to reduce the complexity. A complete ANTLR [126] grammar can be found in Appendix (Appendix D), as well as a transcript of Figure 4.1 (Appendix B). Subsequent sections will gradually introduce each syntactic entity and highlight their benefits to the criteria defined in Section 2.3.

4.2 Interpreting Midas

A Midas program is a declarative description of multimodal interaction patterns. Therefore, the interpretation of the program is different from mainstream programming languages where each line corresponds to a set of processor instructions.

The computation starts when input events are captured by various sensors. Each input event is represented as a *fact* at a particular point in

$mp \in \text{Program}$	$::= \overline{t \mid m \mid r \mid a \mid f \mid x}$	PROGRAMS
$t \in \text{Template}$	$::= \mathbf{template} \ t_{id}$ $\quad \overline{i \mid a \mid f}$	TEMPLATES
$m \in \text{Module}$	$::= \mathbf{module} \ m_{id}$ $\quad \overline{i \mid a \mid f}$	MODULES
$r \in \text{Rule}$	$::= \mathbf{rule} \ r_{id} \ \overline{c \mid x}$	RULES
$a \in \text{Attempt}$	$::= \mathbf{attempt} \ a_{id} \ (\overline{l_{id}}) \ \overline{c} \ (cf)$	ATTEMPTS
$f \in \text{Function}$	$::= \mathbf{function} \ f_{id} \ (\overline{l_{id}}) \ \overline{e}$	FUNCTIONS
$c \in \text{Condition}$	$::= ce \mid te \mid b \mid sf$	CONDITIONS
$x \in \text{Modifier}$	$::= \mathbf{assert} \ t_{id} \ \{ \overline{s_{id} \Rightarrow e} \}$ $\quad \mid \mathbf{modify} \ l_{id} \ \{ \overline{s_{id} \Rightarrow e} \}$ $\quad \mid \mathbf{retract} \ \overline{l_{id}}$ $\quad \mid \mathbf{call} \ (t_{id}.)f_{id} \ \overline{e}$	MODIFIERS
$ce \in \text{Cond. Element}$	$::= t_{id} \ \{ \overline{cv} \}$	CES
$te \in \text{Test}$	$::= e \ < \mid \leq \mid = \mid \neq \mid \geq \mid > \ e$ $\quad \mid (l_{id} \leftarrow) a_{id} \ \overline{e}$ $\quad \mid (t_{id} \leftarrow) a_{id} \ \overline{e}$	TESTS
$b \in \text{Bind}$	$::= l_{id} = (ce \mid a_{id} \ \overline{e} \mid e \mid ar)$	BINDINGS
$sf \in \text{Special Form}$	$::= \mathbf{no} \ \{ \overline{c} \}$ $\quad \mid \mathbf{async} \ (t_{id} \leftarrow) a_{id} \ \overline{e}$ $\quad \mid \mathbf{wait} \ l_{id} \ nr$	SPECIALFORMS
$e \in \text{Expression}$	$::= (t_{id}.)f_{id} \ \overline{e} \mid l_{id}.s_{id}$ $\quad \mid v \mid b \mid e \ \delta \ e$	EXPRESSIONS
$cf \in \text{Computed Fact}$	$::= \mathbf{return} \ \{ \overline{s_{id} \Rightarrow e} \} \ \mathbf{with} \ \overline{m_{id}}$	CFS
$cv \in \text{Constr. Value}$	$::= s_{id} \ == \mid \neq \mid v$	CVS
$i \in \text{Include}$	$::= \mathbf{include} \ m_{id}$	INCLUDES
$v \in \text{Value}$	$::= nr \mid string \mid symbol \mid l_{id} \mid nil$	VALUES
$ar \in \text{Array}$	$::= [\overline{l_{id}} \mid \overline{v}]$	ARRAYS
$\gamma \in \text{Type}$	$::= int \mid float \mid string$	TYPES
$\delta \in \text{PrimF}$	$::= + \mid - \mid * \mid / \mid mod \mid \dots$	PRIMITIVES

$l_{id} \in \text{VariableName}$
 $r_{id} \in \text{RuleName}$
 $s_{id} \in \text{SlotName}$
 $t_{id}, m_{id} \in \text{Template- or ModuleName (Capitalised)}$
 $a_{id}, f_{id} \in \text{Attempt- or FunctionName}$

Figure 4.1: Basic grammar of Midas

time. All facts are perpetually gathered in a database, called the *fact base*. The goal of our approach is to discover a combination of facts that adheres to a description of multimodal interaction, known as a *rule*, written by the developer. Rules enable reactive fusion capabilities when they consist of a number of *conditions* that should be met. These conditions express spatial or temporal relations between facts and vary for each type of input sensor. However, the reactivity of these conditions is important because this allows the engine to discover patterns from new events as they enter the system. The combination of this event-driven behaviour in combination with high-level language abstractions form the basis of the Midas programming language.

A Midas program consists of templates, modules, rules, attempts and functions. The two former entities (i.e. templates and modules) structure data and code, while the three latter (i.e. rules, attempts and functions) involve calculation.

4.2.1 Templates, Modules, Facts and Events

A *template* represents a blueprint of which facts are instances of. It is similar to a class definition in object-oriented languages. Templates have a name and provide an extensible structure based on attribute-value pairs, namely *slots*, for representing input data. For example, Listing 4.1 describes a simple Touch2D template to represent 2D touch events, containing the `time`, `x` and `y` slots. When an event from an input sensor is delivered to the system it is represented as an instantiated template. These instantiated templates are called *facts* and are used throughout the entire system to communicate information.

Listing 4.1: Simple template description of 2D touch events

```

1 template Touch2D
2   finger, x, y, time                                # Define a finger, x, y and time slot
3 end

```

The slot `time` is a default slot and is therefore always part of a template, even if the developer did not specify it. Timing information can be provided by the input source or is automatically set at the time it was asserted to the fact base.

Modules are similar to templates but cannot be instantiated. They are used to group reusable functionality such as attempts, functions and slot definitions (see later). Modules can be included inside other modules and templates and act similar to mixins in object-oriented

languages such as Ruby¹. Mixins provide functionality similar to multiple inheritance by mixing specialised functionality represented by abstract classes (i.e. modules) into classes. Lines 1 to 12 in Listing 4.2 display two reusable modules, namely `Time` and `Space2D`, which are embedded in a more elaborate version of a `Touch2D` template² on line 13.

Templates and modules hold attempts, functions and slots as shown in lines 2 and 3. This allows developers to invoke functionality specialised for a particular fact. Additionally, the `self` keyword, prefixed to attempts or functions definitions as shown at line 9, provides the functionality on a module or template level. This is similar to static class methods in Java. For instance the function `Math.sqrt` (line 10) an invocation at the module level while `f.time` (line 4) is an instance-based invocation through a bound fact (`f`).

Listing 4.2: Template description of 2D touch events

```

1 module Time
2   time                                     # Define a time slot
3   attempt beforeF(f, eps = 0)           # this should happen before fact f
4     time + eps < f.time                 # Access time slot of fact f
5   end
6 end
7 module Space2D
8   x, y                                    # Define x and y slots
9   function self.euclidean_distance(x1, y1, x2, y2) # Calculate Euclidean distance
10    Math.sqrt ((x2-x1)**2) + ((y2-y1)**2) # Call sqrt function of the Math module
11  end                                     # Returns an integer value
12 end
13 template Touch2D
14  include Time, Space2D                   # Include Time and Space2D functionality
15  finger, m, vx, vy                       # Define motion acceleration and x, y speed
16 end

```

4.2.2 Rules with Conditional Elements, Tests, Attempts and Functions

Each rule contains a number of conditions. One of type of conditions are called conditional elements (CE) and express the need for content. A conditional element abides by the definition of template and introduces computation in a rule. For instance the rule `matchTouch` in Listing 4.3 matches any `Touch2D` fact instance and binds it to the variable `t`. This computation is data driven and as soon as a new fact enters the fact base,

¹Inheritance and Mixins in Ruby: http://ruby-doc.com/docs/ProgrammingRuby/html/tut_classes.html#UA

²Based on the TUIO v1.1 Protocol Specification: <http://www.tuio.org/?tuio11>

the engine will look for rules whose conditional elements correspond to its type. When the fact type and the conditional element match, the condition is satisfied and the rule is *activated*. When multiple conditional elements are listed and multiple matching facts are present in the fact base, an extensive cross-product search is performed. It should be noted that this process is highly optimised via a Rete network (Chapter 5) and takes care of intermediate results for data-driven computation.

Listing 4.3: Conditional elements

```

1 rule matchTouch
2   t = Touch2D           # Match any touch fact
3 end

```

Conditional elements can be accompanied by inline constraint values (CV). CVs provide a shorthand syntax for filtering facts that match conditional elements. Constraint values test the (in)equality (`==` or `!=`) of a slot value with one or more values. An example is shown in Listing 4.6 (line 3), where the finger identity of a second `Touch2D` conditional element is unified with the finger identity of the first one. The equality of slot values between two facts without the use of explicit values, such as an integer, string or symbol value, is called unification. This is particularly useful for grouping events. However, CVs are limited in expressiveness and cannot describe mathematical conditions such as smaller or greater than.

While conditional elements introduce computation by matching to new facts, tests filter facts based on their properties (i.e. slot values).

Tests

Tests filter matched conditional elements and consist of mathematical expressions (`<``|``≤``|``==``|``!=``|``≥``|``>`) to express conditions on the slot values. For example, the test `t1.x > 5` filters all facts whereby the `x` slot value of `t1` is less than or equal to 5 (Line 3 of Listing 4.4). Tests also describe relations, such as `t.x > t.y`, which express that the `x` value should be greater than the `y` value. Tests can also express relations between matched condition elements. Such a test looks like `t2.x == t1.x` (line 5), which specifies a unification, meaning that two facts should exist with the same `x` value. Multiple tests listed in a rule are chained by an implicit `and`. A combination of attempts and tests can be bundled by attempts to provide abstraction.

Listing 4.4: Tests

```

1 rule matchTouchLT5
2   t1 = Touch2D           # Match any touch fact (and ...)
3   t1.x > 5              # with an x slot value greater than 5
4   t2 = Touch2D           # Match another touch fact
5   t2.x == t1.x          # with an x slot value equal to t1.x
6 end

```

Attempts

An attempt encapsulates a complex condition by bundling conditional elements and tests in a named abstraction. They *attempt* to discover facts to make their internal conditions true. Attempts allow developers to abstract code whenever multiple rules share similar conditions. Examples are temporal conditions such as Allen’s interval algebra [1] (line 3 of Listing 4.2) and spatial relations. A common scenario is that two rules only differ at a few spatio-temporal values. Code reuse and abstraction is obtained by moving shared conditions into an attempt. Furthermore, attempts provide parameterisation and therefore enable customisation. A dedicated syntactic symbol, namely a left arrow (\leftarrow), is used to explicitly separate them from function calls. This is illustrated at line 4 of Listing 4.5 which expresses that the time of $t1$ is less than the time of $t2$.

The Midas syntax is designed such that attempts can only describe conditions. Therefore they need to be embedded within rules to become useful.

Listing 4.5: Attempts

```

1 rule matchTouchBefore
2   t1 = Touch2D           # Match any touch fact
3   t2 = Touch2D           # Match another touch fact
4   t1 $\leftarrow$ beforeF t2 # and attempt to satisfy the beforeF attempt
5 end

```

Functions

Midas provides a number of primitive functions such as $+$, $-$, $*$, $/$. Besides these primitives, developers can implement their own functions. However, functions are purely functional and cannot express conditions or modifications. They are typically used for mathematical operations such as the translation, rotation and scaling. A number of external functions can also be invoked, albeit in an unsafe manner as we cannot verify if they

are purely functional. Impure function invocations should be preceded by the `call` keyword and are tagged as modifiers.

Through the use of the `self.` prefix, developers can easily group functions inside a module. For example, the `Space2D` module embeds the `euclidean_distance` function (defined by Listing 4.2, line 9 and used in Listing 4.6, line 5). The `self←` construct is available for attempts as well, and behaves similar to a static class method invocation in Java.

Listing 4.6: Detect movement

```

1 rule touchMovement
2   t1 = Touch2D                               # Match any touch fact
3   t2 = Touch2D { finger == t1.finger }       # Match a second touch from the same finger
4   t1←beforeF t2                               # Touch t1 should happen before t2
5   Space2D.euclidean_distance(t1.x, t1.y,     # and their Euclidean distance should be
6                               t2.x, t2.y) > 5.px # greater than 5.px
7   assert Move { finger ⇒ t1.finger }         # Movement detected
8 end

```

4.2.3 Rules with Modifiers

Midas provides four modify statements which can be executed whenever a pattern is discovered. Modifiers are functions that change the state of the fact base. They create (`assert`), change (`modify`) and remove (`retract`) data or invoke external state changing function calls (`call`). Modify statements are executed as soon as all conditions listed before them are met. A common scenario is that a number of primitive conditions are declared and when they are matched, a new higher-level fact is created. This *extends* the knowledge of the system and abstracts from low-level details. For instance Listing 4.6 matches a number of primitive `Touch2D` elements with a spatio-temporal relation between them and creates a high-level fact `Move` (line 7 of Listing 4.6).

In contrast to existing rule languages, Midas allows multiple alternations between conditions (traditionally known as *left-hand side operators*) and modifiers (*right-hand side operators*). By supporting the mix of conditions and modifiers inside a single rule, more advanced rules can be created. This is particularly interesting when subsequent modifiers require a few additional constraints in between. Existing approaches cannot deal with this scenario as the left-hand side can only be followed by a single right-hand side. However, based on our experiments, the alternation between conditions and modifiers was prevalent in the context of multimodal fusion due to the need to provide intermediate feedback while conditions are gradually matched.

4.2.4 A Midas Implementation of the Hold-and-Rotate Gesture

To motivate our language abstractions, we implement the *hold-and-rotate* example described in Chapter 3. The following, recapitulated textual description has been annotated with roman numbers to link constraints to the code in Listing 4.7: *Find three events^I of three different fingers^{II} touching the surface in any order^{III} at about the same time^{IV}. All these fingers should be located near each other (in a bounding circle)^V. Two fingers should be approximately vertically aligned^{VI} and the third finger should be on the left of the others^{VII}. Find follow-up events of the two vertically aligned fingers such that it corresponds to a movement to the right (i.e. with a minimum and maximum x and y displacement)^{VIII}. The third finger should remain approximately stationary, without lifting up^{IX} during the time interval defined by the movement of the two other fingers^X.*

The textual gesture description neatly corresponds to the declarative implementation in Listing 4.7. Lines 2 to 4 match a combination of these facts in the fact base, expressing the first concern^I in the description. These three matched facts should originate from different fingers (line 5)^{II}. Line 6 is a spatial attempt requiring that the stationary finger is located near one of the moving fingers^V. Furthermore, the two moving fingers should be vertically aligned^{VI} within a boundary. This implies that the second moving finger is also near the stationary finger^V. The stationary finger is ensured to be left of the moving finger^{VII} on line 8. Finally, the temporal constraints on lines 9 to 11 express that the movement of both fingers happens at the same time^{IV} while the third finger remains stationary^X. It should be noted that Midas performs its recognition regardless of the order of conditional elements in a rule^{III}. All temporal constraints are therefore explicitly specified by the developer, without expressiveness limitations.

The remaining concerns^{VIII,IX} are handled by abstraction through high-level facts. This happens when existing rules hide processing complex conditions by generating higher-level facts which can be used by other rules. This reduces complexity as rules become more concise and modular.

Listing 4.7: Hold-and-rotate gesture implementation in Midas

```

1 rule holdAndRotate
2   h = Hold                                # Match a stationary finger I
3   m1 = SwipeRight                         # Match a moving finger to the right I,VIII,IX
4   m2 = SwipeRight                         # Another moving finger to the right I,VIII,IX
5   diffs [h.finger, m1.finger, m2.finger]  # Originating from different fingers II
6   h←near m1.x_begin, m1.y_begin, 50.px    # The hold and m1 finger should be close V
7   m1←align_beginF m2, 20.px, 5.px        # Vert. align and ensure m2 is close to m1 VI
8   h.x < m1.x_begin                        # The hold is left of the m1 VII
9   m1←duringF h                             # M1 happens during the hold IV,X
10  m2←duringF h                             # M2 happens during the hold IV,X
11  m1←startsF m2, 0.5.s                    # Equalise begin time of m1 and m2 III,IV
12 end

```

As a reflection with respect to related work, the *hold-and-rotate* Midas implementation is not only quite concise, but also reflects all gesture requirements. Therefore, Midas provides the ability to specify requirements at a high level of detail based on user-defined spatial and temporal conditions. In the following section we discuss more general multimodal language features of Midas and demonstrate an even more refined implementation of the *hold-and-rotate* example.

4.3 Multimodal Language Features

In this section we discuss the properties of Midas with respect to the language features criteria presented in Section 2.3. These language features form the basis for well-structured declarative programming that supports multimodal fusion in a highly efficient way.

4.3.1 Modularisation and Abstraction

Midas fosters a modular approach to separate multimodal specifications in multiple rules. By modularising definitions we can reduce the effort to add extra interactions by not requiring a deep knowledge about previously implemented rules. Each rule r can focus on its core functional concerns and assert a new higher-level fact that improves understanding of the sensor input. Then another rule r' can abstract over the results from the previous rule r and deal with other concerns. By modularising and composing rules in this incremental manner, more complex types of interaction can be built more easily. Midas encourages developers to compose multimodal interaction rules instead of building ad-hoc solutions.

Abstraction Through High-level Facts

A first approach for abstraction is the use of high-level facts. In Midas, each rule can assert a new fact which can be used by software components that have access to the fact base. Rules can require the presence or absence of a particular high-level fact without being concerned with when or how the fact was created. Moreover, low-level details of such a high-level fact are encapsulated by a number of slot values.

High-level fact abstraction as illustrated in Listing 4.7 works well for small gesture vocabularies. In this case, each rule reacts to its specified conditions and asserts a higher-level fact whenever a match is found. However when the gesture vocabulary gets extended and reused, a lot of aspects of lower-level rules can be changed. For example, when a new L gesture wants to reuse the `swipeRight` gesture with a different spatial relation. Modifying this spatial relation obviously has an impact on other gestures that rely on the same facts, such as the `hold-and-rotate` gesture. Existing approaches do not cope well with this problem and developers typically would need to duplicate rules like the `swipeRight` definition in order to customise it.

Code duplication should be avoided as every extra line of code incurs maintenance costs. It is also error prone and introduces redundant computation. Furthermore, these duplicate rule definitions require duplicate templates in order to separate their assertion type and reuse in other rules. This leads to poor software engineering as developers need to know which type of fact to specify in their rules (i.e. `SwipeRight`, `SwipeRight2` or `SwipeRightLong`).

Conversely, when multiple rules assert the same fact type, developers face the problem of *data entanglement* (also known as the specialisation versus generalisation problem for data types). Data entanglement arises when multiple rules assert the same fact type, but use custom slots that are only useful for some of the consuming gestures. For instance the L gesture requires the finger identifier of the original Touch2D facts. Other gestures require velocities or user identity, which might not be known for all rules that assert this type of fact. The adaptation of existing template definitions causes *tight coupling* between multiple rules. This is harmful for reusability. Therefore, in Midas another form of abstraction is available in the form of *attempts*.

Abstraction Through Attempts and Computed Facts

Attempts are a novel type of abstraction that solves specialisation and data entanglement concerns for multimodal languages. Existing abstraction methods in multimodal interaction languages rely on the encapsulation power of data types. However, multiple producers and consumers of such data types cannot always be properly aligned. Firstly, some information might be missing for a new consumer rule (i.e. a rule that is using the fact type as a conditional element) which implies that all existing producer rules (i.e. rules asserting the fact type) need to be updated to generate this information. Secondly, the template needs to be extended with the additional slot as well, which might be too consumer-specific and not reflect the original intention of the template definition itself.

Through attempts, similar conditions can easily be shared amongst rules. Furthermore, a rule can reuse multiple attempts and express relations between them. Attempts can easily be customised through the use of a parameter list. Attempts also have the ability to provide a return value in the form of a *computed fact*. A computed fact is an untyped fact (i.e. there is no corresponding template) containing slot-value pairs. They replace the need for excessive template definitions, while still providing allowing developers to exchange details of the attempt to the outer rule. Listing 4.8 returns such a computed fact on lines 10 to 12. Line 14 requires the `piezoTap` attempt and binds the result (i.e. a computed fact) to the variable `tap`. Consecutively, line 15 uses the two slot values of the computed fact, namely `value` and `time` to display text on the screen³.

Listing 4.8: Tap attempt for piezoelectric sensors

```

1 attempt piezoTap(max_time, min_power)           # Attempt to
2   v1 = Vibration                               # Match a vibration event
3   v2 = Vibration { sensor == v1.sensor }       # Match another event from the same sensor
4   v1←meetsF v2, max_time                       # v2 within time interval relative to v1
5   v1.value + min_power < v2.value             # with at least the ratiometric difference
6   v3 = Vibration { sensor == v1.sensor }       # Match another event from the same sensor
7   v2←meetsF v3, max_time                       # v3 within time interval relative to v2
8   v2.value - min_power > v3.value             # with at least the ratiometric difference
9   return { time ⇒ v2.time,                    # Return the relevant information
10    value ⇒ v2.value }
11 end
12
13 rule shortToughPiezoTap
14   tap = piezoTap 500.ms, 550                   # At least a ratiometric power level of 550
15   display "Tough tap via piezoelectric sensor: ", tap.value, " time: ", tap.time
16 end

```

³The construct `display` is a built in modifier to output text to the standard output.

Computed facts can only be used within the scope of the rule and cannot be serialised or asserted. Its function is to exchange detailed information from inside the `swipeRight` attempt to the outer scope. It retains the unification property, for instance when the outer rule would define `m1.x_end == 5.px` it will be mapped to a `Touch2D` conditional element inside the `swipeRight` attempt. Alphanumeric values and fact matches are also supported as return values.

Because computed facts are untyped, they do not provide instance-based attempts and functions. This requires developers to use module or template level invocations (i.e. `Time<-beforeF tap1, tap2`). This is error prone for two reasons: (1) developers can easily confuse `Time<-beforeF` with `TimeInterval<-beforeF` and (2) the computed facts can be updated from a single timestamp to a time interval, thus the call to `Time<-beforeF` can become incorrect when updating code. Therefore, in order to allow a safer, instance-based invocation (i.e. `tap1<-beforeF tap2`), computed facts can be extended with modules. The return value of Listing 4.9 provides instance-based invocation access for the `Time` and `TimeInterval` modules. This enables a dynamic composition when using multiple attempts while still providing compile-time guarantees.

Listing 4.9 generates a compile-time error message because line 17 invokes the `meetsF` attempt which tries to access a local `time_end` and `f.time_begin` value. Unfortunately, these two slot values are not defined by the computed fact (line 10). Thus, computed facts rely on an explicit list of provided slot values which enables this type of valuable compile-time feedback. Computed facts can also not be asserted to the fact base as they lack an actual type (i.e. template definition).

Midas provides attempts to solve the data entanglement problems of existing abstraction methods while allowing for easier customisation and compile-time feedback.

Listing 4.9: Extending computed facts with modules

```

1 module TimeInterval
2   time_begin, time_end
3   attempt meetsF(f, eps = 0)
4     time_end - f.time_begin < eps
5   end
6 end
7
8 attempt piezoTap(max_time, min_power)
9   ...
10 return { time ⇒ v2.time,
11   value ⇒ v2.value } with [Time, TimeInterval] # Extend with Time and TimeInterval
12 end
13
14 rule shortToughPiezoTap
15   tap1 = piezoTap 500.ms, 550
16   tap2 = piezoTap 500.ms, 550
17   tap1←meetsF tap2                                # No slot 'time_end' for #<ComputedFact>
18 end                                                # No slot 'time_begin' for #<ComputedFact>

```

4.3.2 Inheritance as Composition of Modules

As mentioned before, a module is a container for attempts, functions and slots. This allows related functionality to be grouped together and enables the usage of the overloaded identifiers in a scoped manner (such as `Time<-before` and `TimeInterval<-before`). The definition of slots inside modules reduces the number of parameters required to invoke attempts, as attempts can refer to local slots. An example of this is shown in Listing 4.10, where line 2 defines the slot `time` which is referenced on line 4 via traditional static scoping rules. A complete implementation of the attempts and functions used in the examples of this chapter is shown in Appendix E.

Templates can be extended through module composition. As shown in Listing 4.11, the template definitions of `Touch2D`, `Hold` and `SwipeRight` are concise but provide a lot of functionality by including the modules defined in Listing 4.10 and E.1. Similar to the mixin paradigm in object-oriented composition [14], templates and modules can be composed from modules. The order of `include` statements is important as the latest definition is used when attempts or function names collide. However, this causes a warning message during computation.

Listing 4.10: Reusable attempts and functions

```

1 module Time
2   time
3   attempt beforeF(f, eps = 0)
4     time + eps < f.time           # time refers to the slot on line 2
5   end
6   attempt withinF(f, min, max)
7     f.time + min < time           # f.time refers to the time value of
8     f.time + max > time           # the fact f provided in the argument list
9   end
10 end
11 module Space2DInterval
12   include Space2D                 # Extend this module with another module
13   x_begin, x_end
14   y_begin, y_end
15   attempt align_beginF(f, x_diff, y_diff)
16     Space2D ← align x_begin, y_begin, f.x_begin, f.y_begin, x_diff, y_diff
17   end
18 end

```

Listing 4.11: Fact composition

```

1 template Touch2D
2   include Time, Space2D
3   m, vx, vy
4 end
5 template Hold
6   include Space2DInterval
7 end
8 template SwipeRight
9   include Space2DInterval
10 end

```

4.3.3 Customisation and Extensibility

Midas allows developers to easily adapt existing definitions via four principles in order to use them in a different context:

1. The order of a rule's conditions is decoupled from its semantics. Therefore, to describe temporal relations, explicit temporal conditions are used. This has the advantage that constraints can easily be appended at a later point in time, without having to worry about existing code. For example, the condition that a stationary finger is on top of a figure, in the *hold-and-rotate* gesture, can be easily be appended at the end of a rule. This is in contrast to most existing approaches, such as GeForMT [85, 86] and Proton [94, 95], which use an implicit temporal specification based on the order of the

condition. This hampers later customisation as existing code needs to be modified.

2. `Attempts` allows the reuse of complex conditions with different arguments. In this manner rules can be written that use slightly different specification values.
3. Customisation of templates and modules is achieved through the `include` keyword and the ability to extend existing template definitions at multiple places. It is similar to Ruby's class definitions, Midas templates can be extended at multiple places allowing slots to be added without adapting the original definition. In this manner developers can define generic multimodal descriptions in core files and write custom extensions in additional files which are (or are not) included for a given scenario.
4. By providing generic user-defined conditions, advanced abstraction capabilities and a formal grammar, Midas can be used as a representation language for external tooling. For example, graphical authoring tools can export their multimodal definitions to the Midas language. This allows developers to further edit the rules (i.e. outside the authoring tool environment). In a similar way, multimodal mining tools can offer an initial specification based on examples, which can be further refined by the developer.

In Listing 4.12, we define a more precise implementation of the *hold-and-rotate* example. It relies on `attempts` and `computed` facts instead of abstraction through high-level facts and adds additional conditions such as *negation*. The `attempts` `hold` and `movingTouch` abstract the primitive spatial and temporal relations from the more complex `holdAndRotate` rule. Furthermore, they allow easy customisation such that they can be reused for other complex gestures.

Listing 4.12: Precise hold and rotate implementation in Midas

```

1 template Touch2D
2   UP = 3
3 end
4
5 attempt hold(min_time, max_time)           # Attempt to
6   t1 = Touch2D                             # Match a finger event
7   t2 = Touch2D { finger == t1.finger }     # Match another event from the same finger
8   t2←withinF t1, min_time, max_time       # t2 within a time interval relative to t1
9   t1←nearF t2, 3.px                         # t1 and t2 should be close
10  no { nt = Touch2D { finger == t1.finger }
11    nt.state == Touch2D.UP                 # No touch up from the same finger:
12    nt←during t1.time, t2.time            # - during the interval
13  }
14  no { nt = Touch2D { finger == t1.finger } # No fact from the same finger:
15    nt←awayF t1, 10.px                    # - far away from the initial match
16    nt←during t1.time, t2.time            # - during the interval
17  }
18  return { time.begin ⇒ t1.time,         # Return the relevant information
19    time.end ⇒ t2.time,
20    x.begin ⇒ t1.x, y.begin ⇒ t1.y } with [TimeInterval, Space2DInterval]
21 end
22
23 attempt movingTouch(min_x, max_x, min_y, max_y)
24   t1 = Touch2D                             # Match a finger event
25   t2 = Touch2D { finger == t1.finger }     # Match another event from the same finger
26   t1←beforeF t2                             # t1 happens before t1
27   t2.x > t1.x + min_x                       # t2 within a bounding box relative to t1
28   t2.x < t1.x + max_x
29   t2.y > t1.y + min_y
30   t2.y < t1.y + max_y
31  return { time.begin ⇒ t1.time, time.end ⇒ t2.time,
32    x.begin ⇒ t1.x, x.end ⇒ t2.x,
33    y.begin ⇒ t1.y, y.end ⇒ t2.y } with [TimeInterval, Space2DInterval]
34 end
35
36 rule holdAndRotate
37   h = hold 0.5.s, 2.s                       # Hold at least 0.5s
38   m1 = movingTouch 10.px, 20.px, -3.px, 3.px # Move at least 10px to the right
39   m2 = movingTouch 10.px, 20.px, -3.px, 3.px # Move at least 10px to the right
40   h←nearF m1, 5.px                          # The hold and m1 finger should be close
41   m1←align.beginF m2, 20.px, 5.px          # Vert. align and ensure m2 is nearby m1
42   h.x.begin < m1.x.begin                    # The hold is left of the m1
43   h←duringF m1                              # M1 happens during hold
44   h←duringF m2                              # M2 happens during hold
45   m1←startsF m2, 0.5.s                     # Equalise begin time of m1 and m2
46  assert HoldAndRotate { x ⇒ h.x.begin,    # Assert the gesture
47    y ⇒ h.y.begin, diff ⇒ Math.abs(m1.x.end - m1.x.begin) }
48 end

```

4.3.4 Negation

Negation is a language feature to express that certain patterns should not be true in a rule. This means a rule should not activate in a particular context (i.e. do not accidentally detect a swipe right while scrolling a document) or with particular sensor values (i.e. the gesture is invalid when seated). Midas provides a keyword for negation (`no`) that provides a view over the entire fact based to verify that certain patterns should not occur.

Two examples of negation are shown in Listing 4.12, lines 10 to 13 and 14 to 17. The former expresses that there should be no touch up from the same finger during the interval defined by the two matched `Touch2D` facts. The latter expresses that there should be no `Touch2D` fact from the same finger which is far away from the matched coordinates during the matched interval. Note that this implementation is quite different from mainstream imperative programming solutions and allows the declarative processing engine to perform the complex pattern matching without having to manually track intermediate state. Indeed, many combinations are possible and constraining them in imperative code might not always be trivial. For instance, the time relation on line 12 does not consist of a particular value but uses the relative time of potential matching facts. In this example, the negated expression further allows touches of other fingers in the neighbourhood to be match to other gestures without additional coding effort.

Negated patterns can be arbitrary complex but it should be noted that expressing negated conditions on future events is difficult. This is because related facts can expire and future facts are not available to be checked when the rule is partially matching. The problem of missing future events can be mitigated using the `wait` construct to delay the matching process, such that “future” information can be consulted.

4.3.5 Application Symbiosis

In multimodal applications, the varying state of the application may be important to be consulted. In this section we distinguish two main operations to properly interoperate with applications. This requires a *foreign function interface* between Midas and the language the application is written in. On the one hand, when meaningful information for the application is inferred, the knowledge has to be transferred from the fusion engine to the application. On the other hand application context could

provide meaningful information during the fusion process. Therefore the exchange of information is a two-way process. State-of-the-art multimodal language abstractions, such as HephaisTK, offer a two-way information exchange service but are limited to simple event exchange. Events are unfortunately not well suited to offer a deep application integration as they merely trigger a callback function to inform the application of novel information. This callback function is problematic for three reasons: (1) it needs a generic type to support different kinds of events, (2) it is called by a stateless function (i.e. a callback without access to contextual information) that runs in a parallel with the main application thread, and (3) it causes the inversion of control. This inversion of control happens when the procedural code is not driven by the program, but by events coming from outside the application.

Midas provides different strategies to provide the two-way exchange of data. We enlist them from the simplest to the most advanced solution.

1. As provided by most approaches, we offer a primitive callback-based event interface. An application can register for particular facts types by using a topic-based subscribe API. Our Java topic-based publish/subscribe API is illustrated in Listing 4.13 where an event callback is subscribed for the `SwipeRight` fact type (line 1). The `processEvent` event handler in the Java application is invoked every time Midas “detects” a new `SwipeRight` fact.
2. The topic-based publish/subscribe API passes all events into a single event handler. A more fine-grained solution is to use a context-based subscription method, which is inherently provided by rules. For instance, an application can only be notified whenever a particular event was added to the fact base, given that slots obtain a particular value or other complex patterns are satisfied. A context-based subscription method is thus written as a rule, which invokes the `publish` modifier to inform the interested party.
3. It may be difficult to confine all information into a single event which should be understood by the application-level code. Therefore, it might sometimes be appropriate to invoke the application API directly. In Midas, this is done through the `call` construct. The `call` modifier exposes application functions to rules, and allows developers to pass the relevant arguments required by the application function. It thus provides a way to invoke a sequence of functions

to properly set the application state. A downside of this approach is that the compatibility breaks when API functions get updated.

4. The most refined integration between Midas and an application can be obtained via *shadow facts*. Shadow facts are a *symbiotic* abstraction to provide a fully synchronised representation of application objects in the fact base. Modifications to attributes of a shadow facts (via the `modify` construct) will immediately be percolated to the application. Conversely, whenever the application modifies the state of an object, its shadowed fact is updated as well. Shadow facts are powerful because they can be used as conditional elements in a rule. This allows developers to program multimodal descriptions while relying on the dynamic state of the application. For example, one can express that a rule can only be activated if there exist a shadow fact. This, amongst other things, improves performance as results that are bound to be omitted by the application, will not be computed. We elaborate on the use of shadow facts in Section 4.6.1 where they benefit decision-level fusion processes.

Listing 4.13: Topic-based subscribe API

```

1 mudra.subscribeEventListener("SwipeRight", new EventListener() {
2   public void processEvent(Event e) { ... } });

```

4.3.6 Unbound Variables and Unification

It is often not possible to choose precise values for each spatial or temporal condition. In many cases, intervals or multiple values provide a solution to this problem. However, this does not suffice when multiple conditions should be true for a particular value, regardless of which value it actually is. For example, multiple conditions can be matched to events from a single user, regardless of the user's identity.

In Midas, developers use unbound variables and unification to describe values without a concrete instantiation. This makes it possible to bind a particular value at runtime when it conforms to a given interval. For example, Listing 4.14 binds the `new_x` variable to `x + 5.px` of a matched `Touch2D` fact. As seen from the example, these variables can reference runtime slot values of other conditional elements (i.e. `t1.x`). The ability to bind relative values can be used in additional conditions, such that relations to the x value of another matched `Touch2D` can be tested (line 6). Line 5 of this example demonstrates the use of unbound variables to unify

values. As previously shown in Listing 4.12 (lines 7 and 10), the finger identity of the matched `Touch2D` events is ensured to be equal, without requiring an actual value. In a similar way, developers can use this feature to easily support multi-user descriptions without having to manually separate the state of each individual user.

Listing 4.14: Unbound variables and unification in Midas

```

1 rule unboundVariablesAndUnification
2   t1 = Touch2D
3   new_x = t1.x + 5.px           # Bind the result to new_x
4   t2 = Touch2D
5   t2.finger == t1.finger       # Unify the touch identity
6   t2.x > new_x                 # Reference the relative new_x variable
7 end

```

4.3.7 Event Expiration

Input streams continuously provide data that is represented as facts added to the fact base. However, memory is not infinite and facts also need to be removed. In Midas this is done by rules or via a timespan parameter. In rules, the `retract` modifier can be used on any fact that is matched by a conditional element to explicitly remove it from the fact base. This is shown in Listing 4.15 that compares `Touch2D` facts to other `Touch2D` facts and if the timespan between them is larger than 10 seconds the older fact is removed. Without the need for a global clock, this rule effectively creates a sliding time window that removes old events solely when new events enter the system. However, our experiments indicate that manual retraction likely leads to unintentional activations, for instance when negated conditions become true due to retracted knowledge. Furthermore, manual retraction leads to state management, which should be reduced as much as possible in the context of multimodal fusion. Therefore, Midas provides a second event expiration abstraction based on a sliding time window.

Listing 4.15: Retracting expired facts

```

1 rule gcTouch2D
2   t1 = Touch2D
3   t2 = Touch2D
4   t1 ← beforeF t2, 10.s
5   retract t1
6 end

```

The sliding time window method uses a combination of the time and timespan slot of a fact. The Mudra engine, presented in Chapter 5,

tracks these slots of each fact and compares it to new facts that enter the system with the same template (Section 5.2.2). This behaviour is similar to what one can write in a rule, but is more efficient due to the native C implementation and the use of a sorted data structure. A short timespan allows for a fast expiration of many noisy low-level events, while long timespans preserve important events for a long period. A default value for the timespan is specified in the template definition, as shown in Listing 4.16. During the assertion of a fact this can be overwritten.

Listing 4.16: Setting a default timespan value

```
1 template Touch2D
2   timespan 10.s
3 end
```

It is generally assumed that every fact type originates from the same input source (being a hardware device, external service or rule) or at least has a synchronised time, which makes it feasible to retract based on the difference between the time values of the new and older events.

Without relying on a global clock, we can gather events from multiple devices with different time values. Furthermore, it eases debugging and allows benchmarking at maximum performance (i.e. facts can be asserted faster than real-time). When logging input events for testing purposes, time values are persisted and therefore facts can be used as is when loading them into the Mudra engine. Most CEP systems depend on either a global clock or work without sliding time-windows. Drools⁴ supports both cases but requires a switch between a *cloud* (without a notion of time) and *steam* (forced synchronisation based on a clock) mode⁵. However, the dependency on a clock makes benchmarking more difficult and implies that different versions of the rules might be needed (ones that reason time versus timeless facts).

In the next sections we introduce Midas language features for each of the three fusion levels. The subsections in the discussion of these fusion levels are based on the criteria defined in Chapter 2.

4.4 Data-level Fusion

The process of transforming raw data into more meaningful information is characterised by (1) removing the excess of noise to (2) provide feature

⁴Drools: <http://www.drools.org/>

⁵Drools Event Processing Modes <http://docs.jboss.org/drools/release/5.2.0.CR1/drools-fusion-docs/html/ch02.html#d0e1044>

candidates in (3) a real-time manner [41]. This is a challenging problem since a continuous stream of information containing overlapping patterns from multiple users needs to be segmented at a high frequency. When an engine is not capable of processing the information in an efficient manner, the amount of data might quickly result in a memory and processing bottleneck. Usually data-level fusion focuses on a high recall in order to limit information loss. A higher-level fusion process will then focus on reducing the false positive information. An advanced example of data-level fusion is the multi-touch gesture presented earlier in Listing 4.7. In this example, the combination of spatial and temporal functions allows a declarative specification of a complex gesture with a highly reduced complexity for the developer (i.e. state maintenance, segmentation and overlapping matching is taken care of by the engine). In this section we will go into more details about the spatial and temporal specification, the identification and grouping problem and the use of control points used to specify data-level fusion in Midas.

4.4.1 Spatial Specification

Peak thresholding, 2D and 3D gestures, localisation and other properties can be extracted from raw data streams by means of a declarative spatial specification. A spatial specification can be as simple as expressing that a particular value should be equal, higher or lower than a concrete value. For example, depth sensors such as the Leap Motion⁶ enable hand-based human-computer interaction. Using a spatial peak threshold, developers can specify a minimum depth value of the hand to form a virtual interactive surface (i.e. the z coordinate should be greater than 5.cm), as illustrated in Listing 4.17. A slightly more complex spatial specification is based on spatial intervals (i.e. the z coordinate should be between 2 to 10). The most interesting spatial specification offered by Midas is a relative spatial condition. This is used to declare spatial relations between slot values of multiple conditional elements.

Listing 4.17: Spatial threshold

```
1 rule spatialThreshold
2   hand = Hand
3   hand.z < 5.cm
4 end
```

⁶Leap Motion: <https://www.leapmotion.com>

Relative spatial relations were used in the *rotate-and-hold* gesture specification in Listing 4.12. A `swipeRight` rule defined that multiple touch points have an increasing x coordinate value compared to a previous point in time. The first point is special as it has no actual conditions. Midas uses this point to automatically *segment* continuous input streams. In this case, any `Touch2D` fact can serve as a potential first point and only when follow-up facts are found with an increasing x coordinate, the rule will activate.

We note that the function `.cm` in Listing 4.17 represents a conversion to *centimetres*. This function is part of the `Space` module which is included by all literal numbers (i.e. *fixnums* and *floats*) to provide handy access to spatial dimension. Conversions such as `px`, `dam`, `meter`, `dm`, `cm`, `mm` are built-in and can be extended by the programmer. These functions allow developers to use and extend existing code in a standardised manner.

A number of built-in 2D and 3D spatial operators, such as `euclidean_distance`, `euclidean_similarity`, `angle_degrees` and `translated_near`, have been embedded in the engine. Table 4.1 provides an overview of several spatial operations with their definition. The variables `f1`, `f2` refer to facts and are the first and second argument when invoking the operator. The symbol ε_s can be set to an arbitrarily small positive spatial distance. The symbol $\sigma_{x,y}$ represents user-defined x,y coordinates used for translation (i.e. transforming coordinates along the x and y axis).

Operator	Definition
<code>Space2D.distance</code>	$\text{sqrt}((f2.x - f1.x)^2 + (f2.y - f1.y)^2)$
<code>Space2D.similarity</code>	$1/(1 + \text{distance}(f1, f2))$
<code>Space2D.angle_degrees</code>	$\text{acos}(n1x * n2x + n1y * n2y) * 180/\pi$ with $m1 = \text{sqrt}(f1.x * f1.x + f1.y * f1.y)$ $n1x = f1.x * (1.0/m1)$ $n1y = f1.y * (1.0/m1)$ and similar for $m2, n2x$ and $n2y$
<code>Space2D←near</code>	$\varepsilon_s > \text{distance}(f1, f2)$
<code>Space2D←near_left</code>	$\varepsilon_s > (f2.x - f1.x) > 0$
<code>Space2D←near_right</code>	$\varepsilon_s > (f1.x - f2.x) > 0$
<code>Space2D←translated_near</code>	$\text{near}(f1, \text{translate}(f2, \sigma_{x,y}))$

Table 4.1: Embedded spatial functions and attempts for 2D

User-defined spatial functions and attempts are typically required to express the gestural interaction. This approach is different from existing

multimodal and gesture specification languages where the language itself consists of primitive spatial entities tailored to specific sensors (for instance the `North` attribute of Proton for multi-touch sensors). The spatial dimensions of attributes in existing languages such as Proton introduces two problems: (1) it cannot be easily modified as it requires changing the internals of Proton; and (2) it operates on a system-wide level, which means that the minimum and maximum displacement of the attributes are the same for all gestures. This makes it difficult, or even impossible, to distinguish between short and long swipes.

In Midas, developers need to implement these spatial entities themselves. However, their implementation can easily be reused and customised for multiple gestures and scenarios. For example, the implementation of a swipe up gesture only requires a single line (see Listing 4.18) by reusing the `movingTouch` attempt of Listing 4.12. Due to the accessibility to low-level data in combination with user-defined high-level abstractions, Midas allows a fine-grained solution. This is necessary to properly distinguish many gestures, as Chui [25] puts it about the misidentification of a curve as a straight line and vice versa:

For example, if a short J-curve is misidentified as a vertical line, the character is classified as a ‘T’. In other cases, a long straight line is misidentified as a curve. The reason for these errors is that the two parameters T and k of pre-processing routine are optimally set for characters of a particular size. For characters that are smaller, the T and K should really be decreased to yield optimal performance. (...) Since T is too large for small characters, a curve, for example, would be misidentified as a straight line. Misclassification at this stage causes failure later in properly classifying a character. [25]

Listing 4.18: Attempt for north

```
1 attempt north(min_y)
2   north = movingTouch -3.px, 3.px, min_y, min_y * 2
3 end
```

Given the fine-grained spatial specification ability of Midas, fluid gestures such as a pigtail or a left curly brace can be also implemented (Section 6.5). In Section 4.4.6 we discuss a novel technique to express fluid 2D and 3D gestures in declarative programming languages. The spatial specification in Midas is broader than the gesture scope and can also be used to express distances between persons or objects.

4.4.2 Temporal Specification

Lalanne et al. [100] distinguish temporal behaviour at the quantitative and qualitative levels. Quantitative time is used to express a precise time (for instance at 10.00 am) or exact temporal relation (the maximum time between two taps to form a double tap is 300 milliseconds). Qualitative time deals with the order of events, such as precedence, succession or simultaneity.

In Midas these temporal relations can be expressed by means of constraints on the value of the `time` slot. In contrast to many existing multimodal languages which express time by ordering conditions, Midas uses explicit temporal conditions which can be listed at any line within a rule. This is important because it allows the developer to easily refine rules with additional conditions without requiring deep knowledge of the existing code. It also allows developers to group a type of constraints into several lines, such as spatial relations are followed by temporal relations. This makes the code easier to read and understand.

Listing 4.19: Attempt for before

```
1 attempt before(f1, f2, eps = 0)
2   f1.time + eps < f2.time
3 end
```

Midas provides temporal dimension for numbers, similar to the space dimensions. Temporal conversions such as `day(s)`, `h/hour(s)`, `min/minute(s)`, `s/second(s)` and `ms/millisecond(s)` are built-in to enable expressions such as `5.seconds`. Midas also embeds Allen's temporal operators [1], as shown in Table 4.2. The `Time←before` operator is implemented in Listing 4.19.

4.4.3 Spatio-Temporal Specification

Spatial and temporal conditions can be nested arbitrarily in Midas. This enables the development of custom spatio-temporal specifications to distinguishing between slow and fast walking or a short and long Piezo tap (Listing 4.8). Attempts embed arbitrary conditions and therefore encapsulate spatio-temporal features.

4.4.4 User-defined Attempts and Functions

As mentioned before, Midas encourages the development of user-defined attempts and functions. Therefore, instead of restricting the developers

Operator	Definition
Time←equal	$ f1.time - f2.time < \varepsilon_t$
Time←meets	$f1.time - f2.time < -\varepsilon_t$
Time←before	$f1.time + \varepsilon_t < f2.time$
Time←after	$f1.time > f2.time$
Time←within	$f1.time + \varepsilon_{min} < f2.time$ $f1.time + \varepsilon_{max} > f2.time$
Time←contains	$f2.time < f1.time < f3.time$
TimeInterval←equal	$ f1.time_begin - f2.time_begin < \varepsilon_t$ $ f1.time_end - f2.time_end < \varepsilon_t$
TimeInterval←meets	$f1.time_end - f2.time_begin < \varepsilon_t$
TimeInterval←before	$f1.time_end - f2.time_begin$
TimeInterval←overlaps	$f1.time_end < f2.time_begin$
TimeInterval←starts	$ f1.time_begin - f2.time_begin < \varepsilon_t$
TimeInterval←during	$f2.time_begin < f1.time_begin$ $< f1.time_end < f2.time_end$
TimeInterval←finishes	$ f1.time_end - f2.time_end < \varepsilon_t$

Table 4.2: Temporal operators

to a small vocabulary of built-in functions, the language is open for custom functionality. This enables a lot of functionality outside the scope of our examples without having to modify the engine. However, it also implies that reasoning over properties for optimisation is limited. The current strategy is to identify common patterns and then provide a built-in alternative that can be optimised. An example is the use of a `Space2D←translated_near` spatial operator that checks if a 2D point is nearby a second translated 2D point. This operation has been adopted as a general design pattern to declaratively describe complex 2D and 3D gestures. Therefore a built-in variant has been provided to reduce the computation costs. Similarly, Dynamic Time Warping [32] has been adopted in the C codebase rather than the Midas language.

4.4.5 Identification and Grouping

Certain input sensors provide information related to identity. For instance a multi-touch sensor groups a number of activated pixels to detect a single touch event. This touch event is accommodated with a unique number that remains assigned to the movement of that particular finger in subsequent events. In a similar way, depth cameras with body tracking

software identify limbs. For many fusion processes, identity information is crucial. Identity can either be known (e.g. a fingerprint scanner provides a unique label of a person), partially known (most multi-touch sensors provide a numeric finger identification but it is unknown which finger is actually used) or unknown. Even when identity information is completely missing, as in the case of audio sensors, it is possible to distinguish speech input based on the fusion of input from a microphone array. The user identification can even be further enhanced by relying on the audio-inferred spatial location of the user combined with vision-based face recognition that returns the name of the person speaking.

Another example is the combination of touch and video to identify the type of a finger (such as the index finger of the right hand). A key component of identification processes is grouping. As with most multimodal sensor input, information is incomplete and a number of different combinations are possible that need to be resolved in subsequent steps. For instance when mapping the unique touch number to an actual finger, it might be uncertain during the early fusion process whether it should be mapped to the index or middle finger. Additional information such as video frames or additional sensors can aid this decision.

Unfortunately, existing multimodal languages provide very little programming abstractions to deal with identification and grouping. In Midas we provide identity and grouping abstractions on three levels: instantiated filters, unification-based filters and scoping filters.

Instantiated Identity Filtering The simplest form of identity filtering is achieved by providing concrete values for the attributes on a conditional element. For instance RFID tags can be filtered on the identity level. To match a single RFID tag, one can use an inline constraint value, such as `RFID { id == "L9870I1050S2800ELIA" }`. When conditions need to operate on a known group identity, a similar expression can be used, such as `RFID { kind == "UHF-433" }`. This will match all RFID tags from the same kind.

Unification-based Identity Filtering Unification is used when multiple conditional elements should share a common slot value, but whereby the actual slot value is unknown at development time. As argued in Section 4.3.6, unbound variables and unification allows developers to automatically group a number of events. This is in contrast to imperative languages where all potential combinations need to be manually stored.

Listing 4.20 specifies that two RFID facts from the same kind should be matched. This rule triggers for all pairs of RFID facts from the same kind.

Listing 4.20: Unify the RFID type

```

1 rule unifyRFIDType
2   r1 = RFID
3   r2 = RFID { kind == r1.kind }
4 end

```

Scoped Identity Filtering As rules become more complex, ensuring the identity over multiple conditional elements requires a lot of attention. Indeed, the identity relation has to be repeated for each condition element and is easily forgotten. Therefore, Midas provides a scoped identity construct, namely **group**. Listing 4.21 exemplifies a rule that matches users which smile while walking. The **group** construct unifies the **user** attribute of the conditional elements in its scope (i.e. **Walk** and **Smile**). When a conditional element does not provide the slot, the check is omitted (i.e. if an additional conditional element would be specified in this group (between lines 3 and 5), but the conditional element does not provide a **user** slot, the compiler will still validate the rule).

Listing 4.21: Scoped identity grouping

```

1 rule walkAndSmiles
2   group user
3   walk = Walk
4   smile = Smile
5   smile ← duringF walk
6 end

```

4.4.6 Segmentation and Control Points

In this following section we describe the segmentation properties of Midas, together with a *control points* design pattern to segment 2D and 3D gestures.

Midas rules are event driven and activate as soon as a combination of facts matches its description. This is an important strategy to support segmentation (also known as spotting). Our engine provides an advanced way of segmenting fusion candidates from continuous event streams. For example, the tap gesture using piezoelectric sensors described in Section 4.2.2 and Listing 4.8 matches the sequence of low vibration power level, followed by a short peak, followed again by a low vibration value.

Instead of defining a simple peak threshold (i.e. `value > 550`) to segment the stream, we described three conditional elements relative to each other (such as `second.value > first.value + 500`). This accommodates noise from the sensor (i.e. the power level is never 0), noisy environments (for instance when an engine is running in the neighbourhood) and unanticipated events (such as the vibration caused by a spin up from a rotational hard drive). The problem of segmentation is prevalent when processing input streams from various input sources. A detailed analysis of how our engine copes with the complex segmentation patterns is discussed in Section 5.3.4.

Control Point-based Gesture Spotting

During experiments for segmenting raw data of 2D and 3D input events, we identified a novel design pattern called *control points*. Control points are based on the relative spatial location of multiple facts compared to a first, underspecified, fact. This implies that the first conditional element is unconditioned to particular spatial requirements. Subsequent conditional elements then define a relative spatial relation to this initial conditional element. This spatial relation is typically represented as a translation of an ellipse (2D) or ellipsoid (3D), however many variations are possible.

To define control points, a single, well-formed sample is analysed by an expert. This expert defines a number of control points that characterise the gesture. Figure 4.2 shows four control points to describe a swipe right gesture. In this example, the control points `c2`, `c3` and `c4` are described with an x and y translation relative to `c1` and with a maximum radius. The size of the radius provides the rate of flexibility to match of sloppy or noisy gesture executions. By matching any Touch2D fact as the starting point (i.e. `c1` is unrestricted) automated segmentation is obtained.

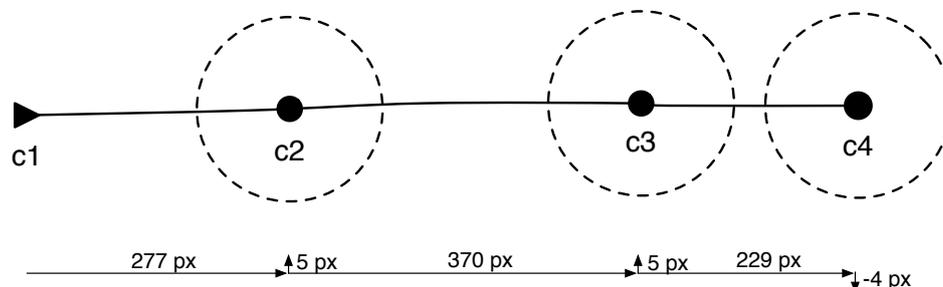


Figure 4.2: Control points for a swipe right gesture

Additionally, the declarative definition of control points in Midas will ignore all “noise” facts in between (i.e. Midas extracts a combination of

facts in the fact base that adheres the description). This implies that noisy facts during the gesture execution between these control points are completely ignored. With this flexibility, control points are highly optimised for high recall. Additional constraints can be encoded as well, for example Figure 4.3 adds a maximum Δy displacement relative to $c1$. Segmented gesture candidates can also be verified using an additional classifier such as Protractor or Dynamic Time Warping.

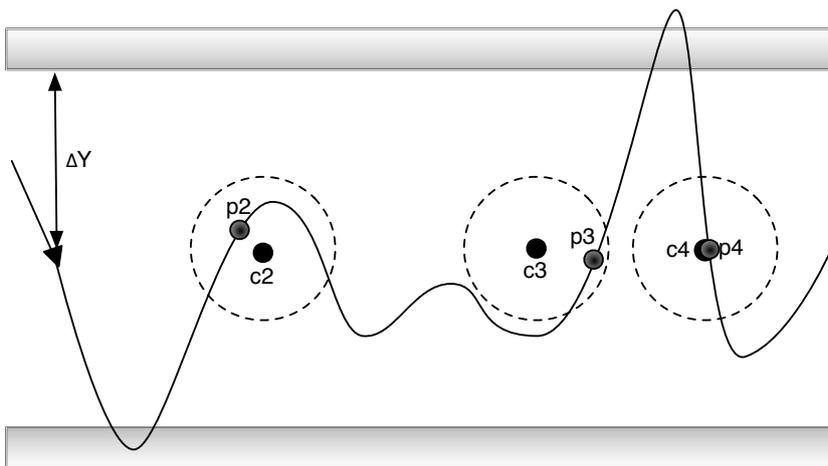


Figure 4.3: Curved line

Listing 4.22 shows a code sample of the control points pattern by implementing a swipe right gesture. The `swipeRight` rule starts with the open control point `c1` and searches for a second point `c2`, which matches the temporal and spatial constraint based on the distance between `c1` and `c2` (*lines 4 and 5*). *Line 5* attempts to satisfy `translated_nearF`, which performs a translation of the `x` and `y` coordinates of the first argument (point `c1`) with the given values of 277 and 5 pixels. The attempt succeeds if the second argument (point `c2`) lies within a circular area around point `c1` with a radius of 76 pixels. The same strategy is used for the remaining $m-2$ control points. For three-dimensional gestures, an equivalent `translated_near` attempt is provided in the `Space3D` module.

Listing 4.22: Swipe right gesture spotting rule

```

1 rule swipeRight
2   c1 = Touch2D
3   c2 = Touch2D
4   c1←beforeF c2
5   c1←translated_nearF c2, 277.px, 5.px, 76.px
6   c3 = Touch2D
7   c2←beforeF c3
8   c1←translated_nearF c3, 647.px, 10.px, 76.px
9   c4 = Touch2D
10  c3←beforeF c4
11  c1←translated_nearF c4, 876.px, 6.px, 76.px
12  c4←withinF c1, 100.ms, 1000.ms           # Maximum time
13  no { b = Touch2D                        # Bounding boxes
14     b←afterF c1
15     b←beforeF c4
16     Math.abs(c1.y - b.y) > 245.px }      # Δy
17  assert SwipeRight { x_begin ⇒ c1.x, y_begin ⇒ c1.y,
18                     x_end ⇒ c4.x, y_end ⇒ c4.y,
19                     time_begin ⇒ c1.time, time_end ⇒ c4.time }
20 end

```

Lines 13 to 16 implement additional constraints on top of this pattern and express that there should be no `Touch2D` fact that happens between the time of the matched `c1` and `c4`, where the difference of the y coordinate compared to `c1` is larger than `245.px`. The effect of this refinement is visualised by the top and bottom bounding boxes. Therefore, the noisy execution, represented by the curvy line, will match all four control points but the rule will not activate to the negated condition. Note that in standard scenarios, 2D multi-touch values are normalised in an interval between `[0..1]` to accomodate various screen sizes.

The control points pattern works for many gestures, including more fluid gestures such as a *Z* as shown in Figure 4.4. In this case, the expert developer can increase the radius of the fourth control point to allow for more flexibility. The use of control points allows for a highly precise segmentation definition without resorting to lossy approximation methods. Many existing approaches tend to either (1) smoothen input and thereby losing valuable information or (2) require a few distinctive cues to enable segmentation (for example peaks or obvious angles). However, gestures with few distinctive spatial cues, such as the swipe right example, are problematic for many techniques. When using control points, a developer has full control over which parts of a gesture should be matched closely and where variation can be tolerated.

In traditional, state machine-based implementations, segmentation is challenging due to the fact that state transitions need to be performed at

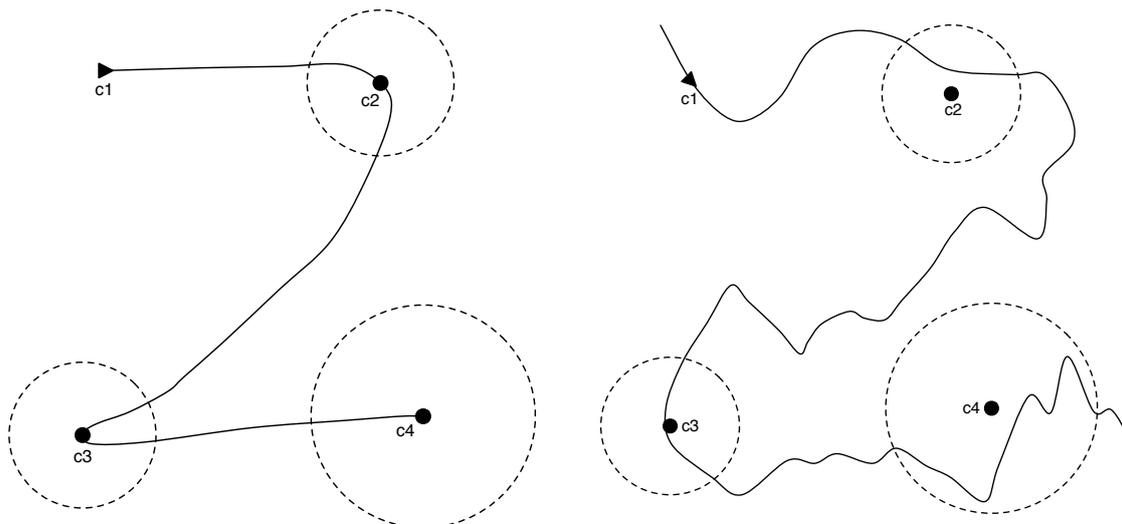


Figure 4.4: Control points for a Z gesture

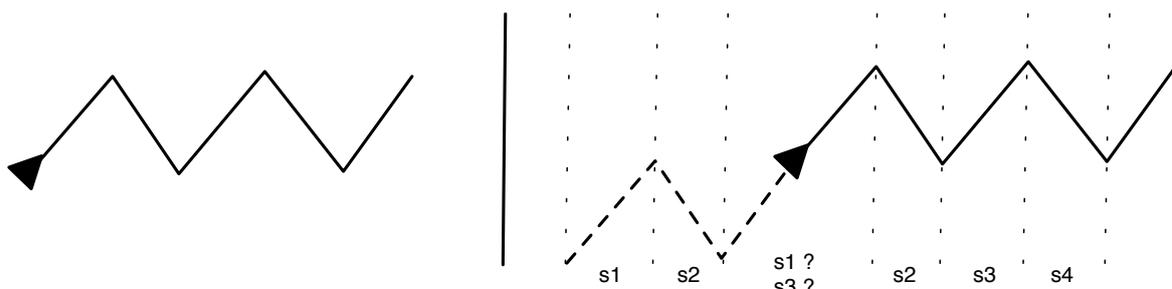


Figure 4.5: Overlap within a single gesture definition

the event level. Whenever a new event triggers the transition to the next state, subsequent data will not be used as a potential start transition. This is illustrated in Figure 4.5 showing the gesture to be spotted on the left-hand side and the ongoing processing on the right-hand side. Initially, the transition from state $s1$ to state $s2$ is valid. However, at state $s3$ the state machine, each incoming event requires a decision to reset the state machine to $s1$ or continue to wait for future data such that $s4$ might still be reached. As future data is not available at this point, state machine approaches are stuck in local decision making and potentially miss candidates.

4.5 Feature-level Fusion

Fusion at the feature level is used to disambiguate between candidates when a single modality falls short. For example, in multi-touch technology, every finger gets assigned a unique identifier. However this does not

provide information whether the fingers originate from the same hand or from different users. The fusion of existing techniques is a task typically performed at the feature level. For example, the Diamond Touch table uses small amounts of electrical current to identify touches from individual users [36]. It is important to stress that Midas does not enforce a strict separation between the data and feature level. This is advantageous for data-level recognisers requiring feature-level information. In this case, a data-level speech recogniser can for example benefit from user identification at the feature level.

4.5.1 Synchronising Streams

At the feature-level, it is common that input from different sources is not always perfectly aligned. In order to resynchronise streams in existing solutions, the programmer has to manually insert a delay parameter for one of the streams. This delay parameter is usually defined at compile time and does not always reflect runtime behaviour.

In Midas, time is handled explicitly through the use of temporal operators. Therefore, the fusion of two facts depends on their timestamps and the relative relation between them. This means that the engine will take care of the pattern matching state and no manual resynchronisation is needed. In effect, a skewed sliding window is created such that facts are matched with facts that are produced at the same time.

Our sliding window approach with a declarative approximation solves the problem found in existing systems which rely on a “current” state [41]. To discard old facts from stream *e*, a cross-template event expiration method can be used.

Therefore, different time units can be employed between templates. For instance, the time unit of `Touch2D` uses milliseconds since the Unix epoch (1/1/1970) while application or context facts can use incremental counters.

Cross-template Event Expiration As explained in Section 4.3.7, facts can be expired using rules and the automated expiration method. If we slightly adapt the rule, we can expire facts across templates to describe the intended behaviour as illustrated in Figure 2.2. Listing 4.23 retracts accelerometer facts which became too old relative to the gyroscope facts. Additionally, distinctive cues from other input streams, such as peaks or characteristic events (e.g. `Touch2D.state == Touch2D.UP`, meaning the finger is lifted from the touch surface), can also be used to expire facts.

Listing 4.23: Cross-template event expiration

```

1 rule crossTemplateEventExpiration
2     a = Accelerometer
3     g = Gyroscope
4     a ← beforeF g, 4.ms           #  $\Delta time$ 
5     retract a
6 end

```

4.5.2 Dynamic Service Instantiation

Dedicated components for feature-level fusion are common and can be embedded within the Midas language and Mudra architecture. Many of these components rely on learning approaches such as [17, 50, 121] and can therefore be treated as black-box services. A longer-term vision is that many of these machine learning components can be accommodated by declaratively specified features, thus relying on higher-level features with more contextual and higher-level attributes instead of primitive features and their limited implementation capabilities. Additionally, certain applications aim to provide a user interface with a dynamic set of available modalities. For instance, when a digital pen is connected it can be used as an alternative to the keyboard to write text in text forms. However, translating 2D pen input into characters requires a handwriting recognition process. It is often difficult for developers to define a particular value for the number of services that will need to run [6].

Midas has the ability to manage *services* at runtime. Services are small dedicated programs to fuse facts, but are implemented outside the Midas programming language. These services are orchestrated by the Mudra architecture, which are split in internal and external services. Internal services are part of the Mudra runtime and consist of algorithms implemented in C, such as Dynamic Time Warping, Protractor [106], \$1 recogniser and other tools such as peak thresholding. External services rely on network communication such that solutions which exist outside the Mudra runtime can be reused. Information exchange happens in the form of facts or serialised key-value pairs (i.e. JSON or XML) and is transparent to the developer.

In listing 4.24, we demonstrate the activation of `PeakCardinalDirections3DService`, which transform accelerometer input into cardinal direction features based on peaks in the 3D acceleration data. Note that the dynamic instantiation is performed through rules, meaning that

as soon as a new accelerometer sensor is attached, a peak thresholding service providing cardinal directions is instantiated. Services can be subscribed to receive facts (such as `Accelerometer`, line 6) and can then publish results back to the fact base (in this case the cardinal directions). Note that services can be allocated to process input from a single sensor. This greatly reduces the internal complexity of the peak detection algorithm.

Listing 4.24: Initialise cardinal direction service

```

1 rule cardinalDirectionService
2   a = Accelerometer
3   no { PeakCardinalDirections3DService { sensor == a.sensor } }
4   assert PeakCardinalDirections3DService { sensor => a.sensor }
5   s = call PeakCardinalDirections3D.initialise("cd3d")
6   call s.subscribe Accelerometer           # Subscribe to Accelerometer facts
7 end

```

In Listing 4.25, the service `RelativePositioning` is responsible for creating relative coordinates between two joints of a human body. Relative coordinates between joints are useful during the description of 3D gestures. During our first experiments, relative joint coordinates were expressed via rules. However we transformed it into a service implemented in the C programming language to optimise performance as the pattern was used for nearly all 3D descriptions. In Listing 4.25, the service is used to calculate the relative position of an event from the right shoulder to an event from the right hand. The service then asserts `RelativeJoint` facts, which can be used in rules. The final parameter of the `startNormalisedJoint` function call (line 6, `s.distance`) denotes a linear interpolation step such that two consecutive events should have at most 1 centimetre distance gap.

Listing 4.25: Initialise normalised relative joint service

```

1 rule start_RelativePositioning
2   s = StartRelativePositioning
3   j = Joint
4   no { RelativePositionService { id == s.id && sensor == j.sensor && user == j.user } }
5   assert RelativePositionService { sensor => j.sensor, user => j.user, id => s.id }
6   call RelativePositioning.startNormalisedJoint(Joint, j.sensor, j.user, s.parent, s.child, s.distance)
7 end
8
9 rule foundRelativeJoint
10  r = RelativeJoint
11  display "Relative joint for #{r.user} (#{r.parent}, #{r.child})"
12 end
13
14 assert StartRelativePositioning { id => "RShoulder-RHand", parent => 12,
15                                     child => 15, distance => 1.cm }

```

4.5.3 Asynchronous Tests

The rule `foundRelativeJoint` in Listing 4.25 relies on a conditional element, `RelativeJoint`, to process results from services. Conditional elements in rules are by default asynchronous, as they wait for a match from the fact base. However, a function invocation is synchronous, which means that it stalls the pattern matching process until the execution is completed. Therefore, Midas provides abstractions to asynchronously wait for a computation. This is similar to the `async/await` programming style advocated by the Microsoft .NET framework⁷.

An `async` construct is useful to offload calculations from the main inference loop to a background thread, such as computationally intensive functions or remote procedure calls. Many of the feature-level calculations consume a lot of processing power and in the current implementation, a continuous stream of input events would cause memory and latency problems. In an `async` case, the matching of following conditional elements will wait until the computation is completed. Thus the `async` command allows other input events and pattern matching processes to continue until the asynchronous computation is completed. An example of this feature is provided in the next section, which asynchronously verifies candidate fusion results.

⁷Asynchronous Programming with Async and Await: <http://msdn.microsoft.com/en-us/library/hh191443.aspx>

4.5.4 Verification

The following language construct to ease the programming of feature-level fusion focuses on verifying candidate results. At the feature-level, candidate results from the data-level fusion are (1) ignored if they do not match the higher-level rules, (2) fused with other sensor input or contextual results from other data-level processes or (3) verified by more heavyweight classification solutions such as Dynamic Time Warping (DTW) and Hidden Markov Models. In our *hold-and-rotate* example, the multi-touch gestures can be verified using the DTW and Protractor classification algorithms. These algorithms cannot be used directly on continuous input data as they do not support segmentation. However, data-level fusion processes from Midas, such as the `SwipeRight` gesture, provide segmented gesture candidates, which can be verified by existing classifiers. An example is shown in Listing 4.26 where the `SwipeRight` fact is matched and its details (such as the touch identity and the begin and end time segment) are used to verify the swipe right movement with a DTW template matcher. Line 1 initialises such a DTW template matching service with an existing dataset (i.e. `data/directions.json`). Then, a global `$dtw` variable can be used to refer to that service and call the `recognise` function. Midas and Mudra also support a dynamic service instantiation mechanism, which is explained in Section 5.2.4.

Listing 4.26: Asynchronous verification

```

1 $dtw = call DTW2D.initialise("data/directions.json")
2
3 rule verifyRight
4   r = SwipeRight
5   async $dtw←recognise("right", "Touch2D", r.finger, r.time_begin, r.time_end)
6 end

```

Fusing results from different classification methods, such as DTW, is called *coupled recognition*. This process is useful to obtain high F-scores (Equation 4.1) as the rules are capable of providing candidate segments out of continuous streams with a high recall, while the verification step of these candidates provides high precision.

$$\text{F-score} = 2 * \frac{\textit{precision} * \textit{recall}}{\textit{precision} + \textit{recall}} \quad (4.1)$$

4.5.5 Cross-level Fusion

The feature-level fusion of two partially related sensors can improve results of the data-level fusion processes. In contrast to pipeline solutions, Midas can easily exchange information across fusion levels by relying on a unified fact base, thereby blurring the distinction between fusion levels (this is also known as multi-level fusion). A simple example is the use of RFID data to personalise Touch2D facts. This identity information can be used to write user-specific gesture rules. In Listing 4.27, a decision-level rule personalises Touch2D facts (line 11) whenever user information is missing (line 7) and an RFID tag was scanned in the neighbourhood (lines 8 and 9). If the RFID value contained the name `Christophe`, the data-level rule `personalTouch` will trigger. This enables developers to specify user-specific rules without having to worry where the data comes from.

Listing 4.27: Feature fusion influencing data fusion

```

1 rule personalTouch
2   t = Touch2D.user "Christophe"
3   display "Christophe touched it"
4 end
5
6 rule userIdentification
7   t = Touch2D.user nil
8   tag = RFID.source "Multi-Touch Table"
9   t←nearF tag, 20.cm
10  t←equalF tag, 500.ms
11  modify t { user ⇒ tag.value }
12 end

```

Often high-level information such as application context, derived information from other fusion processes or historical decisions can also be used in order to improve the recognition rate of lower-level fusion. In Midas and Mudra, the data flow between fusion levels is therefore not strictly unidirectional. Descriptions matching high-level facts can be listed in the same rule as facts from the raw data, without the difficulty of manually maintaining intermediate results or manually aligning high frequency and low frequency streams. The cross-level fusion capability of Midas and Mudra is therefore extremely valuable and progresses beyond state-of-the-art fusion frameworks.

4.6 Decision-level Fusion

At the highest level of multimodal fusion, processes decide whether or not to deliver information to the application layer or to invoke APIs. As explained in Sections 2.2.3 and 3.1.2, decision-level fusion is difficult due to the integration of various high-level events, contextualisation, error handling and switching between dialogues. Additionally, developers need to deal with overlapping matches. For example, when two candidates based on facts f_1, f_2 and f_1, f_3 both form valid combinations for the same interaction pattern. Furthermore fusion results need to be tested with the application state or their relation with specific GUI components.

The Midas programming language provides a number of abstractions to describe the fusion of high-level events. Midas provides dialogue management capabilities, based on the abstractions provided by the SMUIML [40] language, as well as a number of activation policies to deal with overlapping matches. However, obtaining error-free results is hard and currently Midas does not provide abstractions for error handling (Section 2.1). Therefore, to aid developers with proper decision making, Midas provides a number of language abstractions on the decision level.

4.6.1 Shadow Facts

A first decision-level specific abstraction are *shadow facts*. As explained in Section 4.3.5, the details of GUI components from the application level can be represented by a corresponding fact in the fact base. As soon as information, such as the location or colour, from such a GUI component is updated, the fact is instantaneously updated with this information as well. The concept of synchronising replicating application state as facts is known as *shadowing*.

A shadow fact is a fact that represents the state of an application object. For the programming language Java, we use annotations to denote shadowing. Therefore, when a class is annotated as **Shadow**, the framework automatically reifies the instances as facts with the class name as type and the fields of the class as slots. The field annotation **Ignore** is used to exclude specific fields from being automatically reified as attributes of the fact. This accommodates synchronisation issues found in related work as discussed in Section 3.4.2. An example of how to use the Java annotation mechanism is shown in Listing 4.28. Since the **Image** class is annotated as **Shadow**, Midas will reify object instances as facts. The **Image** class has the five fields **x**, **y**, **w**, **h**, **path** and the last field has been

annotated as `Ignore`. This implies that the `path` field will not be reified as an attribute of the fact. From within the rules, the programmer can match application objects of type `Image` by using them as conditional elements.

Listing 4.28: Shadowing GUI elements

```

1 public @Shadow class Image {
2     int x, y;
3     int w, h;
4     @Ignore int path;
5     Image(String path, int x, int y, int w, int h) { ... }
6 }

```

We revisit the running *hold-and-rotate* example that is characterised by a stationary finger and a two-finger move to the right to rotate a particular GUI image. Listing 4.29 integrates application awareness via an *application symbiosis*. The symbiosis allows for a proper contextualisation, as the *Hold-and-rotate* gesture can only occur on top of an image (line 4), and filters many false positive fusion results. Listing 4.29 also demonstrates the *customisation* capabilities of Midas, as existing implementations, such as the `HoldAndRotate` gesture, can be reused in a separate context with additional customised conditions without having to modify the original code.

Listing 4.29: Contextualisation of rules

```

1 rule holdAndRotateImage
2     h = HoldAndRotate                               # Match a complex gesture
3     i = Image                                       # Match a shadowed image
4     h ← inside i                                   # which was held w.r.t its boundaries
5     call i.rotate(h.diff)                          # Call the Java rotate API
6 end

```

The symbiosis also allows more complex decision logic. For instance when images i_1 , i_2 are present on the screen and a user drags the image i_1 while crossing i_2 , the drag gesture should stick to image i_1 . This should also be possible when image i_2 is layered on top of image i_1 (i.e. image i_2 floats above image i_1 across the Z-axis). This behaviour is illustrated in Figure 4.6 and demonstrates that the integration between application information and fusion processes is crucial. Existing multi-touch approaches are mostly limited to pinning a touch event to a particular GUI component without reasoning about the history of these touch events. Therefore they have difficulties to support this scenario. An exception is the approach of Echtler et al. [47] which provides an ad-hoc *sticky* flag to specifically deal with this situation. A different scenario is when gestures are defined on the background GUI component but their activation is



Figure 4.6: Sticky drag

intended for smaller regions. For example to implement a strike-through gesture, Echtler provides a *ubiquitous* flag such that gesture events are delivered to all participating GUI components. Unfortunately, these flags cannot be easily generalised and therefore shadow facts are much more flexible.

The combination of shadow facts, activation policies (Section 4.6.3) and pattern matching reduces the accidental complexity of mapping input to GUI components. As recently described in a LWN article [159], it is straightforward for window managers to decide which window should receive the mouse events in traditional keyboard and mouse setups. Furthermore, if the window refuses the input, it can be propagated to lower layered windows. The mechanism to decide propagation for every incoming event has been working for decades. However, multi-touch input, and therefore multi-touch gestures, introduce a level of uncertainty and delayed decision process. Thus, the concept of shadow facts within a multi-touch recogniser or multimodal fusion engine is increasingly important.

4.6.2 Alternating Between Conditions and Modifiers

The ability to alternate between conditions and modifiers is particularly useful to provide intermediate feedback to the application. However, existing rule languages, such as CLIPS [57], Jess [55] and Drools [7] explicitly separate the conditional side from the modifier side. This implies that working with intermediate results becomes difficult, as adding additional conditions after a modifier requires one to split the logic into multiple rules. Additionally, the intermediate state from the first rule needs to be manually propagated to the next. This is exemplified in Listing 4.30 where the goal is to display text as soon as the user **Christophe** enters the coffee room, followed by another text notice after a second entrance. Therefore, the control flow of this program is as follows: wait for a condition *c1*, execute *m1*, wait for *c2* and execute *m2*. Existing rule languages do not support this scenario well and therefore explicit

intermediate state facts are required to test for additional conditions after a modifier was used.

Listing 4.30: Alternating between conditions and modifiers (1)

```

1 rule intermediate1
2   u = User.name "Christophe"           # c1
3   u.state == User.ENTERS
4   display "Christophe enters the coffee room" # m1
5   assert Intermediate { time => u.time }   # Store intermediate state
6 end
7
8 rule intermediate2
9   i = Intermediate                     # Load intermediate state
10  u = User.name "Christophe"          # c2
11  u.state == User.ENTERS
12  u←afterF i, 150.ms
13  display "Christophe probably forgot his coffee" # m2
14 end

```

Midas resolves this problem on the language level by relying on the type of the statements, being either a condition or a modifier. Whenever a modifier is followed by a condition, the Midas compiler automatically splits the rule (i.e. $r1$ and $r2$) and stores all bound variables of $r1$ into an intermediate fact. This intermediate fact is then listed as the first conditional element of $r2$ which restores the scope without additional developer effort. Furthermore, the intermediate fact is automatically retracted. The result is shown in Listing 4.31.

Listing 4.31: Alternating between conditions and modifiers (2)

```

1 rule entersTwice
2   u1 = User.name "Christophe"         # c1
3   u1.state == User.ENTERS
4   display "Christophe enters the coffee room" # m1
5   u2 = User.name "Christophe"         # c2
6   u2.state == User.ENTERS
7   u2←afterF u1, 150.ms
8   display "Christophe probably forgot his coffee" # m2
9 end

```

4.6.3 Conflict Resolution

Rules activate whenever a combination of facts matches their conditionals. In many scenarios a more fine-grained control over the activation of rules is needed [47]. Echtler et al. exemplify that a *press* gesture should only activate once, namely when the finger touches a region⁸. It should not

⁸This is the typical behaviour of click and double click in today's GUI toolkits

reactivate while the user continues touching it. This is true for many types of input sensors where the stream of information is decided by a refresh rate rather than a distinctive event such as a mouse-click. In the next sections we describe multiple activation policies, namely *priorities*, *bookkeeping facts* and *activation flags*.

Priorities

When designing gestures, parts of interaction patterns might overlap. For example, the gesture for a single tap overlaps with the gesture for a double tap. In Midas, priorities can be assigned to rules. For example, when the priority of the double tap is higher than a single tap (and the double tap rule retracts the matched event), the single tap rule is only triggered once when a user taps twice. Depending on the intended behaviour, a single tap rule can `wait` for the existence of a second tap in the future and block itself from activating at all.

Rules with a higher priority will always be matched before rules with a lower priority. An example of how to use priorities in rules is shown in Listing 4.32. Midas uses the keyword `salience` to denote priority levels. Unfortunately, prioritisation only works when facts are retracted from the fact base. If they were not, the current implementation would also activate the `nonPrioritisedTap` rule.

Listing 4.32: Priorities

```
1 rule prioritisedDoubleTap
2   salience 100
3   t1 = Tap
4   t2 = Tap
5   t1←meetsF t2, 150.ms
6   retract t2
7   display "Double tap"
8 end
9
10 rule nonPrioritisedTap
11   t = Tap
12   display "This will not happen for a double tap"
13 end
```

Bookkeeping Facts

Another strategy to resolve conflicts is by manually bookkeeping the current state in a fact. This pattern uses facts to influence the activation

of rules⁹. For example, rule r can express that a particular fact should not exist using the negation construct. At the same time, another rule r' expresses the assertion of such a fact, for example when the context of the application changes. When r' activates, rule r is blocked from activating due to the existence of this bookkeeping fact. This example is implemented in Listing 4.33. Shadow facts are also a particular form of bookkeeping facts. Bookkeeping facts are a powerful method to control the activations of rules, but can become quite complex to manage.

Listing 4.33: Bookkeeping Facts

```

1 rule r
2   t = Tap
3   no { ContextFact }
4   display "Tap"
5 end
6
7 rule rPrime
8   a = Image { state == Image.SHOW }
9   assert ContextFact
10 end

```

Rule Activation Flags

Based on related work on activation flags [47] and our experiments, we identified the necessity for a number of different types of rule activations, including the *one-shot*, *default*, *shoot-and-continue*, *sticky* and a *hold* flag. The *one-shot* flag is a commonly used type of activation, meaning that a single particular interaction pattern results in a corresponding single action. For example, the *put that there* example performs a single action.

Activation flags are annotated to a callback function. For example, a Ruby program can register a callback on a rule activation using the shoot-and-continue (i.e. `:sac`) flag. This is illustrated in Listing 4.35, line 1 as part of the `register` call on the lasso rule (i.e. `rule(:lasso)`).

The *shoot-and-continue* activation flag means that an interaction should be matched at least once and that following activations are activated in an online manner. To illustrate the functionality of this flag, we describe a lasso gesture. The movement performed while performing a lasso gesture (i.e. to throw a rope) consists out of a rotation of the arm above the head. This is expressed by Listing 4.34, which defines `lasso` as a sequence of relative 3D positions of the hand compared to the shoulder moving in a clockwise circle. To be more precise, line 9 to 12 expresses

⁹Bookkeeping facts correspond to an existing concept, named *control facts* in expert systems [56]

that a combination of four `EnterEllipsoid` facts (i.e. front, right, back and left) should be found. Further, line 13 describes that the control points should be matched in a sequential temporal relation with a maximum timeout parameter. The existence of these `EnterEllipsoid` control facts stems from separate rules, such as the one listed on lines 1 to 6. The coordinates in lines 3 to 4 are obtained by transforming, scaling and rotating the ellipsoid in a graphical editor. A boolean function (i.e. `enterEllipsoid`) checks whether a relative joint falls inside the given ellipsoid coordinates.

The subscription to the `lasso` rule in Listing 4.35 is annotated with the shoot-and-continue (`sac`) flag (line 1). This means that the `spray` function will be called when a user performs a lasso gesture (i.e. one full rotation) and from then on, after every consecutive quadrant of the rotation. Each consecutive step is defined by a conditional element. The parameter `p` on line 1 is an approximate percentage ($0..1$] of the entire set of conditions met in a rule and can be used to provide intermediate feedback of the matching progress. Alternatively, developers can opt for application symbiosis to implement the same behaviour.

Listing 4.34: Lasso gesture

```

1 rule enterLassoC0HandRight0
2   r = RelativeJoint { parent == Joint.HAND_RIGHT && child == Joint.SHOULDER_RIGHT }
3   r←enterEllipsoid 0.09754365, -0.45139461,
4     0.00097420, 0.1444336, 0.5310212, 0.4756404, 0, 0, 0
5   assert EnterEllipsoid { id ⇒ "HR0", time ⇒ r.time }
6 end
7
8 rule lasso
9   e0 = EnterEllipsoid { id == "HR0" }
10  e1 = EnterEllipsoid { id == "HR1" }
11  e2 = EnterEllipsoid { id == "HR2" }
12  e3 = EnterEllipsoid { id == "HR3" }
13  Time←meets4F e0, e1, e2, e3, 0.5.seconds
14  assert Lasso { time_begin ⇒ e0.time, time_end ⇒ e3.time }
15 end

```

Listing 4.35: Shoot-and-continue callback registration

```

1 rule(:lasso).register(:sac, 500.ms) { |p|
2   spray(p * sprays, 100.ms)
3 }

```

Finally, the *hold* flag allows developers to register an event handler to rules which are active over a longer period of time (such as a pose). As long as the conditions are satisfied, a callback will be triggered every x milliseconds (with parameter x defined by the developer).

Based on the requirements of our experiments and related work [47], Midas provides the following activation flags:

- A *one-shot* activation means that a single interaction should correspond to single action. An optional argument defines the minimum time interval in which the action can be re-executed.
- The *default* policy will execute the modifiers whenever a combination of the facts in the fact base matches the conditions of a rule.
- The *shoot-and-continue* (*sac*) means that an interaction should be matched at least once and that subsequent changes that are in line with the gesture definition should be processed in an online manner. Midas provides an additional (approximate) percentage of the online interaction with respect to the number of conditions that are matched.
- The *sticky* flag is a spatial-aware policy to limit the activation of other rules within a 2D or 3D distance. An optional spatial argument defines the minimum spatial interval in which another gesture cannot be triggered.
- When an interaction is matched and remains valid when new information enters, the *hold* flag will trigger the action for every x milliseconds. This allows developers to filter duplicate information.

4.7 Multimodal Language Patterns

During our experiments we observed a number of common multimodal interaction patterns. These patterns are translated into language abstractions to further reduce the development effort and rule complexity. In this section we highlight a number of these abstractions and show how they simplify the design of common multimodal interactions.

Inline Constraints

Inline constraints allow developers to express multiple filters in a single statement. For example, we can abbreviate the pattern presented on lines 2, 3 and 4 of Listing 4.36 to `e0 = EnterEllipsoid { id == "HR0" && user == "Lia" }` using inline constraints.

Listing 4.36: Implementation of a lasso gesture

```

1 rule lasso
2   e0 = EnterEllipsoid
3   e0.id == "HR0"
4   e0.user == "Lia"
5   e1 = EnterEllipsoid
6   e1.id == "HR1"
7   e1.user == e0.user
8   e2 = EnterEllipsoid
9   e2.id == "HR2"
10  e2.user == e0.user
11  e3 = EnterEllipsoid
12  e3.id == "HR3"
13  e3.user == e0.user
14  e0←meetsF e1, .5.seconds
15  e1←meetsF e2, .5.seconds
16  e2←meetsF e3, .5.seconds
17  assert Lasso {
18    time.begin ⇒ e0.time, time.end ⇒ e3.time }
19 end

```

Unification and Arrays

Existing rule languages such as CLIPS use unbound variables to enable unification between slot values. Listing 4.37 shows an example on how to unify the `name` slot of two conditional elements. It is important to stress that the `?name` variable has to be exactly the same for both conditional elements. Unfortunately, developers easily introduce typing errors in unbound variable names and these are particularly challenging to spot during debugging. Therefore, Midas introduces the use of slot access on conditional elements via the `dot` (`.`) operator. This allows the compiler to identify invalid slot access and therefore invalid unifications. This is illustrated in Listing 4.37 on line 8. Unbound variables in their original form are still supported but rarely used.

Listing 4.37: Unification using explicit unbound variables

```

1 rule unifyName
2   EnterEllipsoid { name == ?name }
3   EnterEllipsoid { name == ?neme }           # Oops (e != a)
4 end
5
6 rule unifyNameImproved
7   e = EnterEllipsoid
8   EnterEllipsoid { name == e.neme }         # Causes compile-time error
9 end

```

Next to these accidental typing errors, unification in existing systems can still be improved. In particular, Listing 4.36 and 4.37 introduce a lot of variable names to bind the various matched conditional elements. This is an additional source for errors. Therefore, Midas provides the concept of arrays to group a number of values or conditional elements. We observed that many multimodal patterns describe a sequence of conditional elements from the same template. Therefore helper operators such as `times(n)` and `sequence(n)` are introduced that expand to n conditional elements. The `sequence(n)` operator further adds a temporal constraint on each conditional element such that they can only be matched in order (of time). Thus, the expression `e = EnterEllipsoid.sequence(4); e←time_intervals [.5.s, _, .5.s]` expresses the need for four `EnterEllipsoid` conditional elements that happen in order with a maximum interval of 0.5 seconds between the first and second, and between the third and fourth match. The underscore (`_`) can be used to omit a particular value in this `time_intervals` operator. The `sequence` operator is also frequently used to describe gesture control points.

Optional Arguments

Midas supports default values for parameters in attempts and functions as illustrated in Listing 4.38. Rules or attempts that reuse this `leftF` spatial operator can therefore omit the second parameter. Default values for parameters in the middle of the argument list are also supported (i.e. `leftF(f, min = 5.px, max)` where `f` and `max` are required and `min` is optional).

Listing 4.38: Optional arguments

```
1 module Space2D
2   attempt leftF(f, min = 5.px)
3     x + min < f.x
4   end
5 end
```

Extending Modules and Templates

Modules and templates can be extended with additional slots, attempts and functions by reopening the entity. Therefore modules such as `Space3D` can easily be implemented spread across multiple files. To load other files,

the `require` primitive can be used. Furthermore, the `self`, `super`, lexical scoping rules work by leveraging the Ruby language as an internal DSL.

4.8 Developer Feedback

The presented version of Midas has been developed based on knowledge gained from experiments with a former Midas incarnation [63,141]. Often, the lack of compile-time feedback was problematic during the development of prototypes. Issues such as accidental typing errors were frequently present in the analysed code fragments. Further, the improper use of domain-specific functionality, type mismatches and duplicate binding were common mistakes that we identified.

The current Midas language and compiler provides a number of compile-time guarantees to prevent or reduce accidental mistakes in multimodal descriptions such as illustrated in Listing 4.39, including:

- An invalid number of arguments for attempts and functions is detected at compile time (line 24).
- Unification through slot access gives additional compile-time guarantees, as the use of inexistent slot access will result in an error message (line 25). This also works across attempts as all code paths are traversed during compilation.
- The use of inexistent local variables is detected and reported (line 26).
- Slot values cannot be equal to two values at the same time. Therefore the combination of lines 13, 23 and 27, which defines that the `user` slot should be both equal to `"Lode"` and `"Christophe"`, is invalid and results in a duplicate binding error.
- Slots can be optionally typed, which means that expressions such as the one on line 28 can be caught at compile time. This is because the `x` slot of a `Touch2D` template is defined as a float, while the `user` slot is inferred to be of type string (as defined in the `Speech` template). Furthermore, types can be inferred at compile time. In this example, lines 13 and 23 allow us to infer that `s.user` is of type string. Therefore, line 28 still raises an exception at compile-time when the `user` slot would be untyped.

- The syntactic distinction between attempts (\leftarrow), functions (\cdot) and modifiers (`assert/modify/retract/call`) improves the correctness of the interpretation. Attempts can only consist of conditions and function calls that do not modify the state of the application. This maintains the declarative nature of the language and ensures that no state will change when relying on existing attempts or functions in conditions.

Listing 4.39: A code sample that cause errors at compile time

```

1 template Touch2D
2   float x, y
3 end
4 template Speech
5   string user
6 end
7 module Space2D
8   attempt self $\leftarrow$ leftF(p1, p2)
9     p1.x < p2.x
10  end
11 end
12 attempt lodeSpeaksF(f)
13   f.user == "Lode"
14 end
15 attempt checkXF(f, min = 5.px, max)
16   f.x > min
17   f.x < max
18 end
19
20 rule someCompileTimeErrors
21   t = Touch2D
22   s = Speech
23   lodeSpeaksF s
24   checkXF t                                # wrong number of arguments (1 for 2)
25   Space2D $\leftarrow$ leftF t, s                  # no slot 'x' for Speech
26   r $\leftarrow$ beforeF s                          # undefined local variable 'r'
27   s.user == "Christophe"                   # slot 'user' already bound to "Lode"
28   t.x == s.user                             # invalid type for 'x', expecting float
29 end

```

4.9 Conclusion

In this chapter we presented Midas, a novel declarative language to express multimodal interaction patterns. The core idea is that developers can focus on the essential complexity of describing their interaction patterns and have to deal with less accidental complexity such as handling irrelevant events, storing intermediate state and dealing with inversion of control.

The Midas language consists of five primitive entities: *templates (and facts)*, *modules*, *rules*, *attempts* and *functions*. An input event is translated into a fact and stored in the fact base. Rules can then try to find a combination of facts that matches their conditions in a reactive manner. Modules, attempts and functions modularise the multimodal description logic into small reusable parts such that they can be composed to form more complex interaction descriptions, without requiring a developer to have a deep knowledge of all particular details [63, 64, 141].

We have explored how complex gestural interaction can be described in 2D and 3D space using a technique called *control points* [66]. This allows developers to automatically *segment* candidate gestures from a continuous stream of data while retaining full expressive control over the recognition process (which is normally lost in existing machine learning-based solutions).

Furthermore, we have shown how developers can seamlessly express *cross-level fusion* [64] using Midas. There is no manual chaining of composition boxes required (such as required by data stream-oriented solutions), and there is no loss of expressiveness compared to semantic inferencing solutions. Additionally, *shadow facts*, *alternating between conditions and modifiers* and the *application symbiosis* provide adequate language constructions to integrate declarative multimodal interaction patterns with application logic. Unification is improved through direct slot access and our syntactic distinction between attempts, functions and modifiers should improve the mental model of the programmer when implementing declarative rules.

Finally, the Midas specification allows us to provide compile-time developer feedback for many of the mistakes that can be made when declaratively implementing multimodal interaction patterns.

5

Mudra: A Unified Multimodal Interaction Architecture

To extract meaningful information from the vast amount of events produced by hardware sensors, a multimodal framework requires a number of components. This includes a programming API as presented in Chapter 4, an infrastructure to connect hardware devices to the framework, a fusion engine and an application interface to deliver results to the outside world. The three latter components form the basis of a multimodal architecture.

Existing multimodal architecture designs can be generalised in two main strands: the data stream-oriented solutions (Section 3.1.1) and the semantic inferencing solutions (Section 3.1.2). On the one hand, data stream-oriented solutions advocate a pipelining architecture where events get filtered and aggregated in a chain of composition boxes. These composition boxes need to process event per event and produce higher-level information to be used by the next composition box in the chain. Figure 5.1 demonstrates the pipeline architecture in a graphical way. This type of architecture is efficient to process many low-level events but typically lacks high-level programming abstractions [41].

On other hand, semantic inferencing solutions propose a high-level API where a number of conditions that define a valid interaction can be

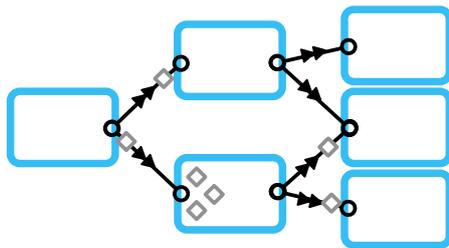


Figure 5.1: Data stream architecture

specified. Each condition is a form of requirement that can be matched with a particular event. Whenever the conditions match incoming events, a high-level event can be created. A graphical representation of these types of architectures is shown in Figure 5.2. Unfortunately, this model does not scale well to process raw input as each requirement can only be matched with a single event and therefore several combinations are not considered. Semantic inferencing architectures typically lack support for dealing with concerns such as online processing, overlapping matches, segmentation, event expiration and concurrent interaction.

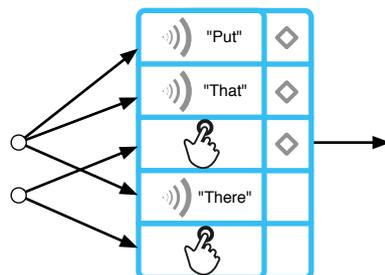


Figure 5.2: Semantic inferencing architecture

In the previous chapter we described a programming language that allows developers to express data-level, feature-level and decision-level fusion. In this chapter we discuss our unified architecture which makes the real-time execution of that language possible. We first present a conceptual architecture to illustrate the general idea. We then describe the unified architecture in more detail, including its infrastructure, distribution, core, service and application layer. Next, we analyse generic multimodal processing features of our solution, followed by specific data-level, feature-level, and decision-level fusion features. Afterwards we discuss how external multimodal tools can benefit from our approach when compiling their results into the abstraction we provide. We conclude with implementation details of Mudra.

5.1 Conceptual Architecture of Mudra

A multimodal fusion architecture must facilitate an execution engine which reacts to events according to the conditions specified by the developer. In our work, multimodal descriptions are provided in the form of declarative rules (Chapter 4) or implemented by services (Section 4.5.2, 5.2.4). Declarative rules are interpreted by a reactive execution engine (Section 5.2.3) which incrementally matches facts with the conditional elements of rules. Whenever a rule is activated, its inferred results can be *asserted* to the fact base. This creates a modular and flexible system as the execution engine automatically manages the event flow, whereas existing data-stream solutions require manual wiring of the boxes in a pipeline.

Our conceptual Mudra architecture is illustrated in Figure 5.3 and shows the aggregation of input events and results of various fusion-level processes into a single entity, the fact base. Color gradients are used to illustrate the fact that Mudra blurs the distinction between data- (red), feature- (green) and decision-level (yellow) fusion.

The fact base allows developers to combine low-level data, produced at a high rate, with high-level data produced at a low rate. This facilitates cross-level multimodal fusion and the integration of services regardless of their multimodal fusion level. We illustrate fusion across levels through six examples.

5.1.1 Motivating Examples

In this section we demonstrate six short examples with a focus on multimodal fusion across the three fusion levels. Their explanation is annotated with forward references to concepts that are discussed in this chapter. The examples are sorted from low-level to higher-level fusion tasks:

1. **Swipe Right** In the first example of Figure 5.3, we extract swipe right gestures from a raw touch input source. Its implementation is shown in Listing 5.1, which defines a data-level fusion rule. This rule uses four control points (as defined by Section 4.4.6) to express the gesture in a declarative manner. This example uses negation (Section 4.3.4), supports concurrent multi-user and multi-touch input (Section 5.3.6), sliding windows (Section 5.2.2) and partially overlapping matches (Section 5.3.3).

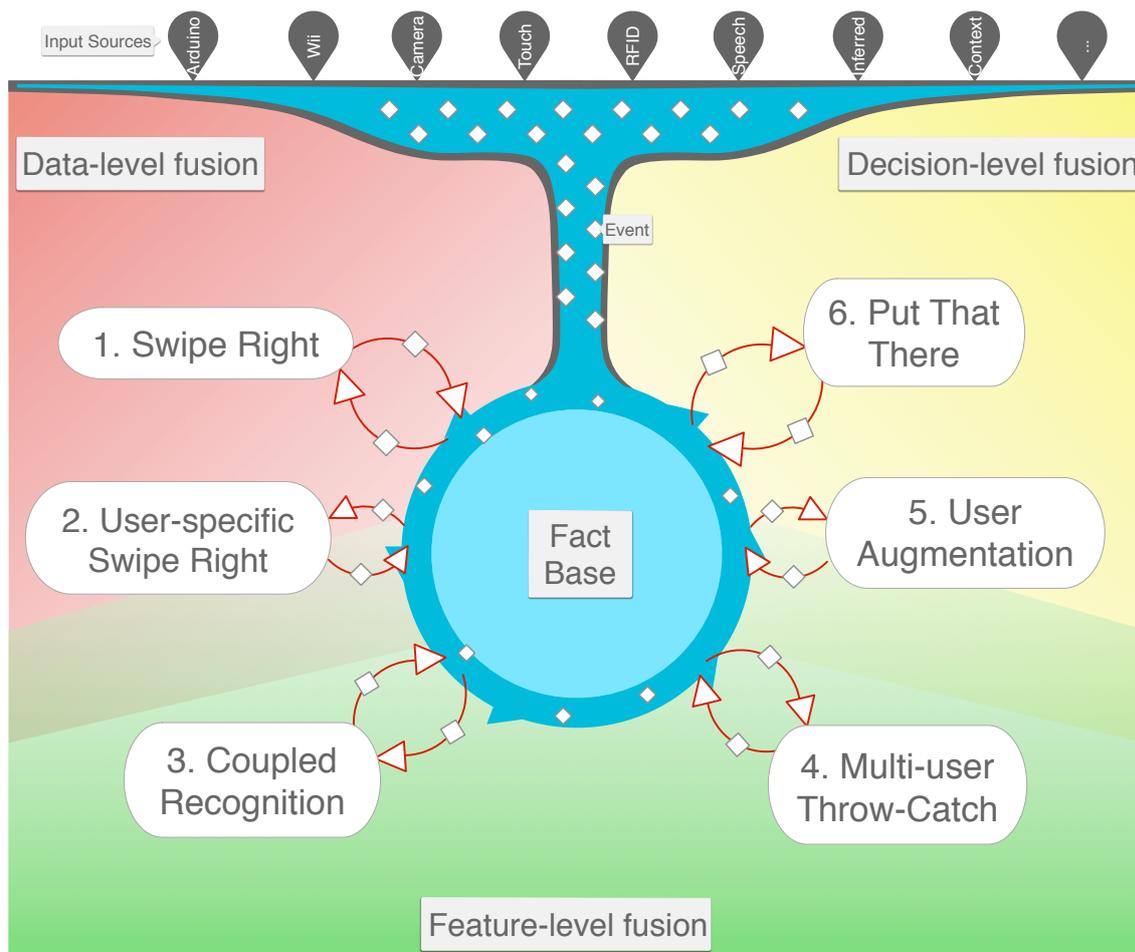


Figure 5.3: Conceptual architecture of Mudra

Listing 5.1: Swipe Right gesture

```

1 rule swipeRight
2   group finger
3   c1 = Touch2D
4   c1.user != nil
5   c2 = Touch2D
6   c1←beforeF c2
7   c1←translated_nearF c2, 277.px, 5.px, 76.px
8   c3 = Touch2D
9   c2←beforeF c3
10  c1←translated_nearF c3, 647.px, 15.px, 76.px
11  c4 = Touch2D
12  c3←beforeF c4
13  c1←translated_nearF c4, 946.px, 11.px, 76.px
14  c4←withinF c1, 100.ms, 1000.ms
15  no { b = Touch2D
16    b←afterF c1
17    b←beforeF c4
18    Math.abs(c1.y - b.y) > 245 }
19  assert SwipeRight { user ⇒ c1.user,
20    x_begin ⇒ c1.x, y_begin ⇒ c1.y,
21    x_end ⇒ c4.x, y_end ⇒ c4.y,
22    time_begin ⇒ c1.time, time_end ⇒ c4.time }
23 end

```

2. **User-specific Swipe Right** In the second example (Listing 5.2), we ensure that a `SwipeRight` gesture was executed by a particular user within a short timespan. The `QuickSwipeRightChristophe` is therefore a special gesture dedicated to an expert user. This example shows that existing rules can easily be reused and customised for new purposes. It also highlights the expressiveness and flexibility of temporal operators.

Listing 5.2: User-specific Swipe Right gesture

```

1 rule swipeRightChristophe
2   s = SwipeRight { user == "Christophe" }
3   Time←meets s.time.begin, s.time.end, 500.ms
4   assert QuickSwipeRightChristophe { time ⇒ s.time.end }
5 end

```

3. **Coupled Recognition** The third example fuses two `Circle` gestures from different recognisers, as discussed in Section 4.5.4. Line 3 of Listing 5.3 matches a `Circle` gesture detected by a rule-based recogniser while line 4 matches a `Circle` detected by the dynamic time warping algorithm. The fusion process relies on two classifiers as a form of verification to increase the precision.

Listing 5.3: Unify template matching and rules

```

1 rule startPresentation
2   group finger
3   rCircle = Circle { recogniser == "Rule" }
4   dCircle = Circle { recogniser == "DTW" }
5   dCircle.score > 0.7           # Score at least 0.7
6   rCircle.id == dCircle.id      # Same finger identifier
7   rCircle←withinF dCircle, 15.ms, 15.ms # Similar spotting
8   call PowerPoint.startPresentation # Start Presentation
9 end

```

4. **Multi-User Throw-Catch** The next example detects a “throw and catch” gesture between two users (Listing 5.4)¹. This collaborative gesture is defined by relying on more primitive gestures such as `Throw` and `Catch`. The temporal relation (line 4) specifies the order and the spatial relation (line 5) ensures that the throw is directed towards catch. This rule, in combination with our execution engine, illustrates how to enable the concurrent interaction of many users

¹The term Hoccer refers to an Android application using this gesture to share data: <https://youtu.be/2Fn1t8culTc?t=24s>

throwing and catching towards each other without requiring further implementation effort.

Listing 5.4: Multi-user throw-catch scenario

```

1 rule hoccer
2   throw = Throw
3   catch = Catch { user != throw.user }
4   throw←meetsF catch, 2.s
5   throw←in_direction_offF catch
6   call Hoccer.exchangeData(throw, catch)
7 end

```

5. **User augmentation** Example 5 (Listing 5.5) augments low-level multi-touch events with user identification. In this case, a fusion process identifies users by combining camera and RFID sensor information. Note that the RFID rate is much lower than the rate of multi-touch input devices. However, no manual re-alignment was necessary from the developer, as explained in Section 4.5.1. This example enables user-specific interaction patterns at lower levels, as for example the gesture defined in example 2, which assumes the user slot is known. Example 5 therefore highlights the ability of our approach to fuse information across layers.

Listing 5.5: User augmentation

```

1 rule userIdentification
2   t = Touch2D { user == nil }
3   tag = RFID.source "Multi-Touch Table"
4   t←nearF tag, 20.cm
5   t←equalF tag, 500.ms
6   modify t { user ⇒ tag.value }
7 end

```

6. **Put That There** Bolt’s “Put That There” interaction pattern [11] is a decision-level fusion process which combines pointing gestures and speech input (Listing 5.6). As discussed in Section 2.3.2, there is a duality between continuous (pointing) and discrete (words) input, which causes problems in existing fusion frameworks. It is challenging to decide which pointing event should be used, and how to keep track of them while the speech recogniser is still analysing its input. However, in our approach, each potential combination is analysed in an incremental manner, thereby efficiently processing both a low and high rate of input events.

Listing 5.6: "Put That There"

```
1 rule bolt
2   put = Speech { word == "put" }
3   that = Speech { word == "that" }
4   p1 = Point
5   that←equalF p1, 400.ms
6   there = Speech { word == "there" }
7   p2 = Point
8   there←equalF p2, 400.ms
9   put←meetsF that, 3.s
10  that←meetsF there, 3.s
11 end
```

These examples illustrate the need for various criteria to enable a unified fusion architecture. In the next sections we explain how Mudra, a unified fusion architecture, supports these criteria.

5.2 Mudra's Unified Fusion Architecture

We now discuss the basic building blocks of the Mudra unified fusion architecture. We propose a shared bus architecture to accommodate the need for performance and fusion across multiple fusion levels. Figure 5.4 illustrates our five-layered architecture with the fact base as a central information hub. However, it should be noted that these layers are used to separate the functionality in a conceptual manner, as all components communicate using facts.

At the *infrastructure layer*, we support the collection of arbitrary input modalities. Their low-level event details are translated into facts, which are enqueued in the fact queue of the distribution layer. The *distribution layer* is responsible for distributing facts across the interested services. It consists of an event broker that manages the subscriptions of services to particular fact types. The distribution layer is also responsible for tracking event expiration times and for automatically retracting facts from the fact base as soon as they expire. In the *core layer*, the rule execution engine matches facts to the conditional elements of rules reactively. The *service layer* enables the reuse of external fusion implementations such as ad-hoc code, various template and machine learning algorithms as well as other functionality including persisting data to disk. Finally the *application layer* provides a bridge between the fusion processes and the application context. It consists of a service interface for external applications and a shadow fact abstraction. Shadow facts enable an automated synchron-

isation of information between Mudra and the application. Furthermore, rules can also invoke function calls on shadowed objects. In what follows, we explain each of the five layers in more detail.

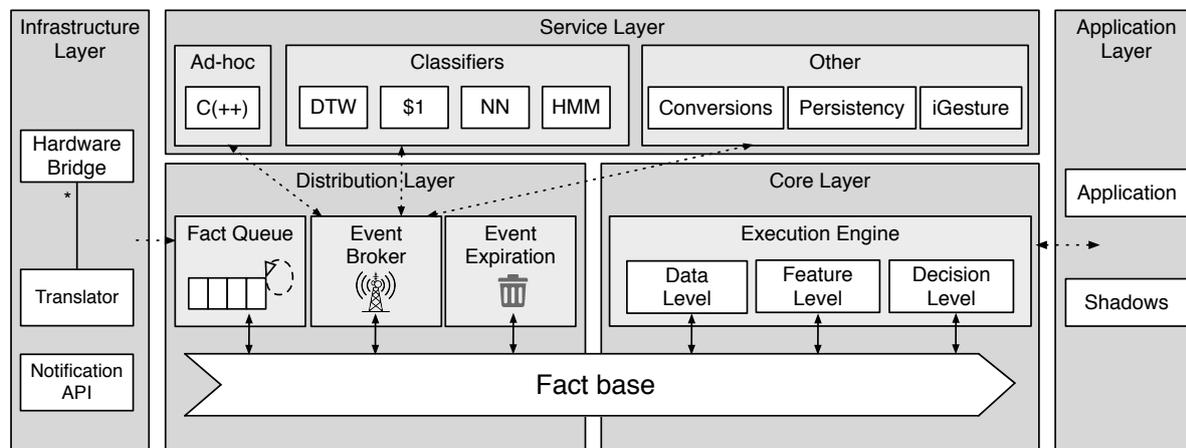


Figure 5.4: Mudra architecture

5.2.1 The Infrastructure Layer

The infrastructure layer is responsible for receiving input events from various hardware sensors (i.e. producers) and translating them into facts. On arrival, input data is converted into a uniform representation, called facts, and time stamped by the *translator* (as seen in Figure 5.4). For efficiency reasons, the translation of events to facts is done in a multithreaded fashion. Each translator operates in its own thread of execution and forwards the fact to the distribution layer. In the following, we discuss implementation details of the infrastructure layer.

Data Formats and Communication Protocols

Often information from hardware sensors can only be obtained through platform-specific or language-specific software development kits (SDKs). For example, the Microsoft Kinect SDK² is only available on Microsoft Windows³. This makes it hard to use the Microsoft Kinect in combination with sensors or applications that are only supported on other platforms such as Ubuntu Linux⁴. Therefore, Mudra provides a platform-independent network infrastructure which is used as a hardware bridge

²Microsoft Kinect SDK: <https://www.microsoft.com/en-us/kinectforwindows/develop/default.aspx>

³Microsoft Windows: <https://windows.microsoft.com>

⁴Canonical Ltd. Ubuntu: <https://www.ubuntu.com>

(see Figure 5.4) to accommodate all kinds of modalities and to support remote sensor capturing machines, such as IP cameras or wireless RFID readers. Mudra supports the following content formats:

- *S-Expressions* [134] are a notation for code and nested data based on the programming language Lisp [114]. The main advantage of using S-Expressions in Mudra is that both the code (Section 5.5.4) as well as the data are expressed in the same format. This is generally referred to as homoiconicity. Therefore, rules and input data can be transmitted over the same connection without additional protocol wrappers.
- *Comma-separated values (CSV)* [143] are frequently used to exchange data. Mudra supports template and fact definitions in the form of CSV in order to ease the loading of existing datasets and benchmarks.
- *JavaScript Object Notation (JSON)* [30] has become the dominant content format in recent years. It allows mainstream web services to provide and retrieve information from Mudra.
- *Extensible Markup Language (XML)* [16] is still widely used today and it enables the integration of many libraries that provide additional functionality such as remote procedure calls by means of XML-RPC [161].

These content formats are transmitted over a variety of networking protocols supported by Mudra:

- *TCP sockets* [21]. Raw UTF-8 [165] TCP connections are supported. Therefore, exchanging information with Mudra can be as simple as opening a TCP/IP socket and transmitting data in a particular input format. Heuristics are used on the receiver side to automatically recognise the input format, such that producers and consumers can rely on existing libraries without any format annotations or prefixes.
- *HTTP POST Requests* [52]. For example, to assert the x, y coordinates of a hand captured by a video stream, we can issue a HTTP POST requests as follows: `wget --post-data='type=Hand&x=1&y=2' http://localhost:4337/midas`

- *WebSockets* 0.9 [51]. Websockets enforce HTTP requests with full-duplex communication. This allows two-way communication between Mudra and a webpage by means of a single socket.
- *ZeroMQ* Message Transport Protocol 2.0 [20]. ZeroMQ is a communication framework that provides in-process, inter-process, TCP and multicast functionality. It has an asynchronous I/O model and supports fan-out, pub-sub, task distribution, and request-reply tasks. ZeroMQ helps integrating external services and applications that require advanced communication patterns.
- *Open Sound Control* (OSC) [163]. Open Sound Control is a content format together with an UDP protocol to share data between electronic musical instruments. However, it has been widely adopted to exchange input data, including multi-touch [83] and skeleton⁵ events. As part of our artefact, we maintain three OSC producers that are widely used in the community. The first application is an up-to-date OSCeleton for Microsoft Kinect SDK⁶ application captures the body joint coordinates (i.e. the hand of user 1 is located at x, y, z) from the Kinect SDK and writes them to an OSC client. The second application is the OSCeleton for NITE and OpenNI⁷ and provides the same OSC skeleton input stream based on the OpenNI SDK⁸. The third application is a cross-platform multi-touch bridge for Stantum based on the TUIO protocol⁹.

External producers (i.e. input modalities) and consumers (i.e. services and applications) can connect to Mudra through one of the above-mentioned protocols. These protocols and content formats were implemented based on the requirements of our use cases. However additional protocols can be easily added in the future. A set of Mudra interoperability abstractions are available as libraries for the programming languages C, Java, C# and Ruby. The libraries abstract these communication details and can easily be ported to other languages. For example, a simple `assert` procedure call can automatically serialise an object into one of the data formats and send it to Mudra over a preferred communication protocol. Our libraries provide support for asserting facts and calling functions with an automated serialisation strategy.

⁵OSCeleton: <https://github.com/Sensebloom/OSCeleton>

⁶OSCeleton for Microsoft Kinect SDK: <https://github.com/Zillode/OSCeleton-KinectSDK>

⁷OSCeleton for NITE and OpenNI: <https://github.com/Zillode/OSCeleton-OpenNI>

⁸OpenNI: <https://github.com/OpenNI/OpenNI>

⁹Stantum TUIO bridge: <https://github.com/Zillode/Stantum-TUIO-bridge>

Mudra currently supports multiple modalities including skeleton tracking via Microsoft's Xbox Kinect in combination with the NITE package or the Microsoft SDK¹⁰, cross-device multi-touch data via TUIO¹¹, speech recognition via CMU Sphinx¹² or Microsoft's Kinect SDK, accelerometer data via SunSPOTs¹³, infrared control via IR Toy¹⁴ and various other sensors via Phidgets¹⁵.

Embedded Midas Templates

Mudra embeds several Midas template definitions in its core to bootstrap new fusion processes and to interoperate with existing input producers such as OSCeletion and TUIO. For example, the `Joint` template defines which sensor was used, the user, the joint, the x , y and z coordinates and its confidence, along with built-in constants such as `Joint.HEAD` and `Joint.TORSO`¹⁶. These built-in embedded template definitions are listed in Appendix F.

Notification API

Mudra provides a notification API to inform producers about a completed translation of a particular event. This is particularly useful for load balancing between a set of services and guaranteeing the quality of service. Our notification API is compatible with Quality of Service (QoS) input devices and protocols such as TUIO. These input sources are robust against network loss or overloaded consumers in a similar way as TUIO. Protocols such as TUIO and OSCeletion use unreliable low-latency UDP communication but address packet loss by including redundant information. For example, TUIO appends a list of active fingers on the multi-touch surface and a frame sequence (i.e. incremental numbering) to each packet. In this manner, important events such as the lifting of a finger from the surface will not be omitted. In contrast to TUIO, our API additionally allows producers to track the time it takes before their events are processed by Mudra. This offers the ability to reduce the event

¹⁰OSCeletion for Microsoft Kinect SDK: <https://github.com/Zillode/OSCeletion-KinectSDK>

¹¹Stantum TUIO bridge: <https://github.com/Zillode/Stantum-TUIO-bridge>

¹²CMU Sphinx: <http://cmusphinx.sourceforge.net>

¹³SunSPOT: <http://www.sunspotworld.com>

¹⁴IR Toy: http://dangerousprototypes.com/docs/USB_Infrared_Toy

¹⁵Phidgets: <https://www.phidgets.com>

¹⁶Each body joint is part of an enum defined by the OpenNI API: <https://github.com/OpenNI/OpenNI/blob/1e9524ffd759841789dadb4ca19fb5d4ac5820e7/Include/XnTypes.h#L614>

rate by bundling or skipping some information or to increase the event rate. Vataavu et al. [157] show that a lower event resolution for finger tracking can still provide adequate recognition results.

5.2.2 The Distribution Layer

Mudra's distribution layer is responsible for three main tasks: the assertion of facts coming from the infrastructure layer, the bookkeeping of subscriptions and event expiration. We now explain each of its architectural components in more detail.

Fact Queue

Fact assertions are handled sequentially. In the current implementation, the fact base consists of a highly optimised single-threaded implementation. Therefore a synchronised fact queue is required to guarantee thread safety. Producers of facts (i.e. translations of the infrastructure layer or services) can inspect the size of the fact queue in order to assess the load of the system. For example, a TUIO translator checks the size of the fact queue and asserts or omits multi-touch events accordingly.

Event Broker

The distribution layer is implemented according to the publish/subscribe model [49]. Facts are automatically transformed into the content format requested by the consumer. The event broker will efficiently copy the data (i.e. `memcpy`) if more than one component is subscribed. Each service can subscribe and publish facts (i.e. consumer and producer). This aligns with our goal to infer knowledge from facts, whereby the knowledge itself can be represented by a fact and be consumed in its turn. Mudra's fact base automatically subscribes to all facts, but fact types can be excluded by invoking an `unsubscribe` call.

Filters Mudra's event broker system supports *filters* in order to block the distribution of facts based on a simple content predicate. Often, a consumer is only interested in a small selection of facts. Therefore, the copying of facts and their needless distribution (possibly over the network) causes overhead for Mudra, the communication bus and the consumer. This is substantial when many subscribers are interested in a fact type with particular slot values, such as a user or finger identifier. Consumers can

therefore install native content-based filters on top of each subscription. These filters cannot modify the message as it is annotated with a `const` declaration (in C), thereby reducing the cost of `memcpy`. However, filters can have state (i.e. using a dedicated `void *aFreeReference` reference) to keep track of previous facts. Alternatively, developers can opt for content-based filtering. Content-based filtering is described by rules as demonstrated in Listing 5.7.

Listing 5.7: Content-based filtering

```

1 rule filterTouch
2   t = Touch2D                                # All Touch2D facts
3   s = Touch2DService { id == t.finger }      # with a particular finger ID
4   call s.publish t                            # are published to the interested consumer
5 end

```

Communication The protocols and content formats presented in the infrastructure layer (Section 5.2.1) are also supported by the event broker. I/O communication between Mudra and external services is unidirectional. However, multiple connections can be opened from a single service to provide bidirectional communication as illustrated in Figure 5.5. By default all connections are registered as input (i.e. a producer), except when the first message by a consumer specifies otherwise, as shown in Listing 5.8 on lines 2 to 3. Producers can explicitly register as shown on lines 6 to 7.

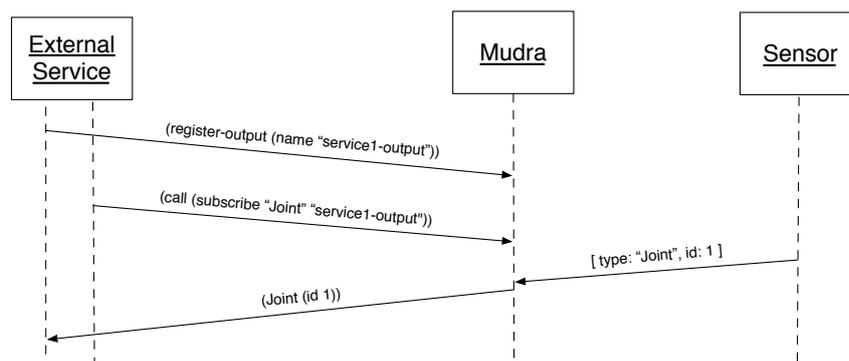


Figure 5.5: Communication protocol

Listing 5.8: Registration and communication protocol

```

1 # Register a consumer with the name "service1-output" using S-Expressions
2 (register-output (name "service1-output") (retries-on-fail 3)
3               (recover-timeout 60000) (result-router "my-result-router1"))
4
5 # Register a producer with the name "service2-input" using JSON
6 ["register-input", { "name": "service2-input", "recover-timeout": 60000,
7                   "notify-router": "my-notify-router1", "result-router": "my-result-router1" }]

```

The registration functions shown in Listing 5.8 accept a number of optional arguments:

- The option `retries-on-fail` specifies the number of retries when transmitting a message. When Mudra fails to transmit the message three times, as specified in Listing 5.8 (line 2), it will be omitted.
- `Recover-timeout` defines a timeout in milliseconds before the service registration is removed in case of disconnection between Mudra and the producer or consumer. When the producer or consumer does not reconnect within the given period of time, all queued messages and subscriptions will be removed.
- The `notify-router` option defines the name of a second connection that will receive the notification message when a particular event has been received (Section 5.2.1). This can be used to monitor processes on remote machines.
- A `result-router` option specifies the output connection to which results of a function call need to be transmitted.

The two former optional arguments enable developers to deal with volatile network connectivity, while the two latter arguments provide bidirectional communication capabilities.

Event Expiration

Event expiration is a form of garbage collection for multimodal fusion. Event expiration is challenging in a multimodal environment where segmentation and context-aware reasoning is required. In these cases, one cannot simply discard events as soon as something happens (e.g. the user leaves the room) or whenever a rule is activated. As discussed in Section 4.3.7, our approach uses *template-local relative time event expiration*: older facts are discarded when a newer fact exists which exceeds

the timespan of the old fact. Our expiration mechanism is formulated in Listing 5.9, where F_0 represents the old fact and F_n any newer fact.

Listing 5.9: Template-local relative time event expiration

```
1 discard( $F_0$ ) if  $\exists F_n: F_0.time + F_0.timespan < F_n.time$ 
```

The use of a relative event expiration mechanism, in the context of multimodal fusion, is beneficial for following reasons:

- By discarding events based on relative time, no global clock is required. This allows Mudra to distribute and receive events to and from multiple machines without precise time synchronisation. This happens in the context of remote input devices, when fusion is performed by external services that exist on other machines.
- It enables developers to use exactly the same setup between benchmarks, debugging sessions and production. Existing fusion and rule engines often require slight modifications to the multimodal descriptions for benchmarking and experimentation purposes where the input event rate is higher in comparison to the production case. This involves rewriting rules, modifying benchmark data and changing system flags in order to eliminate the global clock dependency. In Mudra, time is specified optionally which allows time-annotated benchmarks and experiments to run at full processing performance without affecting the outcome.
- It supports the alignment of non-synchronised streams. Some multimodal input sources introduce a bigger latency than others. For example, speech input sources would be hampered by a high-latency speech recogniser (i.e. a speech recogniser waits until a word or sentence is pronounced completely). Existing fusion approaches use a delay construct to address the latency mismatch between multiple input sources. However, (1) the delay parameter is hard-coded and needs to be defined at compile-time, (2) this requires modification from developers when applying the system in another setup and (3) it implies that unstable streaming, for instance due to network latency, results in further misalignment of the data. Using rules, no time synchronisation of facts is required. In Mudra, facts are discarded based on the relative time between multiple event sources. This event expiration was described in Section 4.5.1 and is an al-

ternative for the hard-coded expiration mechanism described in this section.

- Facts produced at a low rate can exist much longer in the fact base without having a major impact on the memory usage. As event expiration is relative for each template, templates with a low rate of instantiation can be configured with a high default timespan to remain present in the fact base for a longer period of time. Therefore, it is easy for developers to store high-level information such as context or previous recognition results to increase accuracy for the future fusion.
- Developers can annotate individual facts that contain highly valuable information manually with a high timespan. The timespan value is part of a first-class slot and allows any fact to outlive the standard expiration time for its template.

Mudra’s automated event expiration is implemented using a double linked priority (heap-based) queue for each template. For each new fact the relative timing is compared to the nearest fact to expire. In practice, the fact base evolves using a sliding window for every individual fact template.

Next to the relative time event expiration method, Mudra offers a sliding window based on a *bounded size*. This option expires the oldest fact of a particular template in case a new fact is asserted that no longer fits in the window size. This method is used in combination with the relative time event expiration method in order to improve reliability on machines with a limited amount of volatile memory (i.e. DRAM). However, in contrast to time-based reasoning, a bounding size offers no correctness guarantees as potentially matching facts can be discarded prematurely.

5.2.3 The Core Layer

The Mudra core layer consists of an inference-based execution engine in combination with a rule base. As discussed in Chapter 4, rules are driving the reactive computation and allow the description of complex multimodal interaction patterns via small reusable attempts. The execution engine matches facts against conditional elements of a rule. It is based on the C Language Integrated Production System (CLIPS) [57]¹⁷. CLIPS is an

¹⁷CLIPS: <http://clipsrules.sourceforge.net>

inference engine and expert system tool developed by the Technology Branch of the NASA Lyndon B. Johnson Space Center. We have substantially extended its codebase in order to support a continuous, reactive evaluation mechanism. The Midas language presented in Chapter 4 is an abstraction on top of the CLIPS language, as discussed in the compilation process (Section 5.5.1). We adopted CLIPS because of its highly optimised implementation which is based on the Rete algorithm [54].

A Rete Network

Rules provide many-to-many matching functionality. Therefore, they involve a search through all possible combinations of facts which correspond to conditional elements. A naive brute force implementation is infeasible when matching a massive amount of input events to many multimodal rules. Rete is an algorithm designed to speed up the pattern matching problem in logical rules by trading computation time for storage space. Additionally, the eager evaluation semantics of the Rete algorithm, in contrast to related algorithms such as TREAT [117], LEAPS [118] and Constraint Handling [156], makes it an excellent fit to our problem domain. This means that every new input event that satisfies a rule, will trigger as fast as possible. This is at the cost of additional processing power and memory consumption for computing intermediate results. However practise and research shows that reducing latency (i.e. the delay between input action and output response) is extremely important for human interaction with computers [110].

Rete analyses the conditions of all rules to create a direct acyclic graph, such that:

- Conditions shared between the antecedent of multiple rules are unified in the Rete graph in order to reduce (or eliminate) redundant computation.
- Partial matches are cached in memory to avoid re-evaluation of all facts whenever a change (i.e. assertion, modification or retraction) to the fact base happens. This implies that Rete works best when the majority of facts in the fact base is stable. This assumption is true in our multimodal fusion context, which involves matching events over a period of time. Only a few new facts are asserted at each point in time, compared to the vast amount of facts already present in the current sliding window.

- Retraction is optimised by retaining pointers to the partial matches.

This direct acyclic Rete represents a graph with filter and join operations to match facts to rules. This allows us to feed input facts at the top which are then filtered and combined with other events to extract more meaningful information. As an example, we derive a Rete graph based on a declarative description of Bolt’s multimodal interaction rule as seen in example 6 presented in Section 5.1.1). The Rete graph corresponding to Listing 5.6 is shown in Figure 5.6 and consists of four types of nodes: a template node, an alpha node, a join node and a terminal node. A *template node* is used as an entry point for dispatching facts based on their type. *Alpha nodes* are used as a simple filter for concrete slot values. In this example, this corresponds to filter facts based on the words “put”, “that” and “there”. *Join nodes* (also known as beta nodes) perform joins between “left” and “right” input provided by other nodes. These correspond to the implicit “and” operation that connects two conditions in the antecedent of a rule. For example line 9 of Listing 5.6 defines a temporal relation between the time slot of the `put` and `that` conditional elements. This relation is translated into a join operation that defines the cross product of all `put` and `that` elements and only passes those that satisfy the temporal condition. Join nodes also keep track of their joint matches, which means that they retain partial matches. This is done by storing a *token* which is a pair that refers to both conditional elements for which the join is successful. Finally, *terminal nodes* are triggered when all conditions are met and invoke the necessary modifiers. Duplicate matches are filtered by performing verification (Section 4.5.4) and conflict resolution (Section 4.6.3).

The Rete algorithm has been implemented in many rule engines, including CLIPS, Drools, Open Rules¹⁸, Jess¹⁹, IBM Operational Decision Management²⁰, BizTalk²¹, Nools²². Mudra relies on the Rete implementation of CLIPS, which is designed as a forward chaining (thus reflecting an data-driven system) and provides many enhancements, such as hashed alpha and beta memory in order to speed up lookup.

¹⁸Open Rules: <http://openrules.com>

¹⁹Jess: <http://herzberg.ca.sandia.gov>

²⁰IBM Operational Decision Management: <http://www.ibm.com/software/products/en/odm>

²¹Microsoft BizTalk: <https://www.microsoft.com/en-us/server-cloud/products/biztalk>

²²Nools: <https://github.com/C2F0/nools>

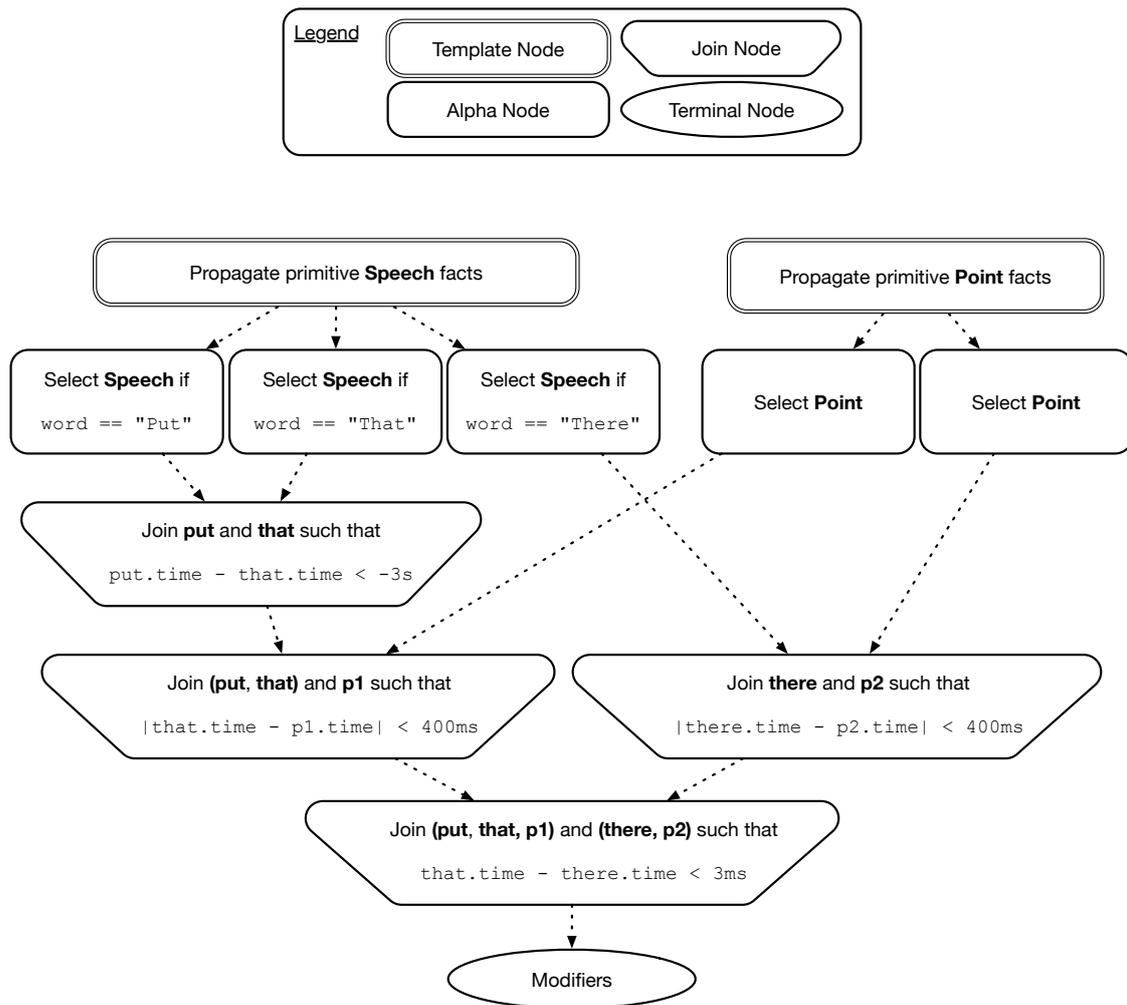


Figure 5.6: Rete network for Bolt's interaction rule

5.2.4 The Service Layer

The goal of Mudra's service layer is to integrate third-party fusion processes and gesture recognisers. Each such process or recogniser is called a *service*. Services can subscribe to one or more fact types (such as `Tuio2DCursor` or `Vibration`) in order to receive a copy of any future fact assertion for that particular type. The delivery of facts and notifications is handled asynchronously by the event broker, as explained in Section 5.2.2. In Figure 5.4 this is illustrated via the dashed arrow from the event broker to various integrated services. We assume that external code can be wrapped in a service component which can send and receive facts.

We first discuss the ability of Mudra to instantiate and discover services at runtime. We then demonstrate the integration of existing processes

into the Mudra architecture and we illustrate its interoperability with external frameworks such as iGesture [145].

Dynamic Service Instantiation

Mudra allows developers to instantiate as many services as needed based on runtime requirements. For example, *input sources* can become available dynamically. As explained in Section 4.5.2, rules can be used to describe dynamic service instantiation. However, for other scenarios, service instantiation can depend on more refined information based on the details of *input events*. This includes the number of users or the number of fingers on a multi-touch surface. This means that the number of input sources does not map directly to the number of services, but rather depends on a particular data pattern, as presented in Listing 5.10. In Listing 5.10 services are instantiated whenever the calculation of the relative position of the left hand and the left shoulder for each individual user is needed. The call of `RelativePositioning.startNormalisedJoint` initialises the service and this service subscribes to fact types it is interested in. Such a highly dynamic service instantiation is particularly useful in a multi-touch scenario where every finger should be monitored for gestures. Existing gesture recognisers such as DTW, Protractor or the \$1 Unistroke Recogniser require a new recogniser instance for each finger on the multi-touch surface.

Listing 5.10: Dynamic instantiation of services (line 8)

```

1 rule startRelativePositioning
2   s = StartRelativePositioning
3   j = Joint
4   no { RelativePositionService { id == s.id && sensor == j.sensor && user == j.user } }
5   service = String.concat("RP-", s.id, "-", j.sensor, "-", j.user)
6   assert RelativePositionService { sensor => j.sensor, user => j.user,
7     id => s.id, service => service }
8   call RelativePositioning.startNormalisedJoint(service, "Joint", j.sensor, j.user,
9     s.parent, s.child, s.distance)
10 end

```

Service Pooling

When a consumer gets saturated by an excessive amount of received facts, an ever increasing latency bottleneck is created. This is a well-known problem of publish/subscribe systems and can be overcome by balancing tasks over redundant subscribers [72]. In this manner, Mudra provides a *pooling service* which distributes tasks across replicated services. Service

pooling can, for instance, be used to delegate classification tasks to a pool of machine learning classifiers. This eliminates potential bottlenecks and improves responsiveness.

Service Discovery

The Mudra architecture supports spontaneous service discovery. Services can announce themselves in the network through the ZeroConf protocol [23, 24], which is based on multicast DNS technology. Whenever Mudra discovers a service on the local network, it translates the name and connection properties into a fact such that its presence can be detected by rules. Mudra also announces itself as a ZeroConf service, thereby enabling the connection of multiple Mudra instances across a network. Note that the orchestration of multiple instances in combination with the subscription model can become complex. Additional research is being conducted to automatically load balance and distribute the interpretation process of rules [151].

Ad-Hoc Services

Mudra integrates a number of built-in ad-hoc services based on C/C++ code, such as:

- The *peak detection* service monitors a stream of events using a moving average over a time window and asserts a fact whenever a particular threshold is crossed.
- The *vectorisation* service transforms 2D and 3D events into cardinal directions.
- The *translation* service moves x,y (and z) coordinates by the provided parameters.
- A *scaling* service scales x,y (and z) coordinates so that they fit in a corresponding interval. This allows developers to use normalised spatial measurements across rules and services. For example, the x and y coordinates of multi-touch events can all be confined to a 2D bounding box between 0 and 1.
- An ad-hoc *rotation* service rotates the x,y (and z) coordinates by a given angle. Currently Mudra only supports principal rotations (i.e. across the x , y or z axis).

- *Linear interpolation* is available as a service to artificially increase the rate of input data. Due to hardware specifications or networking issues, input sources may not always be able to provide a stable data rate. Linear interpolation is often beneficial for peak detection, control points and other mechanisms that require a stable input rate.
- The calculation of *relative positions* can be offloaded to an ad-hoc service. Relative positions are used for different purposes including the analyses of 3D gestures (i.e. to describe movement from the hand relative to the shoulder).

It should be noted that other spatio-temporal ad-hoc functionality can be embedded in Mudra. For example, `Time←before` and `Space2D.similarity` are integrated in the language runtime and do not require a service abstraction. Extending the built-in functionality of Mudra is relatively easy, as these services were implemented in a few hours in order to run our experiments.

Classification Services

A classification service assigns a list of facts to a particular class (also known as a label), thereby raising the abstraction level. Mudra integrates a number of existing classification methods as a service, in particular DTW, the \$1 Unistroke Recogniser and Protractor. These three template classifiers output a label for which the distance between an incoming sequence and a pre-recorded labelled sequence is the smallest. Therefore, this method requires annotated training data to be persistently stored.

In contrast to template classifiers, learning-based methods, such as neural networks (NN) or Hidden Markov Models (HMM) can be used. These are currently not integrated in Mudra, but are available as an external service by frameworks such as iGesture or Weka (Section 3.3.3). Mudra provides the built-in template for representing classification results shown in Listing 5.11. Each classification task and its result are linked using a unique query identifier (i.e. line 3). This unique query identifier is used to know which classification task has been completed.

Listing 5.11: Asynchronous verification

```

1 template Classification
2   label                               # the classification result (i.e. label)
3   query                               # original query identifier
4   time_begin                          # the time of first relevant fact
5   time_end                            # the time of last relevant fact
6   score                               # score given by the classifier
7   ratio                               # score interval defined by the classifier
8   recogniser                          # the type of classifier
9   origin                              # the type of input data
10  extra                               # additional information by the classifier
11 end

```

Most classification services rely on segmented data. Therefore the combination of Midas and classification services provides a powerful platform to describe multimodal interaction patterns. On the one hand, Midas rules typically focus on extracting meaningful patterns from large continuous streams of input data while providing software engineering constructs such as modularisation and composition. On the other hand, classification techniques offer the ability to verify the results obtained by rules (as seen in Section 4.5.4). It also allows developers to detect patterns which are too irregular to describe using rules.

Other Services

In this section we discuss three remaining built-in services of Mudra, including a conversion service, a persistency mechanism and an integration service with the iGesture classification framework.

Conversions Often facts need to be converted from one format into another. For example, TUIO-based multi-touch information (i.e. `Tuiio2DCur`) is converted into a `Touch` template in order to be compatible with existing rules. This conversion can be done by rules or by a built-in conversion service, which is more time efficient thanks to the use of multi-threading. Moreover, such a conversion service can use existing C programming code for translation, rotation, scaling and interpolation without communication or interpretation overhead.

Persistency As volatile memory is rather limited, the fact base needs to expire old data, thereby providing a limited scope of facts in time. In order to offer developers the ability to persist old facts and later recall them, Mudra provides two persistent services based on text files and an

SQLite database. Storing facts onto disk is useful for debugging purposes: event input is gathered and later replayed in order to verify the dynamic behaviour of the multimodal fusion process.

In the text-based persistent service, facts are serialised to a file using one of the supported data formats defined in Section 5.2.1. The developer simply has to subscribe the persistent service to the intended fact type. Developers can also use content-based filters for more fine-grained storing.

Alternatively, facts can also be stored to a relational database through the SQLite persistent service [125]. Such an SQLite database service is initialised with a filename as shown in Listing 5.12. Facts published to an SQLite service are translated into database records whereby every template corresponds to a database table and each slot to a column. Facts can later be retrieved through dedicated `select` and `import` methods. The `select` method simply retrieves facts from the database, while the `import` method also asserts them into the fact base. Note that imported facts will expire using the same expiration mechanism described in Section 5.2.2. However, many options are available to deal with this issue, such as overriding the timespan of imported facts, convert them into a separate template or publish them to a service outside the fact base.

Listing 5.12: SQLite persistent service

```

1 $sqlite = call SQLite.initialise("data/storage.sqlite")
2 rule importantLesson
3   s = Speech { user == "Beat" || user == "Wolf" }
4   # Persist everything uttered by Beat or Wolf
5   call $sqlite.publish s
6 end
7 rule recallLessons
8   u = User
9   # (Re-)Assert all speech between yesterday and now
10  call $sqlite.import Speech, u.time - 24.hours, u.time
11  # Select all speech in a list (i.e. without assertion)
12  ss = call $sqlite.select Speech, u.time - 24.hours, u.time
13  # Assert facts according to a custom SQL statement
14  call $sqlite.importWhere "SELECT * FROM Speech WHERE user == ?", u.id
15 end

```

The combination of rules and persistent data storage provides an elegant interface to capture training data as all functionality of Mudra can be used during data gathering and annotation. For example, rules can filter noise by analysing contextual information, either from the application logic or multimodal sensor input. Likewise, input data can be automatically annotated by rules in order to easily bootstrap machine learning services.

A Generic Service-oriented Classification API

In order to accommodate these machine learning services, Mudra offers a generic service-oriented classification API. This API generalises common properties of classification frameworks, such as Weka and Orange [35]. These classification frameworks typically consist of three basic methods, namely `initialise`, `recognise` and `remove`. A client of this framework instantiates and configures classification algorithm by using the `initialise` method which configures the parameters. The `recognise` method transforms a set of events into a classification label. The `remove` method is used inform the framework that a particular service will no longer be used. The service API is presented as a Java interface, as shown in Listing 5.13. In this section we focus on the classification of 2D gestures, but the same API can be used for other data types.

Listing 5.13: Interface definition of a gesture classification service

```

1 public interface GestureServiceAPI {
2     public boolean initialise(String identifier,           // Service identification name
3                             String algorithm,           // Name of the classifier
4                             Map configuration);
5     public boolean recognise(String identifier,
6                             int queryId,
7                             List<Map> trajectory,      // key-value pairs (i.e. x,y coordinates)
8                             int maximumResults = 1,
9                             double minimumScore = 0);
10    public boolean remove(String identifier);
11 }

```

The Initialise Method Each classification algorithm requires a number of parameters, such as which features to be used, a minimal threshold for a successful recognition of a gesture or the set of gesture samples to be used. Therefore, our API supports the initialisation of gesture recognisers with a number of parameters which may vary for different algorithms. As shown in the `initialise` function signature in Listing 5.13, the method accepts a unique `identifier` to specify the service name, together with the name of the `algorithm` and its `configuration`. This `configuration` is list of key-value pairs to configure the classifier. The `initialise` method will set up a communication channel to the service and return whether or not the classifier was created.

The Recognise Method In order to classify a segmented gesture candidate (i.e. a trajectory), the `recognise` method is used. The method takes as parameters an identifying name for the initialised service, a query identifier (i.e. `queryId`), a list of points and optionally a maximum number of results to be returned, as well as a minimal threshold for gestures to be

recognised. Note that the `recognise` method returns a `boolean` instead of a classification result. It indicates whether the message was well-formed, queued correctly and if an instance of the specified classifier is active. Our Mudra wrapper for this classification API supports the `async` construct, presented in Section 4.5.3, to hide the `queryId` and `boolean` details. Through the `queryId`, which is copied into the results, we are able to resolve the asynchronous requests with their values.

The Remove Method Finally, initialised gesture classifiers can be removed through an explicit invocation of the `remove` method. This method simply takes the registered name of the classifier. The service is then responsible for cleaning up all algorithm-specific resources.

This service-oriented classification API provides a simple mechanism to integrate various existing classification algorithms. We implemented DTW, the \$1 Unistroke Recogniser, Protractor and iGesture framework extensions using the same three basic methods.

iGesture The iGesture framework [145] focuses on helping developers to parameterise the gesture classification process. It offers various gesture classification algorithms, such as Rubine [137], SiGeR [152] and Hidden Markov Models [160], that perform well on 2D segmented trajectories. Most services discussed in this chapter rely on the data-centric approach (i.e. publish and subscribing facts). However, we integrated iGesture by using an XML-RPC API to demonstrate the extendibility of the Mudra architecture [99]. In the next section we define a simple generic service-oriented gesture computing API to accommodate the integration needs of existing classifiers in multimodal frameworks.

Distributed Mudra Mudra instances running on different machines can be discovered and can act as a service to each other. This way, rules and facts can be transmitted to other Mudra instances to offload the computation from a single node to another. In order to connect Mudra instances via IPv4 [71] or IPv6 [34], three `Mudra.connect*` operators are provided, as shown in Listing 5.15. When a connection was successful, a fact is created in the fact base which can be used to offload data to the new Mudra service. Listing 5.14 demonstrates the offloading of pen input data to a Mudra instance called `ly-1-11`.

Listing 5.14: Offload pen input data to a specific Mudra instance

```

1 rule offloadToLy111
2   m = Mudra { name == "ly-1-11" }
3   p = Pen
4   call m.publish p
5 end

```

Listing 5.15: Connecting to a remote Mudra instance

```

1 call Mudra.connect "mudra-ly-1-11", "ly-1-11", 4337, Mudra.DATA_FORMAT_SEXP_V1
2 # Mudra.connect_ipv6 <router-name>, <hostname>, <port>, <format>
3 # Mudra.connect2 <router-name>, <hostname>, <port>, <format>,
4 #                               <retries>, <retry-sleep>, <retry-timeout>

```

5.2.5 The Application Layer

Finally, Mudra's application layer allows multimodal applications to communicate with Mudra by subscribing and publishing facts. An interesting consequence is that at any multimodal fusion level, rules can benefit from application context in order to improve accuracy. For example:

- At the data level, multiple fingers can be grouped together based on their spatial relation to GUI components that exist underneath. This simplifies the grouping problem when processing raw data to form gesture candidates. This was illustrated in Listing 4.29.
- When the application provides user identification information, specific features can be enabled or disabled for input of that particular user, as illustrated in Listing 5.2. Such user-specific features allow developers to process pen, accelerometer or eye input for users with a form of tremor [138].
- At the decision level, application state allows the adaptation of dialogue rules (Section 3.4.2). Therefore multimodal fusion can adapt to a context of use (e.g. car, home or work), type of task (e.g. information search or entertainment) or type of user (e.g. visually impaired or elderly) [38].

Shadows

As mentioned in Section 4.3.5, Mudra's application layer provides the ability to replicate application objects as shadow facts. This allows

the developer to reason over application level entities, such as GUI components, inside the multimodal fusion architecture. Whenever a class is annotated with **Shadow**, the Mudra library automatically reifies its instances as facts with the class name as type and the fields of the class as slots.

Currently, shadowing is supported for the Ruby and Java programming languages. The synchronisation process between Ruby and Mudra is implemented using Ruby's reflection properties. For Java, we rely on the JavaBeans technology [60]. Note that the shadow facts are not automatically kept consistent with their corresponding object. Therefore, for every change in state, developers have to call the `firePropertyChange(oldValue, newValue)` method. This notifies the shadowing process and activates synchronisation. Fortunately, the Java Mudra library provides an option to rely on the Java reflection API, which automatically propagates changes from (and to) Java objects using `get*` and `set*` methods. A publish/subscribe pattern is used (as defined by JavaBeans) to synchronise information. For Ruby, we wrap all fields to intercept state changes whenever an assignment is used.

5.3 Multimodal Processing Concerns

In this section we discuss the multimodal processing concerns of the Mudra framework based on the criteria we defined in Chapter 2 (Section 2.3).

5.3.1 Online Processing

The ability to process input data in an online manner allows applications to provide user feedback in a responsive manner. Mudra enables online fusion processing in three ways:

- Rules trigger immediately as soon as their conditions are satisfied. A developer is thereby equipped with fine-grained control over the feedback process. Additionally, the Midas language provides the ability to seamlessly alternate between conditions and modifiers within a single rule. Moreover, through abstraction and composition of attempts and modules (Sections 4.3.1 and 4.3.2), developers are encouraged to modularise as much code as possible, thereby enabling the use of modifiers for fine-grained, online user feedback.

- The Mudra engine offers a number of activation flags to control the execution of rule modifiers, as shown in Section 4.6.3. These flags, such as the *shoot-and-continue* flag, specify standard behaviour to implement online user feedback.
- When multiple rules match after the assertion of a fact, rules with a higher priority value (i.e. salience) gain priority. When the priority is equal or undefined, the order in which the rules are defined is used as an ordering mechanism. This helps developers to prioritise rules in an online manner.

5.3.2 Offline Processing

Offline analysis happens based on distinctive cues which indicate that the mandatory input data is available to make an informed decision. In rules, simple cues such as a pen down and up event, can be expressed as conditions to initiate an offline processing task on the data captured between both events. As rules are able to describe complex patterns, more complex offline cues are also supported. Mudra embeds several services, such as peak detection (Section 5.2.4) to further ease the implementation of offline processing tasks. Additionally, the `wait` keyword (Section 4.1.1) allows developers to specify explicit delays even after conditional elements are matched. This is useful to ensure that all data (i.e. input and GUI) is available to negated patterns.

5.3.3 Partially Overlapping Matches

Frequently developers write rules that rely on the same conditions. Therefore, a single fact can match multiple candidates from different rules. Figure 5.7 demonstrates the partial overlap of two candidate gestures in the middle of an event sequence. In this case, each event following `c1` is potentially part of the *S* as well as the *Circle* gesture.

In related work, stateful implementations require developers to reject the potential start of the *Circle* gesture. One way to overcome missing overlapping matches is to extend Alon et al.'s [2] approach, which uses an offline classifier that automatically generates this subgesture list during the training phase. However, there are still two cases where potential candidates are incorrectly rejected. Firstly, in Alon et al., only the best scoring candidate gesture is added to a candidate list for each input event. Secondly, in many cases the subgesture does not follow the exact

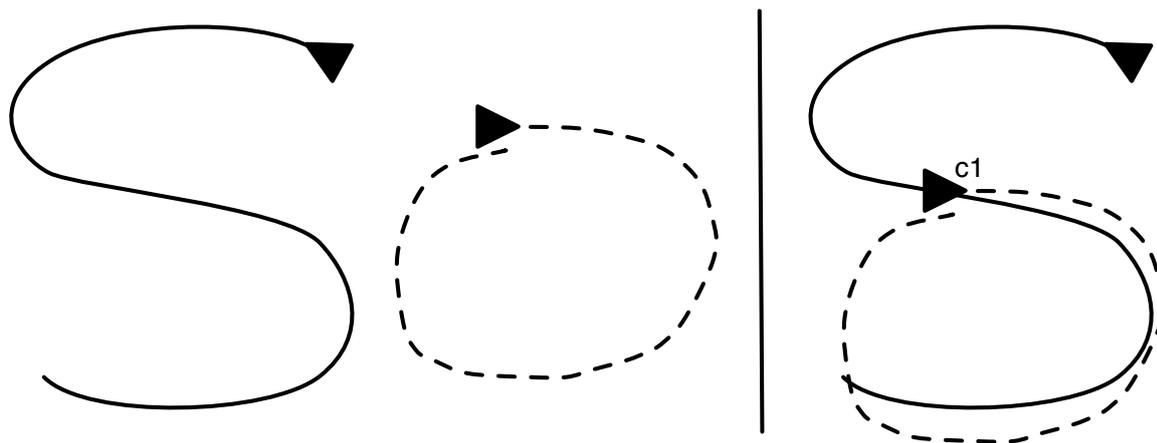


Figure 5.7: Overlapping within multiple gesture definitions

trajectory of the supergesture and if the supergesture is for example rejected at a later stage, subgestures might be incorrectly missing from the candidate list.

In Mudra, the Rete network automatically keeps track of all candidate solutions from the same input data using tokens. Like this, a particular input event can be shared between multiple rules, allowing partial overlap. This feature enables the processing of concurrent multi-user interaction and adequately deals with the *identity and grouping problem* by validating all potential combinations. Bolt's example (Example 6 described in Section 5.1.1) demonstrates the need to make a decision at the wrong level of abstraction in many frameworks. As soon as an additional pointing event comes in, a decision to overwrite the existing partial match or to reject the new potential match has to be made without proper information about the context or future events. This is illustrated in Figure 5.8 where events $e1$ and $e2$ are mapped to the same condition. Existing techniques rely on ad-hoc solutions without adequate software engineering guarantees. In our work, both combinations will be matched which means that subsequent conditions decide which combination to execute. This disambiguation can be resolved by implementing additional rules, configuring conflict resolution (Section 4.6.3) or encoding a solution at the application level. Most importantly, no potential matches will be filtered due to the partial overlap feature. In a similar way, segmentation also requires keeping track of many alternatives until a decisive conclusion is formed, as explained in the next section.

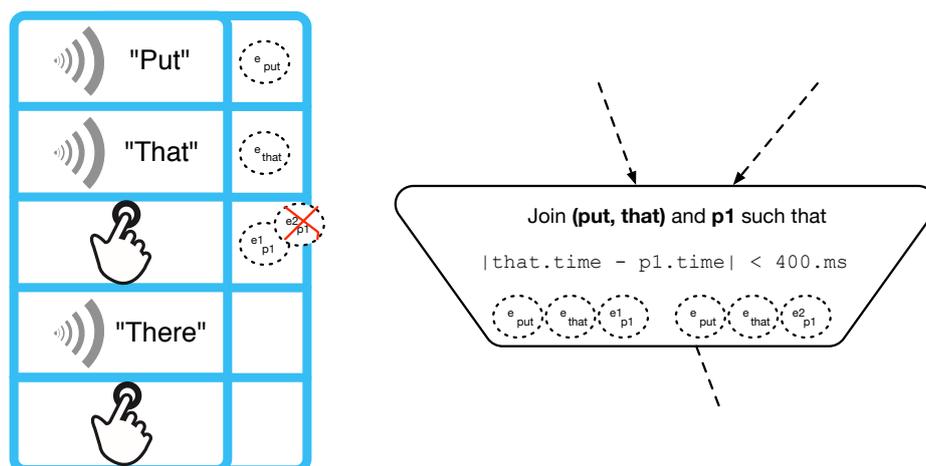


Figure 5.8: Meaning frames can only hold one item (left-hand side) while Rete stores partial overlaps (right-hand side)

5.3.4 Segmentation

Segmentation is the process of extracting the begin and end points of a particular pattern from a continuous input event stream. The Midas language excels at defining complex patterns to discover segmented fusion candidates. The Mudra engine performs the necessary pattern matching to implement segmentation. Segmented candidates can then be verified by more refined rules that include additional application context. The candidates can also be verified by classification algorithms such as DTW or HMM.

Segmentation is necessary to transform continuous streams of low-level data into discrete high-level events. The presented method allows the segmentation of multiple streams, containing information from multiple users at the same time by caching each combination. This seamlessly enables complex functionality such as identification and grouping. Furthermore, it filters noise as many events that do not match the pattern are automatically discarded. However, this noise filtering step is far from trivial as Mudra provides the ability to match non-subsequent events.

We illustrate the process of segmentation in a Rete network by means of Figures 5.9 and 5.10 and Listing 5.16. In this example we extract a 2D Z gesture from a continuous input stream. This input stream delivers events one by one. For performance reasons one needs to keep track of intermediate results. Therefore, many events can be regarded as potential starting points and need to be stored for future analysis. This is illustrated in Figure 5.10 where the first join node (which joins the p1 and p2 conditional elements), uses all input events as a potential starting

point. Then, it stores the combinations that join $p1$ and $p2$ and satisfy the given spatio-temporal conditions. Note that the code inside the join nodes represents Listing 5.16 in a compiled form based on optimisations such as constant folding and sparse conditional constant propagation (see Section 5.5). Figure 5.9 illustrates various candidate combinations when segmenting a Z gesture:

- $(e1, e3)$, $(e2, e4)$ are potential starting points that will not be matched further because of a missing third control point. Note that the combinations $(e2, e7)$ and $(e3, e8)$ are also found. This is intentional because at that point in time $(e8)$, these combinations form potential gestures. However, in this case, these two candidates will be rejected at a later stage.
- Both $(e5, e7)$ and $(e6, e8)$ will match with another fact in the second and third join node. Therefore both are seen as potential gesture candidates and will be filtered by activation policies (Section 4.6.3) and verification processes. As intended, only a single high-level Z gesture fact will be asserted for this example.
- The combination $(e9, 10)$ prepares a starting point for a potential overlapping gesture. This necessary because we are dealing with a continuous input stream which could consist of a valid Z gesture in the near future.

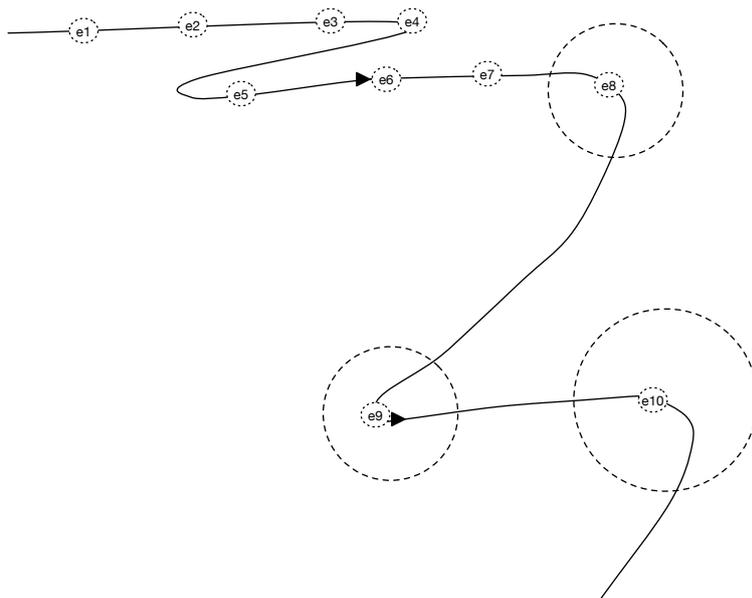


Figure 5.9: The segmentation of a 2D Z gesture

Listing 5.16: Declarative definition of a 2D Z gesture

```
1 rule z
2   c1 = Touch2D
3   c2 = Touch2D
4   c1←beforeF c2
5   c1←translated_nearF c2, 409.px, -13.px, 52.px
6   c3 = Touch2D
7   c2←beforeF c3
8   c1←translated_nearF c3, -14.px, 476.px, 52.px
9   c4 = Touch2D
10  c3←beforeF c4
11  c1←translated_nearF c4, 483.px, 457.px, 76.px
12  c4←withinF c1, 100.ms, 1000.ms
13  assert Z { time_begin ⇒ c1.time, time_end ⇒ c4.time }
14 end
```

Non-Subsequent Event Matching One of the most challenging problems in segmentation is to identify patterns between noise facts. The “noise” facts that do not match the specified conditions should be skipped. This is called non-subsequent event matching. Unfortunately, in many cases it is unclear whether or not to classify a particular fact as noise or not. This depends on contextual information (such as the application state), information from other input sources or the presence of previous and future events from the same input source. The latter case is illustrated in Figures 5.11 to 5.13 which represent three simple subcases when matching a swipe right gesture, namely (1) a few noise events can be discarded (Figure 5.11); (2) a couple of events indicate a potential start of the gesture but can later be discarded as the first pattern is continued (Figure 5.12); and (3) the continuation of the swipe right pattern is noise itself as the subsequent events match a right square bracket gesture (Figure 5.13).

In order to support non-subsequent event matching, it is important to note that a skipped event is not discarded from the fact base as it can form the starting point of another gesture or be part of an intermediate match of combination with other events.

5.3.5 Long Term Reasoning

The ability to recall fusion results from patterns discovered in the past can improve accuracy in the current situation. Long term reasoning is supported in Mudra through the following options:

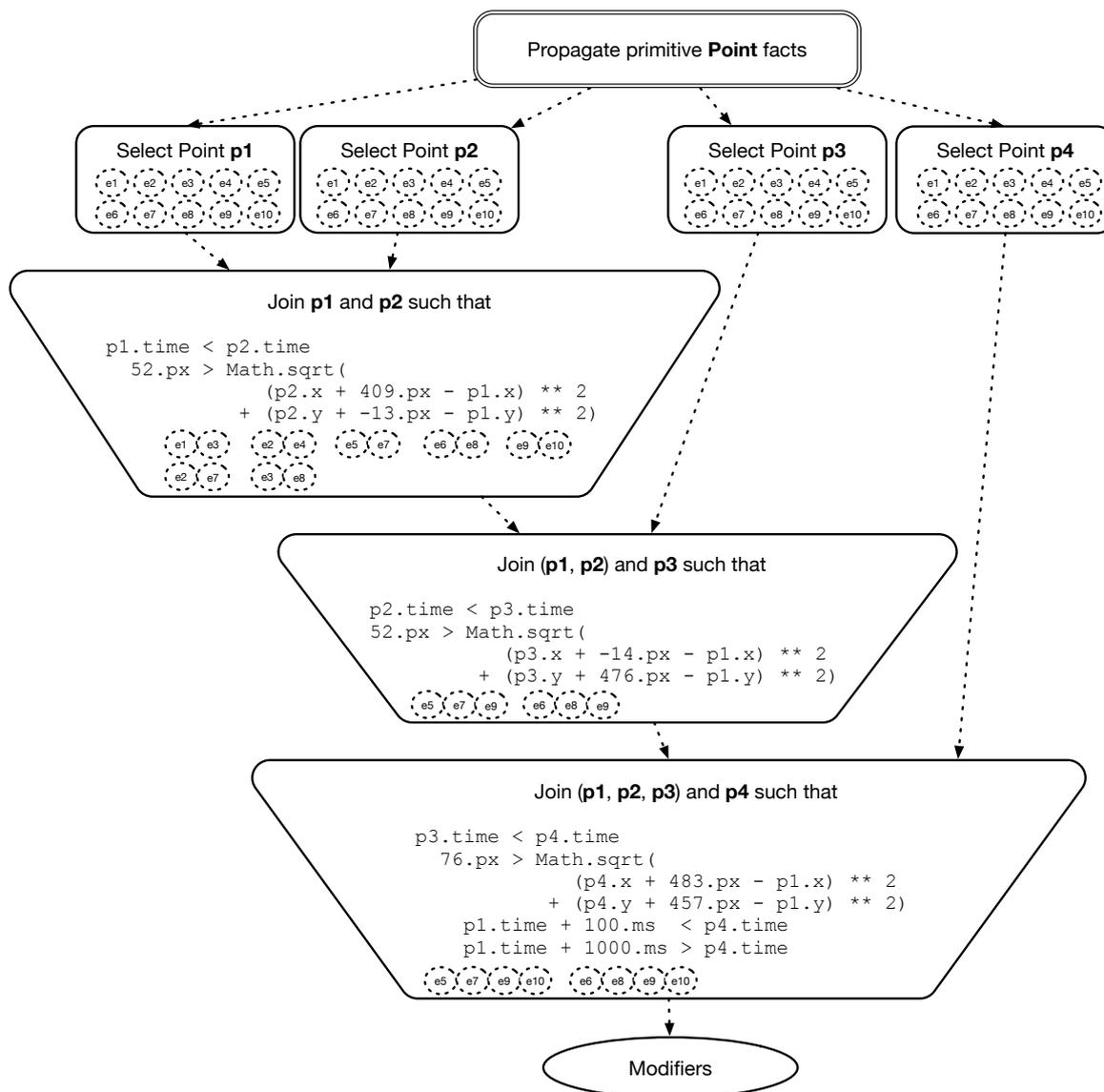


Figure 5.10: State of the Rete network during segmentation

- Developers can define a large timespan window for a few important facts. These facts can originate from the input source or be inferred.
- Developers can employ multiple persistent services, which store a list of facts outside the fact base. They are exempted from data expiration and can be (re-)asserted to the fact base or transmitted to a service at a later point in time.
- Facts can be serialised to and loaded from a text file using Mudra's text-based persistent service.

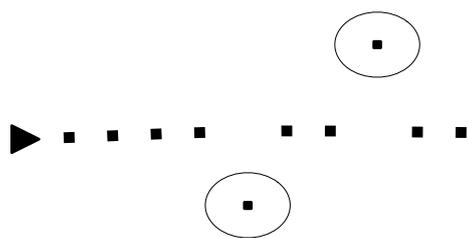


Figure 5.11: A few noise events

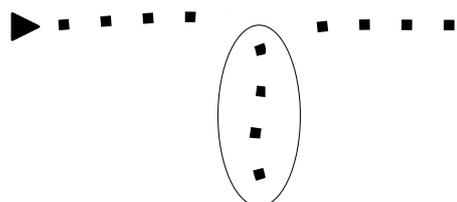


Figure 5.12: Noise subpatterns

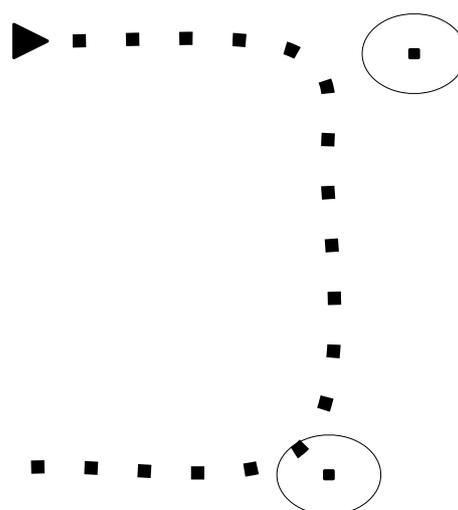


Figure 5.13: Alternative patterns

- Facts can be stored and recalled from an in-memory SQLite relational database. This behaviour resembles the behavior of persistent services but stores data in memory rather than on disk.

These four implementation strategies allow developers to access valuable information for a long period of time.

5.3.6 Concurrent Interaction

To support the concurrent interaction of multiple users, the fusion architecture needs a way to group events for each user. Existing data-stream solutions replicate composition boxes for each user and require developers to manually connect the input and output ports from these boxes for each user.

In Mudra, Rete tokens inherently enable the interaction by multiple users at the same time. As mentioned before, each token represents its own combination of conditional elements that partially match the multimodal description. Therefore, the *put that there* example shown in Listing 5.6 can be satisfied by one user uttering the “*put*” while the other user points his arm to a specific location. If this is not the desired behaviour, developers can specify that each conditional element should originate from the same user. In this example, user identification can be achieved in fusing acoustic source localisation and 3D positioning

based on the camera feed, or additionally, using the voice timbre and face recognition.

5.3.7 Portability, Serialisation and Embeddability

We implemented the Mudra architecture focusing on performance and cross-platform compatibility. Currently it runs on the Linux kernel²³ (compiled using GCC²⁴), Mac OS X²⁵ (LLVM²⁶), Microsoft Windows²⁷ (Visual Studio Compiler²⁸), iOS (running on the ARM architecture)²⁹, Android (ARM, x86, x86_64)³⁰ and other embedded hardware such as the Raspberry Pi³¹.

Mudra is delivered as a background service (i.e. a daemon process) or a library (both static or dynamically linked). Therefore it can easily be embedded in existing programs. Unfortunately, there is no integration or Midas rules in an existing language (such as achieved in LINQ [115]) which means that rules are represented as plain strings. On the positive side, this allows rules and facts to be added at runtime by applications implemented in any programming language.

5.3.8 Runtime Definitions and Device Instantiation

Mudra supports the addition, modification and removal of rules at runtime. This helps the debugging process and allows developers to adapt a running system without downtime. For each added, modified or removed rule, the Rete network is automatically reorganised such that it will deliver optimal performance for future matching.

Dynamically adding devices is trivial as well, since the infrastructure layer can simply produce new templates or facts at any time (Section 4.3.1). Furthermore, dedicated services for these new devices can be instantiated using rules, as shown in Section 5.2.4.

²³Linux: <https://www.linux.com>, and its Ubuntu flavour: <https://www.ubuntu.com>

²⁴GNU GCC: <https://gcc.gnu.org>

²⁵Mac OS X: <https://www.apple.com/osx>

²⁶LLVM: <http://llvm.org>

²⁷Microsoft Windows: <http://windows.microsoft.com>

²⁸Visual Studio: <https://www.visualstudio.com>

²⁹iOS: <https://www.apple.com/ios>

³⁰Android: <https://www.android.com>

³¹Raspberry Pi: <http://www.raspberrypi.org>

5.3.9 Reliability and Scalability

The Mudra architecture provides three simple reliability features, such as:

- APR thread pools³² are used whenever a new service is instantiated. This means that thread creation can be statically defined at startup or be adequately controlled at runtime.
- Mudra provides a set of compiler flags in order to improve sandboxing, such as file I/O, in order to provide increased reliability and safety. Applications can do little harm if its access to the underlying operating system is appropriately restricted, as mentioned by Goldberg et al. [58].
- Rules can be written to dynamically offload facts to multiple Mudra instances or external services. This enables the manual control over the amount of events processed on a single machine and caters to scalability.

However, these features do not provide formal guarantees such as actual real-time processing capabilities. Related research efforts have achieved promising results on parallelisation and scalability with soft real-time guarantees [113, 131, 151] based ideas from the Midas language and Mudra architecture. In order to increase the reliability against network communication delays, Mudra supports out-of-order event input.

Out-of-order Event Input

In practice, it not uncommon to receive events in a different order (i.e. $e1, e3, e2$) than the one that was originally produced by an event source (i.e. $e1, e2, e3$). However, most existing data-level fusion frameworks assume a consistent order of events (see Section 3.4.1).

In Midas and Mudra, out-of-order event input is supported without requiring any special code modification from developers. The Midas semantics dictate that conditional elements are matched regardless of the assertion order. This is because rules rely on explicit time tests, thereby taking the problem of out-of-order input concerns out of the hands of the developer.

Notice however that out-of-order events that are matched with templates whose timespan is more narrow than the time gap between the

³²`apr_thread_pool.h`: http://ci.apache.org/projects/httpd/trunk/doxygen/apr_thread_pool_8h.html

normal order and the out-of-order events may result in a miss. Listing 5.17 demonstrates the data-level fusion of events from the accelerometer (i.e. $a1, a3, a4, a2$) and the gyroscope (i.e. $g1, g2, g3, g4$) in order to improve direction- and motion-sensing. However, when the timespans of gyroscope facts are specified too narrow such that $g4$ expired $g2$, a potential match between $a2$ and $g2$ is lost. In practice, the timespan is rarely set to such low values and it can easily be increased if support for out-of-order event input is needed. Furthermore, as shown in Section 4.5.1, cross-template event expiration can also be used to deal with this issue.

Listing 5.17: Potential missed match due to the combination of out-of-order and event expiration

```

1 rule potentialMissingMatch
2   a = Accelerometer
3   g = Gyroscope
4   a←meetsF g, 2.ms
5 end

```

5.4 Authoring Tools

One of the main goals of our approach is to retain control over recognition results and be able to verify and comprehend them (as opposed to machine learning approaches). In order to further raise the abstraction level and to demonstrate the flexibility of our language as a compilation target, we implemented two authoring tools. An important aspect of these authoring tools is that the resulting rule definitions are provided in a form that is accessible to expert developers. Expert developers can further refine the generated descriptions using hand-coded logic. Therefore our authoring tools focus on generating an *external representation* in the form of the Midas language.

5.4.1 Inferencing and Refining Control Points

As discussed in Section 4.4.6, rules can be used to describe 2D and 3D gestures using the *control points* design pattern. In this section we describe a method to automatically derive these control points from a single representative sample which has been captured by the expert. This form of inferencing is called *one-shot* learning. The current implementation uses a tangent-based calculation where major changes in a small section of the trajectory are stored as potential characteristic control points. The

top m points are then chosen while preserving a good spatio-temporal distribution over the trajectory to ensure that not only distinctive curves but also longer straight lines are used for differentiation.

By defining spatial and temporal constraints between detected control points, a developer has full control over which parts of a gesture should be matched closely and where variation is desired. Figure 5.14 demonstrates inferencing results on four other gestures. Control points can be visualised, translated, rotated and scaled using a graphical interface as shown in Figure 5.15. The output of this application are rules, where additional conditions can be programmed in a textual rule format. Default values for the circular areas surrounding the control points can be easily modified or additional strictness using spatio-temporal operators can be applied.

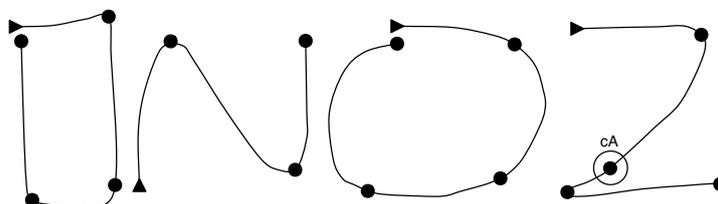


Figure 5.14: Automatically inferred control points

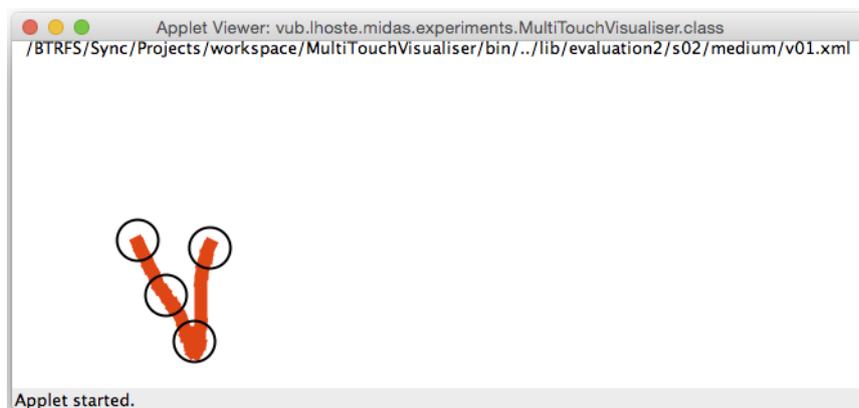


Figure 5.15: Authoring tool for inferencing and refining control points

We opted for a simple but effective solution to automatically infer control points where the result can be visualised and manually refined by the gesture developer. Control points focus on segmentation rather than classification, therefore they are used for high recall. We demonstrate the recall and precision performance on a standard 2D gesture set in Chapter 6 (Section 6.5).

Listing 5.18: Detecting a football kick gesture

```
1 rule kickRightFoot
2   r1 = RelativeJoint { parent == Joint.HIP_RIGHT && child == Joint.FOOT_RIGHT }
3   r1←enterEllipsoid 0.1288, 1.0314, -0.4319, 0.8219, 1.4056, 0.2, 6.1667, 0.0545, 6.2768
4   r2 = RelativeJoint { parent == Joint.HIP_RIGHT && child == Joint.FOOT_RIGHT }
5   r2←enterEllipsoid 0.1436, 0.3249, -1.0984, 0.7740, 0.2629, 0.7391, 0, 0, 0
6   r1←meetsF r2, 1.s
7   assert Kick { time ⇒ r2.time }
8 end
```

5.4.3 Summary

The presented authoring tools highlight the capabilities of Midas and Mudra as a compilation target language. By means of an external representation, developers can understand and modify code output from these tools without much effort. This approach is fundamentally different to many tools that output their knowledge in a data format which cannot embed programming logic. As highlighted by Kadous [82], the comprehensibility of existing spotting and recognition approaches is rather limited. Additionally, it is hard to know when a black box classifier is trained sufficiently in terms of generality or preciseness.

5.5 Compilation and Runtime Model

Mudra provides an extensible and modular architecture. At startup, each layer is initialised in a few milliseconds and becomes ready to receive and publish facts. Midas rules are handled through the Mudra compiler.

5.5.1 Compilation Flow

The Midas language is compiled in multiple steps before being executed by the core engine based on CLIPS (see Section 5.2.3). Throughout this dissertation, the term Midas refers to the latest version of our language specification as presented in Chapter 4. However, the latest version is actually labelled as v2.0, the second major release of our language specification. This specification is an abstraction layer on top of previous versions, as illustrated in Figure 5.17. As argued in a similar way by Sarna-Starosta [140] for Constraint Handling Rules, this form of source-to-source transformation is used to (1) specialise the syntax for the problem

domain, (2) improve static analysis and (3) optimise performance. The compilation process works as follows:

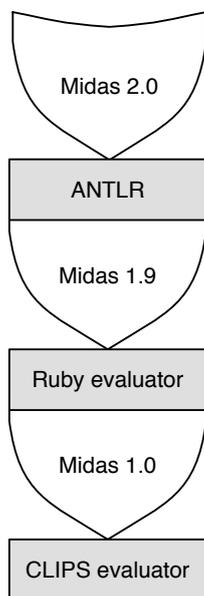


Figure 5.17: Midas compilation flow

1. The Midas 2.0 language is defined by an ANTLR 4 [126] grammar (see Appendix D). A Java program transforms the Midas 2.0 code using the ANTLR 4 libraries into Midas 1.9 program code.
2. Midas 1.9 exploits the metaprogramming capabilities of the Ruby language. Therefore Midas 1.9 code is also legal Ruby 1.9 code and gets evaluated by a standard Ruby interpreter (currently we use JRuby 1.7.12³³). By defining an internal DSL (i.e. creating a new language through metaprogramming), we can rely on many features provided by the host language, including object orientation, functions, blocks, lambdas and scoping mechanisms. The evaluation of Midas 1.9 code returns a single string, containing Midas 1.0 code. This means that all potential evaluation paths of the Midas 1.9 code are executed. By evaluating all paths, we can detect many common mistakes (such as type errors or missing values) and also perform whole program optimisation. These compile-time measures were discussed in Section 4.8. Several optimisation strategies are discussed Section 5.5.3.

³³JRuby 1.7.12: <http://jruby.org>

3. Midas 1.0 code is CLIPS code with additional functionality. This additional code is responsible for initialising and communicating to various services.

In the next sections we provide additional details about the goal of each compilation step.

5.5.2 Midas 2.0 ANTLR Compiler

The main objective of the Midas 2.0 abstractions is to provide syntactic sugar on top of the Midas 1.9 language. As Midas 1.9 relies on the metaprogramming capabilities of Ruby, several syntactic operators of Midas 2.0 cannot be supported in Ruby. Therefore, the Midas 2.0 ANTLR compiler translates an `attempt` into a regular function and a `←` into a dot operator. It also wraps constructs that cannot be captured by the metaprogramming capacities of Ruby, such as index access on arrays (i.e. `array[0]`) and certain infix operators (i.e. `(5 < 2)`). Finally it allows us to prohibit invalid semantic combinations through syntax, such as the use of attempts inside a function (as presented in Chapter 4). Midas 2.0 also provides a `require` operator that allows one to load files from different files.

5.5.3 Midas 1.9 Ruby Compiler

Midas 1.9 is implemented as an internal DSL. This concept is popular in the Ruby and Lisp community and has been exploited by the well-known Ruby on Rails (RoR) framework [153]. However, in contrast to RoR where the web pages are served by the Ruby interpreter, the execution of a Midas 1.9 program results in a single string containing Midas 1.0 (aka CLIPS) code. Therefore, Midas 1.9 compiles its programs into another language using metaprogramming.

Most of the multimodal abstractions presented in this dissertation are made possible through the Midas 1.9 compilation step. This includes, the ability to compose and inherit modules, to alternate between conditions and modifiers, to embed intermediate progress notifications, to allow various activation flags, to provide compile-time errors and to recorder conditions for optimisation purposes.

By relying on the Ruby interpreter we can program a compiler to a high-level language and rely on many existing features. It allows for rapid adaptations and extensions of the Midas language but due to the

extent of metaprogramming, the code is however quite complex. In what follows, we demonstrate the inlining, unlinking and condition reordering capabilities of the Midas language.

Constant folding

The Midas 1.9 compiler performs constant folding on as many values as possible by evaluating primitive operations in the Ruby environment. For example, the expression $(5 + 2)$ is always 7, therefore it can be *folded* at compile time. Several complex mathematical operations, such as `Space3D<-enterEllipsoid` as shown in Listing 5.19 can also be partially computed at compile-time. Listing 5.19 demonstrates a rule which specifies the 3D movement of the right foot in order to recognise a football kick. The highlighted fragments (lines 9 to 17) are operations that are folded by the compiler by using the instantiated values of lines 29 and 32. Operations such as cosine, sine and raising to the power are therefore automatically calculated only once and not for every incoming event. Constant folding has a big impact on runtime performance because of the many spatial and temporal relations that are required in multimodal fusion in the face of thousands of events that are entering the system.

Sparse Conditional Constant Propagation

The Midas compiler supports sparse conditional constant propagation by traversing all execution paths. For example, when slot *s1* is used in a complex mathematical computation and it is later unified with another slot *s2*, for which a value is known at compile-time, the inlining of 1 with that value occurs. Additionally, tests that can be verified at compile-time output a warning and are omitted from the compiled code. These examples are illustrated by Listing 5.20, where the `leftF` attempt in the `deepInline` rule is always true and therefore becomes irrelevant at runtime.

Listing 5.19: Inlining mathematical operations of a football kick

```

1 module Space3D
2   x, y, z
3   previous_x, previous_y, previous_z
4   # Arguments: ellipsoid, radius, rotation and a point
5   function self.distEllipsoid(cx, cy, cz, rx, ry, rz, degx, degy, degz, px, py, pz)
6     diff_x = px - cx
7     diff_y = py - cy
8     diff_z = pz - cz
9     x = (Math.cos(-degz) * diff_x) + ((-Math.sin(-degz)) * diff_y)
10    y = (Math.sin(-degz) * diff_x) + (Math.cos(-degz) * diff_y)
11    x2 = (Math.cos(-degy) * x) + (Math.sin(-degy) * diff_z)
12    z = ((-Math.sin(-degy)) * x) + (Math.cos(-degy) * diff_z)
13    y2 = (Math.cos(-degz) * y) + ((-Math.sin(-degz)) * z)
14    z2 = (Math.sin(-degz) * y) + (Math.cos(-degz) * z)
15    result = ((x2 ** 2) / ((rx / 2) ** 2)) +
16      ((y2 ** 2) / ((ry / 2) ** 2)) +
17      ((z2 ** 2) / ((rz / 2) ** 2))
18  end
19  attempt enterEllipsoid(cx, cy, cz, rx, ry, rz, degx, degy, degz)
20    Space3D.distEllipsoid(cx, cy, cz, rx, ry, rz, degx, degy, degz, x, y, z) <= 1
21    Space3D.distEllipsoid(cx, cy, cz, rx, ry, rz, degx, degy, degz,
22      previous_x, previous_y, previous_z) > 1
23  end
24 end
25
26 rule kickRightFoot
27   r1 = RelativeJoint { parent == Joint.HIP_RIGHT && child == Joint.FOOT_RIGHT }
28   r1←enterEllipsoid 0.1288, 1.0314, -0.4319,
29     0.8219, 1.4056, 0.2, 6.1667, 0.0545, 6.2768
30   r2 = RelativeJoint { parent == Joint.HIP_RIGHT && child == Joint.FOOT_RIGHT }
31   r2←enterEllipsoid 0.1435, 0.3249, -1.0984,
32     0.7740, 0.2629, 0.7391, 0, 0, 0
33   r1←meetsF r2, 1.s
34   assert Kick { time ⇒ r2.time }
35 end

```

Listing 5.20: Sparse conditional constant propagation

```

1 module Space2D
2   attempt leftF(p)
3     x < p.x
4   end
5 end
6
7 rule constantPropagation
8   a = Point
9   a.x == 0
10  b = Point
11  b.x == 3
12  a←leftF(b)
13  assert LeftOf { x1 ⇒ a.x, x2 ⇒ b.x }
14 end

```

Unrolling

All the language abstractions presented in Chapter 4, such as attempts and functions, are unrolled to their most primitive form. This means that all computation is translated into primitives as defined in the formal grammar (Section 4.1.1). Therefore, attempts such as `translated_nearF` and `beforeF` are translated into a combination of simple mathematical operations (see Section 4.4.1 and 4.4.2).

Unlinking

Attempts, functions or rules that are unused or have no external effect (i.e. without modifiers) are eliminated during compilation and will therefore not be computed at runtime.

Condition Reordering

The order of conditions in the Midas language can be changed without affecting the semantics of the rule. As mentioned by Khandkar et al. [91], the order of conditions can significantly affect the overall performance of a Rete system. Our compiler exploits this fact by reordering conditions. Concretely, tests construct filter combinations (i.e. tokens in the Rete network) and therefore reduce the time and space complexity of the computation. Tests should therefore be executed as soon as possible. However, most tests rely on values that are bound by conditional elements at runtime. To optimise performance, tests should be reordered as close to the conditional element that binds the required arguments. Thanks to unrolling, the granularity of the conditions that can be reordered is quite refined. Note that this is similar to query optimisation techniques as performed by database engines.

The condition reordering is performed using a stable sort. Therefore, tests that require values from the same conditional element and conditional elements without annotated input rate retain their order. This allows the expert developer to stay in control of the optimisations.

5.5.4 Midas 1.0 Core Engine

The matching of facts to the conditional elements of rules is performed by a modified CLIPS interpreter. This highly optimised single threaded interpreter is written in C and uses a LISP-based syntax to define rules. CLIPS was intended as an expert system shell based on production

rules and has been modified into a reactive multimodal core engine. Details on the CLIPS language and implementation are discussed by Giarratano et al. [57]. G. Riley actively maintains CLIPS and we cherry-pick³⁴ these upstream changes.

5.6 Conclusion

We presented Mudra, a unified multimodal interaction framework for processing low-level data streams as well as high-level semantic inferences. Our approach is centred on a fact base that is populated with multimodal input data coming from various devices. Various recognition and multimodal fusion algorithms can actively react to changes in the fact base and enrich it with their own interpretation.

Mudra's shared bus architecture and efficient interpretation of declarative Midas rules blurs the distinction between data-, feature- and decision-level fusion. This is made possible by three main ingredients: an event broker, a template-local event expiration mechanism and an efficient execution engine which uses the Rete algorithm.

Through the use of the Rete algorithm we are able to provide high-level language abstractions that result in a form of implicit data flow programming. In contrast to existing data-stream approaches, the chaining of components is done in an automated fashion with much more expressiveness to tackle partially overlapping matches, segmentation, online gestures and many other criteria. With respect to a recent categorisation of toolkits performed by Cuenca et al. [31], we position our work as a token-based approach. However, unlike existing token-based solutions, Midas does not require the developer to exhaustively list all potential state transitions.

The Midas language is a fifth generation programming language, which until now, were only supported by decision-level architectures. However, *with declarative rules and Rete we obtain high-level abstractions with the ability to also process a large number of low-level events*. Furthermore, our language and execution engine are extensible because they enable user-defined conditions on raw input events and they allow developers to integrate existing publish/subscribe services. Finally, our declarative language facilitates compiler-based optimisations and seamless support for multi-core and many-core extensions [113, 151].

³⁴Git cherry-pick: <http://git-scm.com/docs/git-cherry-pick>

The Mudra infrastructure supports a wide variety of data formats and communication protocols, which makes the integration of existing solutions quite simple. Added services can be coordinated using rules to respond dynamically to changing needs at runtime based on various patterns in the input data. Mudra provides several powerful features which are not present in other architectures, including the ability to process out-of-order events, to process non-subsequent events and to deal with partial overlap and segmentation. The combination of Midas and Mudra also forms an adequate target platform for authoring tools that want to generate a high-level external representation.

In the next chapter, we argue that our presented solution improves upon existing work according to various criteria. We also show how well Midas and Mudra perform when applied to real-world scenarios.

6

Midas & Mudra at Work

In this chapter we evaluate our approach based on 30 criteria defined in Chapter 2. Besides this argumentation-based assessment, we perform an in-depth comparison between Midas and two state-of-the-art domain-specific programming languages. Afterwards, we present a number of demonstrators of Midas and Mudra in real-world settings. These demonstrators were exhibited at international conferences, science education centres and public events.

6.1 Midas and Mudra: A Qualitative Evaluation

In the following section we discuss the scores that are assigned for each criterion within the four categories defined in Section 2.3: Language Features, Multimodal Processing, Multimodal Specification and Accessibility and Tooling. Figure 6.1 provides an indicative classification of Midas and Mudra in contrast to existing multimodal solutions, including QuickSet [27, 74, 164], MIML [102], PATE [129], OpenInterface [103, 142], Squidy [97], HephaisTK [39, 40, 42, 43] and DynaMo [5, 6] (Section 3.2). The scores were rated on a scale from 0 (not supported) to 5 (fully supported). The classifications of existing approaches are visualised through a box-and-whisker plot, and the classification of Midas and Mudra is visualised on the same plot in a diamond shape.

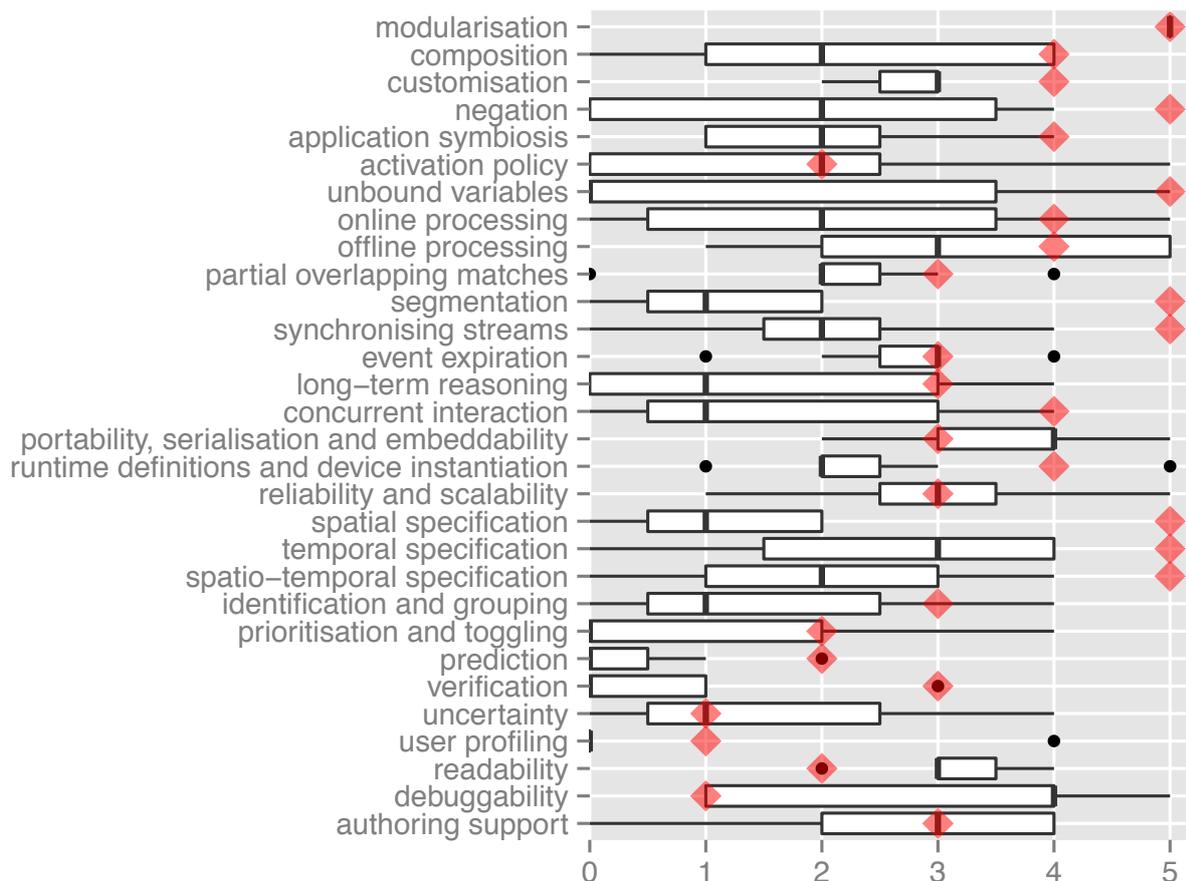


Figure 6.1: Indicative classification of Midas and Mudra (score visualised as a diamond) in comparison to existing approaches (visualised as box plot)

6.1.1 Language Features

In the next subsections we analyse the capabilities of the Midas multimodal language based on the seven language criteria defined in Section 2.3.1. For clarity, the score of our approach is listed in brackets for each criterion.

Modularisation (5) The first criterion of the language features is *modularisation*. As described in Section 2.3, modularisation focuses on the separation of concerns and allows for a reduced effort when gradually increasing the number of fusion processes. In Midas, each fusion process can be described in a rule (Section 4.3.1), thereby supporting additional fusion processes easily. Furthermore, developers can easily group functionality into attempts (Section 4.2.2, functions 4.2.2 and modules 4.2.1). The former language construct, namely an attempt, allows developers to group several conditions into a single construct, which can be reused by any rule. The second construct, a function, provides an abstraction

for computation without side effects. These side-effect free computations are typically used to calculate spatial or temporal distances. The latter module construct, groups attempts, functions and slot definitions to offer a specialised functionality in a dedicated bundle.

Composition (4) In Midas, multimodal interaction pattern descriptions can be composed using high-level facts (Section 4.3.1), attempts and computed facts (Section 4.3.1) as well as module inheritance (Section 4.3.2). We argue that *attempts* solve a fundamental problem with data entanglement found in existing approaches, which typically compose descriptions by exchanging high-level events. Additionally, Midas provides module inheritance which allows developers to easily embed functionality in templates (Sections 4.2.1 and 4.3.2). This enables access to attempts and functions through their relevant conditional element (Section 4.2.2). Midas' composition constructs and their use in rules are also verified at compile time (Section 4.8).

Customisation and Extensibility (4) Adapting functionality in Midas is relatively easy based on four features: (1) rules decouple the order of conditions from the semantics of the multimodal description and their temporal relation; (2) attempts allow parameters to solve the specialisation problem (Section 4.3.1); (3) Templates include modules to provide additional functionality and can be extended multiple times, even across files, such that customised functionality can be grouped; (4) conditional elements, attempts and functions are user-defined and can be expressed in the Midas language without having to escape to the host programming language.

Negation (5) In Midas, negated patterns can be of arbitrary complexity and be nested (Section 4.3.4). Midas also supports the inexistence of future events. However, this requires the manual addition of a `wait` condition.

Application Symbiosis (4) Midas offers various ways to integrate application state in the fusion engine (Section 4.3.5). It consists of traditional constructs such as callbacks and topic-based publish/subscribe APIs, but through the use of rules, content-based subscriptions are also an option. However, the most refined integration of application state is provided by shadow facts (Section 4.6.1). Shadow facts represent

application objects in the fact base and can therefore be used to express advanced conditions in terms of rules. Furthermore, the state of an application object is automatically synchronised with its respective shadow facts and visa-versa.

Activation Policy (2) To control the activation of multimodal rules, Midas supports simple activation flags, including *one-shot*, *shoot-and-continue* and *hold* (Section 4.6.3). Additionally, the activation of rules can be programmed by manually bookkeeping facts as shown in Section 4.6.3. Although bookkeeping facts can be used to express a refined activation policy, they can become quite complex to manage.

Unbound Variables and Unification (5) Midas makes extensive use of variables that are bound to runtime information. The most prominent example is that slot values of a condition element are modelled as unbound variables. Complex runtime variable bindings, such as `new_x = t1.x + t2.x + 5.px`, are supported as well. Explicitly unbound variables are denoted with a question mark, which is similar to QuickSet [27] and PATE [129] (Section 4.7). However, in our experience, explicitly unbound variables have been a source for bugs (Section 4.7) while references to slots provide compile-time feedback (Section 4.8). When two unbound variables should be equal (i.e. `p1.x == p2.x`), unification is used (Section 4.7) and is optimised by the underlying CLIPS [57] Rete [54] engine (Section 5.2.3).

6.1.2 Multimodal Processing

We use the eleven multimodal processing criteria defined in Section 2.3.2 to analyse the capabilities of the Mudra unified multimodal interaction architecture.

Online Processing (4) Whenever conditions in a rule are satisfied, consecutive modifiers are executed. By allowing arbitrary conditions with a fine-grained specification, Midas and Mudra support online processing and feedback (Section 5.3.1). To enable online user feedback, Midas allows an alternation of conditions and modifiers inside a single rule (Section 4.6.2). Additionally, intermediate progress notifications can be enabled (Section 4.6.3).

Offline Processing (4) Midas and Mudra provide abstractions and architectural support to ease offline processing. For example, the recognition of multi-touch gestures can be delayed until a finger lifts or even until all fingers are removed from the surface. Additionally, the `wait` construct allows for a relative time delay (Section 4.1.1) and the `async` abstraction eases the offline verification process (Sections 4.5.3 and 4.5.4).

Partially Overlapping Matches (3) The Mudra engine inherently supports partially overlapping matches and even optimises them through the Rete algorithm (Section 5.3.3). Unfortunately, in contrast to approaches such as Proton [94, 95], developers are not informed when conditions overlap.

Segmentation (5) Segmentation is an important aspect of multimodal fusion. However, existing multimodal frameworks have largely neglect this concern (Section 3.2, Section 3.5 and Figure 6.1). The combination of Midas and Mudra hides the accidental complexity of segmentation when describing multimodal interaction patterns. For example, complex 2D and 3D gestures can be defined in rules with automated segmentation (Section 4.4.6). Furthermore, Mudra supports non-subsequent event matching, thereby automatically discarding noisy events during the segmentation process (Section 5.3.4).

Synchronising Streams (5) Mudra provides a number of abstractions to synchronise multiple input streams. Firstly, temporal relations can be expressed in Midas, which allows an arbitrary ordering of input events (including out-of-order event input, Section 5.3.9). Secondly, these temporal relations make use of a `time` slot value time stamped by the input source, which discards accidental complexities with network latency (Section 4.2.1). Thirdly, Mudra uses non-subsequent event matching to discard excess events when fusing streams with different input frequencies (Section 4.5.1). Fourthly, automated fact expiration is handled per template, which preserves information for streams with a higher input latency (Section 5.2.2). Lastly, cross-template event expiration can be described in rules to manually synchronise expiration across streams (Section 4.5.1).

Event Expiration (3) Mudra provides a template-local relative time expiration mechanism (Section 5.2.2). Additionally, events can be retrac-

ted in Midas rules as well, which allows for complex pattern-based event expiration (Section 4.5.1).

Long-Term Reasoning (3) Long-term reasoning can be obtained in multiple ways as described in Section 5.3.5. This includes memory (DRAM) and disk (hard drive) persistency with manual recall operators.

Concurrent Interaction (4) Mudra matches any combination of facts to conditions specified in rules. Therefore, if two users perform the same 2D gesture at the same time, the rule will activate twice. To separate the actions of multiple users, identity unification can be used (Section 4.4.5).

Portability, Serialisation and Embeddability (3) Mudra has been ported to multiple platforms and architectures as shown in Section 5.3.7. Furthermore, Mudra is able to interpret serialised data and programming code in order to offload data to services and enable code mobility (Sections 4.5.2, 5.2.4 and 5.2.4). Mudra is embedded as a library or runs as a background daemon (Section 5.3.7).

Runtime Definitions and Device Instantiation (4) Mudra supports the runtime addition of rules (Section 5.3.8), templates, devices and services (Section 5.2.4).

Reliability and Scalability (4) Our architecture provides a number reliability features, such as thread pooling, sandboxing and offloading to external nodes. However, it does not have built-in support to limit the frequency of an input stream. Additional efforts, such as PARTE [113,131] and Cloud-PARTE [151], provide in-place alternatives for the core engine and achieve promising results to provide soft real-time guarantees and scalability up to 64 cores and 8 machines (Section 5.3.9).

6.1.3 Multimodal Disambiguation

In this section we assess the disambiguation criteria of Midas and Mudra.

Spatial Specification (5) Midas and Mudra support arbitrary complex spatial specifications by leveraging user-defined attempts and functions. This user-defined spatial functionality can, for example, express

distance between humans and objects, as well as orientation and movement in various “proxemic dimensions” [112]. Additionally, we provide a number of built-in 2D and 3D spatial operators and custom units (Section 4.4.1).

Temporal Specification (5) Midas and Mudra support arbitrary complex temporal specifications by leveraging user-defined attempts and functions. Additionally, we embed Allen’s temporal operators [1] and custom units (Section 4.4.2) to bootstrap temporal specifications.

Spatio-temporal Specification (5) The spatio-temporal specification and nesting of spatial and temporal operators is supported by the framework (Sections 4.4.3 and 4.4.4).

Identification and Grouping (3) Instantiation-, unification- and scope-based identity features can be used to identify and group facts (Section 4.4.5). Additionally, Mudra matches all combinations such that uncertain identification or grouping can be decided when adequate information becomes available.

Prioritisation and Toggling (2) Midas offers rule-specific prioritisation and bookkeeping facts to enable prioritisation and toggling (Section 4.6.3). However, advanced conflict resolution abstractions are not provided.

Prediction (2) Future information can lead to a completely different interpretation of the input data. In Midas, each conditional element delays the decision process until a fact matches its description, thereby allowing disambiguation based on future information. Additionally, Midas provides a `wait` construct to wait for (non-)existence of future events based on a timeout. However, predictions based on heuristics are not supported by the current implementation of our framework.

Verification (3) Mudra embeds a number of classifiers to verify candidates. These are accessible through services (Section 4.5.2) and an `async` language construct (Section 4.5.4). External services can also be integrated to verify results (Sections 5.2.4 and 5.2.4).

Uncertainty (1) When the validity of information is uncertain, facts can be extended with a probability slot. However, it is up to the developers to deal with this and define what it means to fuse two uncertain facts.

User Profiling (1) Midas and Mudra do not provide user profiling tools. A persistency service to access and store historical data is supported which allows developers to implement user profiling as an extension (Section 5.2.4).

6.1.4 Accessibility and Tooling

In this final category we evaluate the accessibility and tooling criteria of Midas and Mudra.

Readability (2) Kammer [84] measured a low readability of Midas 1.0. Since that work, we focused on raising the level of abstraction and readability of Midas. This gave rise to the flavour of Midas presented in this dissertation (Midas 2.0) which is less syntax intensive compared to Midas 1.0. However, Midas 2.0 and the Mudra architecture remain powerful tools for experts on which authoring tools for end users can be built.

Debuggability (1) Midas provides a number of compile-time guarantees to prevent common accidental mistakes in multimodal descriptions (Section 4.8). Additionally, input events can easily be stored on disk such that they can be replayed for debugging or benchmarking purposes without having to modify any rule (Section 4.3.7).

Authoring Support (3) Besides being a high-level multimodal programming language, Midas positions itself as a representation language for external tooling (Section 4.3.3). We demonstrated this capability with two authoring tools, namely the automated inferencing and manual refinement of control points (Section 5.4.1) and an authoring tool for full-body gestures (Section 5.4.2).

6.1.5 Conclusion

As observed in Figure 6.1, Midas improves on state-of-the-art in multiple criteria, such as composition, customisation, application symbiosis,

unbound variables, segmentation, synchronising streams, concurrent interaction, spatial specification, temporal specification and spatio-temporal specification. Cirelli et al. [26] recently performed a survey of multi-touch frameworks and analysed Midas in a similar positive manner on 14 criteria (see Appendix G for a comparison between our and their criteria). However, certain criteria, such as activation prioritisation, prediction and user profiling are still open for future work. In the next sections we highlight Midas and Mudra at work by comparing them to two existing high-level domain-specific programming languages and demonstrating several real-world applications.

6.2 Comparing Software Engineering Abstractions for Multimodal Interaction

We perform an in-depth comparison between Midas and a representative gesture language (Proton) and decision-level multimodal language (SMUIML).

6.2.1 Comparing the Data-Level Language Abstractions of Midas and Proton

In Section 3.3.1 we observed nine concerns with respect to the declarative implementation of two fingers moving towards each other using Proton [94] code (Listing 3.5, duplicated here as Listing 6.1). In this section we reflect on these concerns using a Midas implementation.

Listing 6.1: A Proton implementation of two fingers moving towards each other

```

1  $D_1^{L:O} M_1^{L:O} * D_2^{R:O} (M_1^{L:O} | M_2^{R:O}) *$ 
2  $(M_1^{L:E} (M_1^{L:E} | M_2^{R:O}) * M_2^{R:W} M_2^{R:W} (M_1^{L:O} | M_2^{R:W}) * M_1^{L:E})$ 
3  $(M_1^{L:O|E} | M_2^{R:O|W}) * (U_1^{L:O|E} M_2^{R:O|W} * U_2^{R:O|W} | U_2^{R:O|W} M_1^{L:O|E} * U_1^{L:O|E})$ 

```

The Midas implementation of the Proton abstractions is presented in Listing 6.2. These abstractions allow us to define the gesture in a few lines of code as shown in Listing 6.3.

Listing 6.2: A Midas implementation of Proton's built-in features

```

1 module SplitScreen
2   attempt leftSide
3     moveRight.x_begin < 0.5.display_x           # On the left side of the surface
4   end
5   attempt rightSide
6     no { leftSide }
7   end
8 end
9 module UpDown
10  attempt downBeforeF(f)
11    u = Touch2D { finger == f.finger && state == Touch2D.DOWN }
12    u←beforeF f
13  end
14  attempt upAfterF(f)
15    u = Touch2D { finger == f.finger && state == Touch2D.UP }
16    u←afterF f
17  end
18 end
19 module Touch2DSpatioTemporal
20  include Space2DInterval, TimeInterval
21  include SplitScreen
22  finger
23  attempt self←boundingBox(x_diff, y_diff)
24    no {
25      n = Touch2D
26      n.finger == finger
27      n←after time_begin
28      n←before time_end
29      x_begin < n.x
30      x_end > n.x
31      y_begin < n.y
32      y_end > n.y }
33  end
34  attempt self←eastStroke(min_x)
35    movingTouchStroke min_x, min_x * 2, -3.px, 3.px
36  end
37  attempt self←westStroke(min_x)
38    movingTouchStroke -min_x, -min_x * 2, -3.px, 3.px
39  end
40  attempt self←movingTouchStroke(min_x, max_x, min_y, max_y)
41    t1 = Touch2D { state == Touch2D.DOWN }
42    t2 = Touch2D { state == Touch2D.UP && finger == t1.finger }
43    t1←beforeF t2
44    t2.x > t1.x + min_x
45    t2.x < t1.x + max_x
46    t2.y > t1.y + min_y
47    t2.y < t1.y + max_y
48    st = { time_begin ⇒ t1.time, time_end ⇒ t2.time,
49          x_begin ⇒ t1.x, x_end ⇒ t2.x,
50          y_begin ⇒ t1.y, y_end ⇒ t2.y } with [Touch2DSpatioTemporal]
51    st←boundingBox 10.px, 5.px
52    return st
53  end
54 end

```

Listing 6.3: A Midas implementation of two fingers moving towards each other

```

1 rule towards
2   e = Touch2DSpatioTemporal←eastStroke 10.px
3   w = Touch2DSpatioTemporal←westStroke 10.px
4   e←equalF w, 500.ms
5   e←leftSide
6   w←rightSide
7   e←align_beginF w, 30.px, 5.px
8 end

```

In the following enumeration, we discuss Midas' solution to nine concerns raised in Section 3.3.1.

1. The excessive repetition of attributes in the superscript (E) and the numbers in the subscript (1) is abstracted in Midas by bundling each stroke (line 2 in Listing 6.3) in an attempt (line 34 in Listing 6.2). Additionally, *unification* is used to group events from a particular finger through the `finger` slot (line 42 in Listing 6.2). Therefore, Midas provides better support for *modularisation* and *composition*.
2. Midas enables the concurrent execution of other gestures by relying on *unbound variables and unification* instead of static numbers to *group* events from one or more fingers (lines 42 and 26 in Listing 6.2).
3. In Midas, the order of conditions does not dictate the sequence of events. Additionally, the flexibility of *temporal operators* allows either finger to touch the surface first. They are aligned in time with an approximate interval defined by line 4 in Listing 6.3.
4. The *spatial specification* in Midas caters a parametric *customisation*. Line 2 in Listing 6.3 defines a spatial interval of 10 pixels which is specific for this gesture. Proton does not provide abstractions to refine a spatial specification of a gesture. Additionally, curvy gestures can be expressed in Midas using the control points design pattern (Section 4.4.6).
5. Midas allows an arbitrary mix of *spatio-temporal operators* such that, for instance, cardinal direction can be calculated over a large time window. This is in contrast to Proton where cardinal directions are calculated per frame. Additionally, Midas' non-subsequent event matching filters noisy events which would cause false negatives in Proton.

6. Proton uses preprocessors to transform event input into cardinal directions. However, these are global for all gestures and are unaware of additional gesture-specific constraints. To extend Proton with a pinch feature that specifies that no more than three fingers should touch the screen, a new attribute needs to be added to the interpreter. In contrast to this, Mudra uses rules to group conditions. Therefore, additional conditions to existing rules can be appended to refine the gesture implementation. Additionally, Mudra enables the *concurrent interaction* of multiple fingers by default. Thus, the implementation given in Listing 6.3 already supports multiple groups of fingers moving towards each other. To specify that this gesture should only activate when no other fingers touch the surface, *negation* can be used.
7. We encourage the development of primitive features within the Midas programming language. Midas hides several accidental complexities, such as segmentation and partially overlapping matching and allows developers to *customise and extend* existing definitions. Additionally, modules bundle concern-specific features and resolve the problem of conflicting names of functions or attempts when the codebase grows.
8. Our approach is able to replicate Proton’s split screen behaviour without relying on interpreter extensions, as shown in Listing 6.2 (lines 1 to 8) and used in lines 5 and 6 in Listing 6.3. Additionally, Midas supports an advanced *application symbiosis* to group gestures based on the state of the application or surrounding GUI objects. Furthermore, relative spatial boundaries can also be used to group gestures to local regions, as defined on line 7. Line 7 also refines the gesture such that two fingers should be located near each other, whereas in the Proton definition, a finger moving to the east at the top of the surface and a finger moving to the west at the bottom of the surface would also activate the gesture.
9. Midas code is Unicode-based (UTF-8 [165]) and can therefore be edited with mainstream text editors.

To conclude, we solve a number of major issues observed in the code presented by Kin et al. [94]. Besides these issues, our approach also scales outside the domain of multi-touch gesture recognition, thereby supporting data-level fusion tasks of other modalities. In the next section we compare

Midas to SMUIML [38], a state-of-the-art decision-level multimodal fusion language.

6.2.2 Comparing the Decision-Level Language Abstractions of Midas and SMUIML

HephaistTK [38, 39] offers a high-level Synchronized Multimodal User Interfaces Markup Language (SMUIML) to define decision-level fusion. In Listing 6.4, the general description of the SMUIML language is outlined. It consists of four main sections. The first section, labelled as `<recognizers>` (lines 4 to 6), is used to bind input modalities to external recognisers. This is similar to our approach, as defined in Section 4.5.2. However, HephaistTK declares this binding in a static manner and thereby does not support the dynamic creation of recognisers at runtime. The following `<triggers>` and `<actions>` sections are used to specify and convert data formats from input modalities. The final `<dialog>` section defines the multimodal state transitions. Each state is defined by `<context>` elements mapping triggers to transitions.

Listing 6.4: A generic SMUIML description

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <smuiml>
3   <integration_description client="client_app">
4     <recognizers>
5       <!-- Bind a modality to a recogniser -->
6     </recognizers>
7     <triggers>
8       <!-- Groups information from recognisers -->
9     </triggers>
10    <actions>
11      <!-- Group information for applications -->
12    </actions>
13    <dialog>
14      <!-- Define multimodal state transitions -->
15    </dialog>
16  </integration_description>
17 </smuiml>

```

Listing 6.5 provides a concrete SMUIML implementation used in a multimodal paint application [38]. The full implementation is provided in Appendix H.1. We use this example to compare the SMUIML decision-level fusion abstractions to our own unified solution.

The first three sections of SMUIML (`<recognizers>`, `<triggers>` and `<actions>`) can be defined in a similar way by means of attempts and functions in Midas (Listing 6.6). For example, the trigger `tools_one_hand`

on lines 23 to 25 in Listing 6.5 can be translated to lines 12 to 14 in Listing 6.6. SMUIML converts input data into a format compatible with existing recognisers, as shown on lines 6 to 9 in Listing 6.5. This is supported in Midas as well, as shown on lines 1 to 4 in Listing 6.6.

The fourth section of SMUIML (`<dialog>`) represents the dialogue management capabilities of HephaisTK. In this section, a number of conditions are enumerated and bound by a single temporal relation (lines 44 to 53 in Listing 6.5). SMUIML supports four temporal relations, designed to enable three out of four CARE properties [38]. The fourth CARE property, assignment, is not supported in HephaisTK due to the lack of support for negation. Midas provides negation and can therefore express the assignment property of CARE. A final feature of the `<dialog>` section is the ability to manage different contexts. Line 35 in Listing 6.5 defines the start context and when a state transfers, the context is modified as illustrated on line 39. In Midas, contextual information can be handled by relying on an application symbiosis (Section 4.3.5) or by manually bookkeeping facts (Section 4.6.3). In Listing 6.6, we opted for bookkeeping facts to keep the implementation as close as possible to the original multimodal description in SMUIML. As observed on lines 16 and 18 in Listing 6.6, contexts and transitions between contexts can be expressed in Midas. In Listing 40 (Appendix H), we provide an implementation of the SMUIML example using application symbiosis. We prefer this second implementation because it automatically synchronises context state between the multimodal descriptions and the application. Without this automated synchronisation, code to manage context is replicated and therefore becomes prone to errors.

With this analysis we conclude that the abstractions in Midas supersede the SMUIML abstractions and are therefore adequate to express advanced decision-level fusion.

Listing 6.5: A SMUIML implementation of a multimodal paint application

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <smuiml>
3   <integration_description client="xpaint_client">
4     <recognizers>
5       (...)
6       <recognizer name="reactivision" modality="reactivision">
7         <variable name="posx" value="xpos"/>
8         (...)
9       </recognizer>
10      <recognizer name="phidgetrfid" modality="rfid">
11        (...)
12        <translate_value from="2342111" to="filled circle"/>
13        (...)

```

```

14     </recognizer>
15     (...)
16 </recognizers>
17 <triggers>
18     (...)
19     <trigger name="begin draw">
20         <source modality="speech" value="begin draw"/>
21     </trigger>
22     (...)
23     <trigger name="tools_one_hand">
24         <source modality="rfid" value="select | line | freehand"/>
25     </trigger>
26     (...)
27 </triggers>
28 <actions>
29     <action name="draw_operation">
30         <target name="xpaint_client" message="draw $oper $shape $posx $posy"/>
31     </action>
32     (...)
33 </actions>
34 <dialog leadtime="2100">
35     <context name="start">
36         (...)
37     <transition>
38         <trigger name="begin draw"/>
39         <result context="drawing"/>
40     </transition>
41     (...)
42 </context>
43 <context name="drawing">
44     <transition name="drawing_one_hand">
45         <par_and>
46             <!-- others : par_or, seq_and, seq_or -->
47             <trigger name="tools_one_hand"/>
48             <trigger name="color"/>
49             <trigger name="thickness"/>
50             <trigger name="position"/>
51         </par_and>
52         <result action="draw_operation"/>
53     </transition>
54     (...)
55 </context>
56     (...)
57 </dialog>
58 </integration_description>
59 </smuiml>

```

 Listing 6.6: A Midas translation of the multimodal paint application

```

1 rule translateReactivation
2   r = Reactivation
3   assert Touch { source => "reactivation", xpos => r.posx, ypos => r.posy, fudicial => r.sourceId }
4 end
5 rule translatePhidgetRFID
6   r = PhidgetRFID { value == "2342111" }
7   assert RFID { source => "phidgets", value => "filled circle" }
8 end
9 attempt beginDraw
10  Speech { word == "begin draw" }
11 end
12 attempt toolsOneHand
13  RFID { value == "select" || value == "line" || value == "freehand" }
14 end
15 rule start
16   c = Context { context == "start" }
17   beginDraw
18   modify c { context => "drawing" }
19 end
20 rule drawingOneHand
21   c = Context { context == "drawing" }
22   t = toolsOneHand
23   cl = color
24   t = thickness
25   p = position
26   Time←equal4 t, cl, t, p, 2100.ms
27   call XPaintClient.draw t.operation, t.shape, p.fudicial
28 end

```

6.3 Case Study #1: The Kinect Presenter

The first incarnation of Midas [141] has been presented at the Fifth International Conference on Tangible, Embedded and Embodied Interaction (TEI 2011). On stage, our presentation was driven by gestures captured by a Microsoft Kinect sensor. At the time, the Kinect was just released for early adopters and our abstractions allowed for a rapid prototyping of five hand gestures within this narrow time frame. We implemented a *double swipe left*, *double swipe right*, *two-handed swipe left*, *two-handed swipe right* and a *number* gesture. Figure 6.2 illustrates the double swipe gesture with the right hand. This gesture was used to advance to the next slide, while a double swipe gesture with the left hand switches to the previous slide. A double swipe was used to reduce accidental activation (false positives) as the hands are moving frequently in a similar way while presenting. Additionally, a spatial relation describes that the gesture can only be activated when the hand is below the waist in order

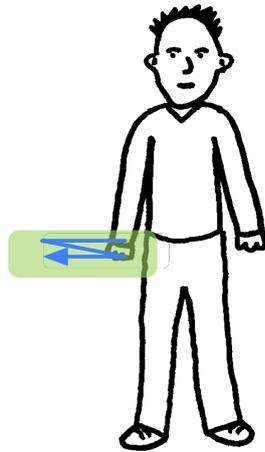


Figure 6.2: Double swipe right

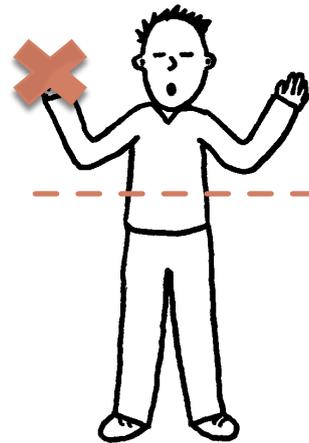


Figure 6.3: Swipe is blocked when hands are above the waist

to further reduce false positives (Figure 6.3). The *two-handed swipe left* and *two-handed swipe right* gestures were used to move to the beginning and end of the presentation. Finally the two stroke *Wave and Number* gesture illustrated in Figure 6.4 allows the user to switch to a particular slide number. The implementation of this *number* gesture is given in Listing 6.7. The definition of this gesture relied on a wave rule and on the \$1 recogniser with peak thresholding (Section 4.4.1) to recognise a number. We received positive and encouraging comments and questions by driving the Midas presentation with our kinect presenter application.

Listing 6.7: Wave and number gesture

```

1 rule showSlide
2   wave = Wave
3   nr = Number { joint == wave.joint }
4   wave←beforeF nr
5   call PowerPoint.showSlide(nr.number)
6 end

```

6.4 Case Study #2: Live Gesture Programming Session

At TEI 2011, we presented Midas during the demo session in order to demonstrate its rapid prototyping capabilities. During this session, different multi-touch gestures were spontaneously proposed by conference participants. The goal was to implement these gestures within a few

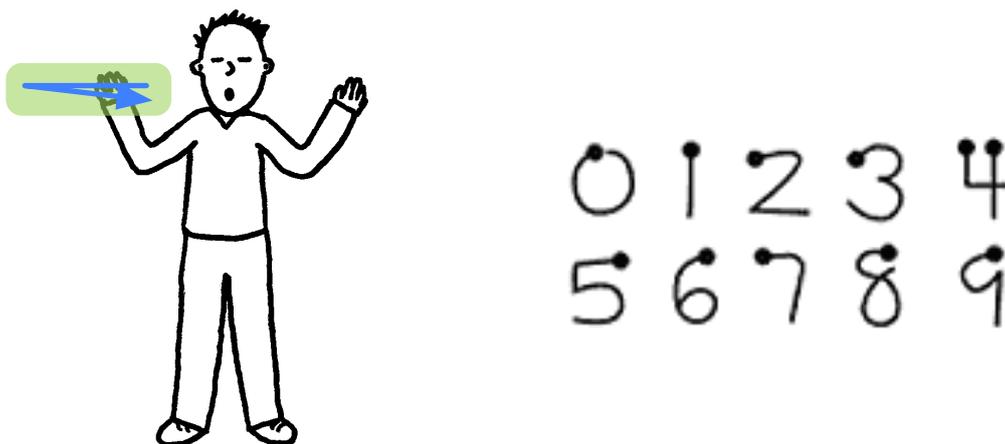


Figure 6.4: Wave followed by a number gesture

minutes. One of the proposals was the ‘Ohm’ gesture outlined in Figure 6.5. The implementation of this gesture is provided in Listing 6.8. The first step to implement a new gesture is to analyse the characteristics of the movement. In this example, the gesture consists of three strokes: two taps and one curvy movement. Taps had already been implemented before the start of the demo session and therefore we could simply reuse parts for our new gesture to significantly speed up the development process. The curvy movement was sampled five times such that new data could be matched using the DTW algorithm (Section 5.2.4). Complex segmentation was unnecessary as the curvy movement was performed as a single stroke (i.e. the touch down and up events serve as begin and end points). The composition of the ‘Ohm’ gesture outlined in Listing 6.8 specifies a spatial relation of the two taps to capture the intended execution. It is important to stress that the time interval definition used in this example does not enforce a particular order on the gestures (lines 5 and 6). A few minutes later, the conference participant who proposed the gesture was able to successfully activate the gesture. We note that the participant performed the gesture at a much slower rate than the recorded samples. However thanks to the intrinsic time-varying properties of DTW and the flexible temporal operators in Midas, this was not an issue



Figure 6.5: ‘Ohm’ gesture sketch

Listing 6.8: Hybrid ‘Ohm’ gesture

```

1 rule ohmGesture
2   curl = HorizontalCurl
3   dot1 = Tap
4   dot2 = Tap
5   curl←equalF dot1, 3.s
6   curl←equalF dot2, 3.s
7   curl←relative_insideF dot1, 1/3, 1.5
8   curl←relative_insideF dot2, 2/3, 1.5
9   assert Ohm { time_begin ⇒ curl.time, time_end ⇒ Math.max(dot1.time, dot2.time) }
10 end

```

A second proposed gesture was the Japanese symbol shown in Figure 6.6. This symbol consists of a single stroke with a lot of spatial movement, therefore it is a candidate to be recognised by the DTW algorithm. After 30 seconds, the participant was successfully able to trigger the gesture, although the threshold was set too strict for the first couple of times.



Figure 6.6: Japanese symbol

We illustrate a last multi-touch example proposed by an attendee of the TEI 2011 demo session. The gesture is illustrated in Figure 6.7 and consists of three curvy vertical lines. The three lines are performed concurrently using three fingers. Five samples were taken for a single line and their composition is handled via the rule shown in Listing 6.9. The rule consists of three `CurvyVerticalLine` facts, each performed by a different finger. These three conditional elements must happen within a time window of 500 milliseconds. A spatial operator is used to ensure they are performed close to each other.

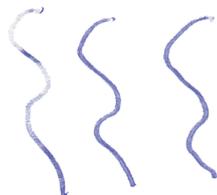


Figure 6.7: Three curvy vertical lines symbol

Listing 6.9: Three curvy vertical lines

```

1 rule threeCurvyVerticalLines
2   line1 = CurvyVerticalLine
3   line2 = CurvyVerticalLine
4   line3 = CurvyVerticalLine
5   line1←near_leftF line2, 30.px
6   line2←near_leftF line3, 30.px
7   Time←starts3F line1, line2, line3, 500.ms
8   assert ThreeCurvyVerticalLines { time_begin ⇒ line1.time_begin, time_end ⇒ line1.time_end }
9 end

```

6.5 Case Study #3: Declarative Gesture Spotting

Our declarative 2D and 3D gesture spotting approach offers a comprehensible representation of automatically inferred spatio-temporal conditions. These conditions can be defined between characteristic control points which are automatically inferred from a single gesture sample. In contrast to existing solutions which are typically constrained to narrow time windows, our gesture spotting approach offers automated reasoning over a complete motion trajectory. Last but not least, we offer gesture developers full control over the gesture spotting task and enable them to refine the spotting process without major programming efforts. This evaluation and the following description have been published at ICPRAM 2013 [66].

In order to evaluate our gesture spotting approach, we used the experimental data set by Wobbrock et al. [162] consisting of 16 unistroke gestures and a total of 1760 gesture samples which have been captured by 10 subjects on a pen-based HP Pocket PC. While the data set consists of segmented unistroke samples, we concatenated the data with additional noise (5%) to simulate a single stream of continuous two-dimensional data input. Unfortunately, datasets containing 2D gestures in a continuous streams of data are not available or are not widely used. Therefore, these results cannot be directly compared to related approaches. However related evaluations, such as performed by Amstutz et al. [3] are using a continuous stream of accelerometer data and demonstrate an average recall rate of 79% with a precision of 93%. This indicates our approach is useful for segmenting continuous streams of data which can later be verified using traditional classification algorithms as described in Section 4.5.4.

For each of the 16 gestures, we used a single representative sample to infer the control points. Table 6.1 highlights the performance of our

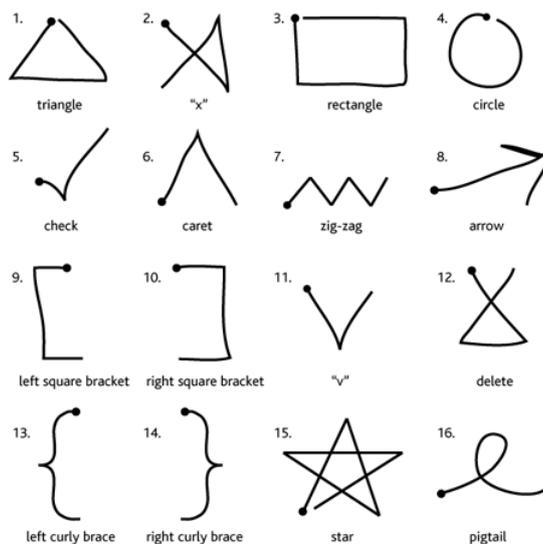


Figure 6.8: Unistroke gestures part of the \$1 data set

gesture spotting approach with 4 to 6 control points per gesture and a sliding window of 160 events. The default spatial variance of the control points is represented by the radius (r). The results in Table 6.1 consist of the recall (RC) as well as the precision (PR). Columns RC-E and PR-E represent the recall and precision of spotted gestures after applying expert knowledge. This involves manipulating conditions of control points and the addition of negation to invalidate certain trajectories.

r	RC (%)	PR (%)	RC-E (%)	PR-E (%)
22	77.50	52.10	78.75	56.50
24	83.13	47.16	84.38	52.53
26	90.63	42.40	91.25	46.79
28	93.75	39.47	94.38	43.26
30	97.50	35.37	97.50	39.29
32	98.75	32.78	98.75	36.41

Table 6.1: Declarative gesture spotting performance

As we can observe in Table 6.1, the automatically inferred control points per gesture allow for a high recall. The few non-spotted gestures originate from differences in the angle in which they were performed, which is a current limitation of our approach. However, in the near future, we plan to investigate methods to incorporate rotation invariant features. Note that our approach reasons over the complete trajectory, while still being able to process more than 400 times faster than real time for 60 Hz input and a gesture set consisting of 16 different two-dimensional gestures.

The relatively high number of wrongly spotted gestures is caused by the fact that several gestures, such as the *left curly bracket* and *right curly bracket* are similar to spot as the *left square bracket* and *right square bracket*. Additionally, the *check* gesture is frequently found as a partial match of other gestures. However, we argue that a gesture spotting solution should be optimised for a high recall since the filtering of submatches can be done at the classification level.

To demonstrate the power of expert refinements, we modified the *right curly bracket* rule to prevent points between the first and final control point to be too far off to the left and did some other small refinements for other gestures. These minor changes to the model took only a few minutes but resulted in an increase of the precision (PR-E) without reducing the recall (RC-E). In a broader context, such as full-body gesture recognition where multiple concurrent trajectories are to be processed, the expression of additional conditions is of major importance for reducing invalid spottings and to improve the performance.

Our control point-based gesture spotting approach automatically matches a combination of events adhering to the defined constrained trajectory at spotting time without any lossy preprocessing steps. It further provides a number of powerful features described in the previous chapters, such as offering an external representation 5.4.3, matching non-subsequent events (Section 5.3.4) and dealing with partial overlapping (Section 5.3.3).

6.6 Case Study #4: Augmented Live Music Performance

Nowadays many music artists rely on visualisations and light shows to enhance and augment their live performances. However, the visualisation and triggering of lights in popular music concerts is normally scripted in advance and synchronised with the music, limiting the artist's freedom for improvisation, expression and ad-hoc adaptation of their show. We argue that these limitations can be addressed by combining emerging non-invasive tracking technologies with an advanced gesture recognition engine. This application and the following description have been published at NIME 2013 [67].

In this section, we present a solution that uses explicit gestures and implicit dance moves to control the visual augmentation of a live music performance. We further illustrate how our framework addresses limit-

ations of existing gesture classification systems by providing a precise recognition solution based on a single gesture sample in combination with expert knowledge. The presented approach enables more dynamic and spontaneous performances and—in combination with indirect augmented reality—leads to a more intense interaction between artist and audience. Our goal was to address the limited interactivity and possibility for spontaneous adaptations and changes in live performances. The artists should be given more possibilities to interact with and influence the audience without the fixed scripted behaviour of visualisations. We argue that through the use of innovative non-invasive technologies, such as depth sensors, artists have a mean to perform expressive gesture control by using their body.

The use of our gesture-based indirect augmented reality solution during three live performances of the band Mental Nomad is shown in Figures 6.10, 6.11 and 6.12. The mix between continuous augmented reality and discrete activations of the visualisation layers provides an interesting view on the two large screens that have been installed at the sides of the stage as illustrated in Figure 6.9. The Kinect sensor was positioned at the front of the runway and the optimal tracking area was labelled on the ground as an aid for the artists. All Kinect skeleton data was transmitted through OSC to a backstage laptop connected to the two large screens. In total, five key gestures (G1 to G5) were used to activate different visualisation stages.

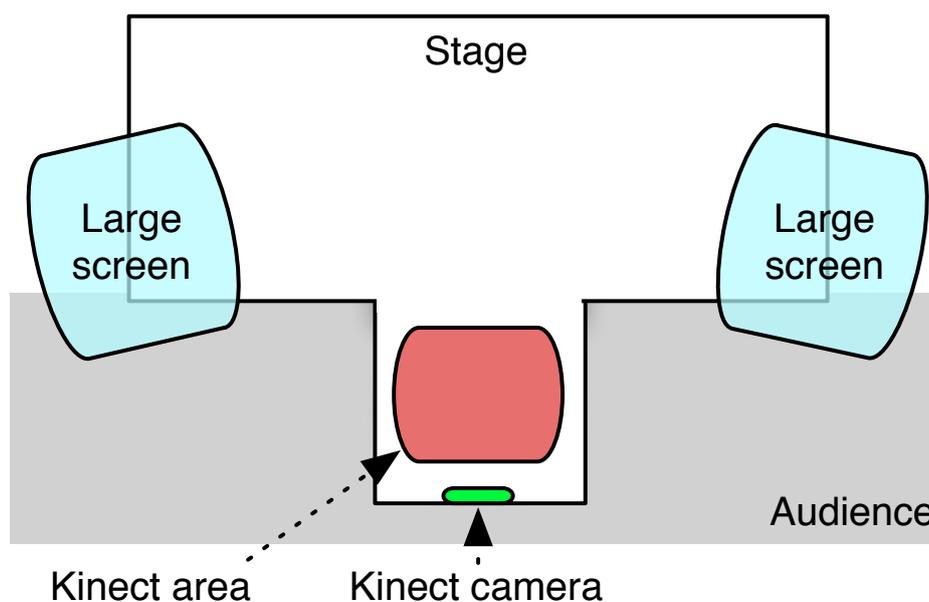


Figure 6.9: Stage configuration



Figure 6.10: Live performance (after gesture G3)

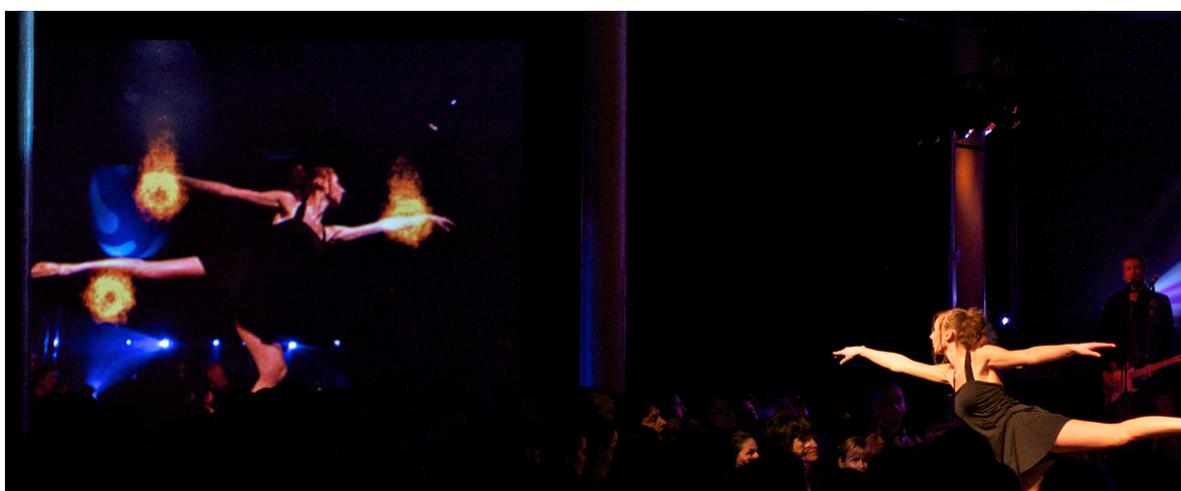


Figure 6.11: Live performance (gesture G4)

6.6.1 Constraints

The input stream contains a lot of non-relevant movement and the recogniser has to process this information in real time. Additionally, we did not have access to all the dance moves due to the minimal time budget for this part of the entire show. This constraint is problematic for many learning-based gesture recognition approaches since the system needs to “learn” both gestures as well as non-gestures [96]. In the discussed setup where explicit gestures and implicit dance moves are used to control the augmentation process, non-gestural movements form the norm rather than the exception. The artists requested us to trigger specific visualisations when a number of key moves are performed. Since a single song takes about four minutes and multiple artists are dancing in a more or less



Figure 6.12: Live performance (gesture G5)

controlled sequence, the key movements take up less than 5% of the overall time. Furthermore, the sequence was not known and variations are common due to the influence of the audience. To summarise, the presented scenario resulted in the following constraints:

One-shot gesture sampling Due to time constraints, we were unable to perform an iterative evaluation of the system together with the artists. Hence, single samples of the five key gestures were recorded and no *garbage data* was available to train a classifier with negative examples.

No garbage data The artist's choreography is scripted in a flexible manner. The robustness in the case of unscripted and uncommon moves was a requirement put up front by the artists. The lack of full-body data for the entire choreography complicates the creation of an idle state and the evaluation phase.

High precision The use of expressive control for a couple of key movements asks for a high precision. The system should not trigger its actions unintentionally as this would break the flow of the live performance.

High recall On the other hand, a high recall should be obtained in order to ensure that the actions are triggered when a key movement is performed. Imagine an artist performing a special jump and no visual feedback is delivered. The required nearly perfect precision and recall complicates the task and because of the limited recorded

data, the use of expert knowledge and reasoning over a larger time period seems appropriate.

Real-time processing Any activation that is based on expressive movements should happen nearly instantaneously and we should be able to process the data as it enters the system.

Non-invasive sensor technology It was requested that the technology should be non-invasive. The embedding of sensors in clothes was not an option due to the indoor scene which implies lightweight shirts and sweating which might negatively affect the sensors. We requested that the movements should be executed in an area of 5m². This allows a single Microsoft Kinect sensor to easily track the artist performing the moves.

Multiple users In our scenario, four artists were moving on stage but requested to be ignored in the dedicated camera area. As it was risky to perform the recognition process on the first artist entering the dedicated area at the beginning of the song, we decided to enable a multi-user gesture recognition process that allows multiple artists to trigger the actions.

6.6.2 Expressive Control

We present several guidelines on how an augmented reality system can be implemented, specifically paying attention to the constraints introduced in the previous section. There are two main parts of the application: the input side with the corresponding real-time input stream processing and the output side which takes care of the visualisation. The initial output modality idea consisted of a virtual avatar which was directly controlled by one of the artists. Key gestures would trigger additional visualisation elements such as fire or electricity. However, this has relatively fast been perceived infeasible due to the fact that many moves, such as a 360 degree rotation, cannot be tracked very accurately by the Microsoft Kinect SDK¹. Furthermore, existing avatar models did not fit well with the overall visualisation concept. An alternative solution is indirect augmented reality, where a live video feed of the artist is overlaid with certain visualisation elements which are triggered by gestures. This allows

¹Microsoft Kinect SDK: <http://www.microsoft.com/en-us/kinectforwindows/develop/overview.aspx>

us to deal with some of the inaccuracies of the Microsoft Kinect SDK without noticeable visual artefacts.

Expressive control through non-invasive technologies creates the opportunity to enable specific visualisations as commanded by the artist. The five gestures G1 to G5 which were used to trigger the indirect augmented reality are highlighted in Figure 6.13 to Figure 6.17 as a sequence of postures. Dealing with multiple postures over time is crucial when only a very few samples are available. The reason for this is to optimise the precision as gestures should not be unintentionally triggered. Without time information, the artist might trigger several end postures while dancing, taunting the audience or even trigger the change in augmentation if the Microsoft Kinect SDK incorrectly tracks the user. Additionally, when incorporating full-body gesture recognition, we do not prohibit the artist from executing other similar moves. By defining a precise body movement sequence to which the user must adhere in order to trigger the intended actions, we can optimise the system's recall and precision.

The interpretation of full-body movements over time requires advanced software engineering features. To achieve the necessary segmentation in gesture recognition, we relied on control points and used three-dimensional ellipsoids as the basic form. This solution generates declarative code from a single sample and allows expert users to further refine the gesture definition to achieve the necessary precision requested by the artist.



Figure 6.13: Gesture G1

Listings 6.10 and 6.11 show a partial implementation of gesture G1 which has been introduced earlier in Figure 6.13. A couple of control points per posture of a gesture and relationships of joints relatively to other joints in space, are defined in Listing 6.10. The conditional elements require that *there should be a relative joint from the torso (parent) to the left foot (child) that meets a number of conditions*. Several spatio-temporal operators are provided as built-in functionality but can be extended with custom



Figure 6.14: Gesture G2



Figure 6.15: Gesture G3

conditions. A developer further adjusts the gesture definition according to specialisation or flexibility requirements. For each control point such a definition has to be created, whereby user-defined operators can help in dealing with distance, angles or other spatio-temporal properties.

Listing 6.10: Control points for gesture G1

```

1 rule controlPoint3LeftFoot0
2   r = RelativeJoint { parent == Joint.TORSO && child == Joint.FOOT_LEFT }
3   r←enterEllipsoid -0.1425, -1.6867, 0.1372,
4     0.6216, 1.5064, 0.6314, 0, 0, 6.1897
5   assert Ellipsoid { name => "g1FootL0", user => r.user, time => r.time }
6 end
7
8 rule controlPoint1RightHand1
9   r = RelativeJoint { parent == Joint.SHOULDER_RIGHT && child == Joint.HAND_RIGHT }
10  r←enterEllipsoid 0.1529, -1.1261, 0.0352,
11    0.4534, 1.6957, 0.3887, 0, 0, 0.1501
12  assert Ellipsoid { name => "g1HandR1", user => r.user, time => r.time }
13 end
14 ...

```

In the definition of gesture G1 shown in Listing 6.11, control points are combined to form a complex gesture. Notice how all feet (torso-foot relative joint) and arms (shoulder-hand relative joint) are important in



Figure 6.16: Gesture G4



Figure 6.17: Gesture G5

this movement (lines 3 to 9), even though the left arm remains steady (Figure 6.13). For this gesture, the steady arm contains a lot of valuable information as we optimise for a precise gesture definition to give the artist the freedom to perform other, but similar movements that should not activate the gesture definition. Lines 7 to 9 describe the movement of the arm in three phases which could even be extended if higher precision is required.

By relying on a declarative language, we can easily manually refine various details without having to capture additional sample data. For instance, the angle of the arm in the end move could be less strict and the movement of the left leg can be used to optimise the precision. Additional control points can be added to make the gesture sequence stricter, including those based on other joints can be incorporated to refine the definition. The further we go back in time, the more precisely a gesture can be defined. However, this forces the user to perform the gesture in the same exact sequence.

Listing 6.11: Gesture G1: pointing up

```

1 rule pointingUp
2   group user
3   e1 = Ellipsoid { name == "g1FootL0" }
4   e2 = Ellipsoid { name == "g1FootL4" }
5   e3 = Ellipsoid { name == "g1FootR5" }
6   e4 = Ellipsoid { name == "g1HandL6" }
7   e5 = Ellipsoid { name == "g1HandR1" }
8   e6 = Ellipsoid { name == "g1HandR2" }
9   e7 = Ellipsoid { name == "g1HandR3" }
10  e1←meetsF e2, 3.s
11  e1←meetsF e5, 1.s
12  e5←meetsF e6, 3.s
13  e6←meetsF e7, 1.s
14  e3←containsF e5, e7
15  e4←containsF e5, e7
16  assert PointingUp { user ⇒ e1.user,
17    time.begin ⇒ e1.time, time.end ⇒ e7.time }
18 end

```

Furthermore, the declarative gesture definition enables an easy explanation to the artist on how a gesture is implemented and to what constraints their movements have to adhere. We do neither require extensive training data to implement new gestures, nor other non-gesture data. The Mudra recognition engine is able to process continuous full-body movements of multiple users over a period of time (i.e. related to online processing, segmentation and concurrent interaction criteria defined in Chapter 2). Furthermore, it allows precise detection (i.e. reducing false positives) and also results in a high recall. In this application, gestures occur in a fixed sequence which means that the activation of gestures can be further refined by adding a previous gesture activation as a conditional element (i.e. demonstrating customisation and extensibility).

In a declarative language, unification can be used to group certain entities. In this case, by using a single unbound variable for conditional elements on the user field, we automatically enable support for multiple users. The user identifier that triggers the gesture will be passed to the final application (line 16), such that the correct user will be augmented with the appropriate visual elements (i.e. related to *identification and grouping*).

Barry et al. [10] mention the “*trade-off between recognition quality and the delay of the real-time recognition*” as one element of future work. By using a declarative gesture spotting language, gestures can be recognised precisely and in real time. It does not require a preprocessing step that splits up the continuous stream into possible gesture candidates,

but rather computes the incremental gesture by using the efficient Rete algorithm.

6.6.3 Discussion and Conclusion

We are happy to report that the artists were extremely satisfied with the accuracy of the system and the added value it produced in terms of an excited audience and more flexibility for improvisation. Indeed, the reception from the audience was intense, partly due to the augmented reality stream, but mainly also due to the matching visual overlays that gradually increased during the song. We also received comments from the audience that the visualisations were extremely accurate and synchronised with the performed choreography. The combination of Midas and Mudra performed flawlessly without any false positives or missed gestures during all three live performances that took place in 2012 with an audience of about 1500 people.

6.7 Case Study #5: Hand Grip Assessment for Effort Discounting Tasks

In the context of a psychological study about effort cost decision making and its association with negative symptoms in schizophrenia [37], Mudra was used as a tool to implement an effort discounting task. An effort discounting task assesses how steeply a certain reward loses its subjective value with increasing effort. The experiment consisted of multiple phases, for which we refer to Docx et al. [37]. In one phase, namely the execution block, participants choose between a high effort task (i.e. squeezing the dynamometer) with a particular reward and a low effort task (i.e. holding the dynamometer) with a lower reward. The required grip strength is indicated by a line on a thermometer as shown in Figure 6.18. In this case, the high effort task corresponds to a minimum grip strength of 90% with a reward of €5 in virtual money. This assessment is performed by Mudra, which includes user-specific calibration, real-time user interface feedback and time constraints. The maximum strength of each user is determined before the start of the experiment by asking the participant three times to squeeze the dynamometer as hard as possible for a couple of seconds. During the execution block, the squeezing effort is represented by the level of the thermometer in real-time. Finally, temporal conditions specify that a user should reach the intended effort within 5 seconds

in order to obtain the corresponding reward. The experiment involved 40 patients with a DSM-IV diagnosis of schizophrenia and 30 age- and sex-matched healthy controls. This application highlights the ability of Midas and Mudra to describe and process data of a squeeze dynamometer input modality.



Figure 6.18: A hand dynamometer and two effort levels with their reward

6.8 Case Study #6: Water Ball Z

A final application to demonstrate the real-world usage of our multimodal framework is Water Ball Z. This application and the following description have been published and demonstrated at TEI 2014 [70]. Water Ball Z is a novel interactive two-player water game that allows kids and adults to “fight” in a virtual world with physical feedback. The body movement of a player is captured via an RGB-D sensor and analysed by a 3D gesture recognition engine. In order to enable tactile feedback without the need for wearable devices, a number of water nozzles are positioned around each user’s play area. The idea is to translate the input gesture of one player to the corresponding water spray hitting the other player. Besides severely reducing the risk of injury in a fight, Water Ball Z engages people in a real and fun experience where a hit is physically manifested via a water spray. Furthermore, power up moves and a live scoreboard extension bring the virtual world of Dragon Ball Z² and Mortal Kombat³ cartoons into real (augmented) life. In addition to our declarative description of

²Dragon Ball Z: <http://www.dragonballz.com>

³Mortal Kombat: <http://www.imdb.com/title/tt0122355>

3D gestures used in the physically augmented game, we show the usage of activation policies for mapping gesture input to haptic output.

Our proposed two-player Water Ball Z setup is illustrated in Figure 6.19. It consists of two circular play areas called battle stations. The size of these stations is defined by the range of the Kinect sensor denoted by (1) in Figure 6.19 and the water nozzle configuration (2). Each battle station requires one Kinect sensor tracking the movement of the player. The water nozzles are activated and deactivated by using solenoid valves, a wireless Arduino board and a dedicated computer running Linux (located near 3).



Figure 6.19: Water Ball Z setup

In our setup, the water sprayed out of the water nozzles shown in Figure 6.21 might easily reach the upper body of a player. For safety reasons, we opted for a setup where the solenoid module with the 12V electronics can operate wirelessly and fully disconnected from the electricity network by using a battery. The Kinect sensor and the computer responsible for the gesture recognition and wireless activation of water sprays are protected against water by covering them with plexiglas.

We currently distinguish seven key input gestures, including a regular jab and a haymaker (wide punch) with the left and right arm, left and right foot kicks as well as a circular motion gesture. Raising the hand to the head level and performing a circular motion triggers the final gesture. All gestures result in a water spray on the opponent's battle station. For instance, a regular jab with the right hand will inflict a short water spray (100 ms) from the front left nozzle aiming at the head of the opponent. As the spray is rather short, a double jab will have the effect of spraying

twice. Similarly, a haymaker is bound to the two sideway water nozzles at the opposite site. The foot kicks trigger a nozzle aiming to the back of the opponent. Finally, the circular motion gesture is a special gesture where all nozzles are enabled one-by-one in the same direction the gesture is performed. However, this action is only activated after one full circular motion has been performed. Note that we call this type of gesture a *shoot-and-continue* gesture.

In order to detect different moves of a player, we need online gesture processing without explicit segmentation. This implicit segmentation of a real-time input data stream has two advantages. First, players are not forced to perform explicit segmentation poses and can therefore try to evade water sprays in many ways while trying to attack their opponent at the same time. Second, by using a precise declarative gesture definition, sprays will only be triggered when a valid move is executed. Declarative reasoning over movement in time makes the recognition system more robust to false positives. The sprays only trigger for the correct movements which improves the feeling of a real battle compared to a system that reacts to less focused movements. If necessary, the gesture rules can be modified to support more variation.

Listing 6.12: Shoot-and-continue registration for the lasso gesture

```
1 rule(:lasso).register(:sac, 500.ms) { |p|
2   spray(p / 100 * sprays, 100.ms)
3 }
```

In Listing 6.12, the option `:sac` is used to express that the gesture should be completed at least once (i.e. shoot-and-continue) (Section 4.6.3). Subsequent intermediate steps of the gesture trigger the callback and make it behave like an online gesture. This results in a direct mapping between the location of the hand and the activation of the respective water nozzles installed at the battle station.

6.8.1 Electronic Schema

We used a wireless XinoRF Arduino board (similar to an Arduino UNO R3 with wireless connectivity) to digitally interface the solenoid valves with our computer. However, the digital pins of an Arduino board do not provide the necessary power to control the solenoid valves. In order to solve this problem, we designed a PCB using N-channel power MOSFETs (40V/23mΩ) and 1N4001 Diodes. We can then draw 12V from the Arduino board using an external battery to drive ten plastic water solenoid valves

(12V-1/2" Nominal). The resulting Water Ball Z solenoid module is highlighted in Figure 6.20.



Figure 6.20: Water Ball Z solenoid module

6.8.2 Solenoid Valves and Nozzles

For our prototype, we used off-the-shelf gardening hoses in combination with plastic solenoid valves. The water input is split into multiple hoses and is then switched by the solenoid valves. The final output of each water hose is attached to a nozzle aiming at a player's front, head, back or other parts. An installation of our prototype is shown in Figure 6.21.

Water Ball Z demonstrates the intrinsic capabilities of future NUI applications when developers are provided with adequate software engineering abstractions to exploit the potential of novel input modalities.

6.9 Conclusion

Midas and Mudra reconciled a high-level language with a performant architecture capable of fusing data-level, feature-level, and decision-level information (Section 3.1.3). In this chapter we have shown how our Midas and Mudra abstractions correspond to the criteria defined in Chapter 2. These 30 criteria illustrate that our work improves the state-of-the-art extensively in terms of language and architectural support for multimodal fusion. Note that our findings have been replicated in a recent survey by



Figure 6.21: Tests with an early Water Ball Z prototype

Cirelli et al. [26] by means of 14 compatible criteria (see Appendix G for more details).

Furthermore, in Section 6.2, we compared language abstractions for defining data-level gestures and decision-level multimodal fusion to our approach. These discussions in combination with many experiments using input modalities such as touch, depth cameras, piezoelectric sensors (Section 4.3.1), accelerometers and gyroscopes (Section 4.5.1), RFIDs (Section 4.4.5), Speech (Section 4.8) and digital pens (Section 5.2.4), demonstrate the applicability of our work in the multimodal domain. Furthermore, Midas and Mudra enable rapid prototyping and the implementation of complex gestures in challenging contexts, which are problematic in previous work.

7

Conclusions

The human-machine interaction is rapidly changing with the introduction of new commodity hardware, such as Apple's iPad, HP's Sprout and Microsoft's PixelSense and Kinect. This hardware consists of novel input sensors which facilitate an interaction paradigm which is much more advanced than the traditional keyboard and mouse setup. The development of *natural user interfaces* (NUI), whereby the machine tries to understand and anticipate the user's interaction, typically relies on a continuous monitoring of multiple input channels. In this context, the term machine should be interpreted widely, including traditional computers such as a laptop, embedded devices (in everyday tangible objects, such as an interactive coffee table¹) and seemingly invisible devices (such as the Nest thermostat² which detects the presence of people to regulate the heating in a room). The big challenge of NUI applications is the ability to correctly interpret a user's intention from sensory input.

The correct interpretation of sensory information from such powerful hardware sensors is complex [9, 38]. This is because relevant information is hidden in a continuous stream containing noise. The implementation complexity hinders the development and deployment of novel applications. In literature, the term *multimodal fusion* is used to describe the process of extracting relevant input information by combining multiple input modalities. Multimodal fusion allows developers to provide *complementary*,

¹Ideum's Coffee Table: <http://ideum.com/touch-tables/platform-coffee>

²Nest's Thermostat: <https://nest.com/thermostat/life-with-nest-thermostat>

assignment, redundancy or equivalent (CARE [29]) interaction modalities. However, central to these multimodal fusion assumptions is the ability to properly encode multimodal interaction patterns in a specification language that the machine understands. Furthermore, the machine needs to process the input in real-time such that the user's intention results in an appropriate action within a reasonable timespan.

Our analysis of related work revealed that existing multimodal frameworks can be categorised in two main strands: data stream-oriented architectures and semantic inferencing solutions. On the one hand, data stream-oriented architectures are efficient in processing high-frequency low-level input information but lack important programming abstractions needed by developers to express, reuse and combine multimodal interaction patterns. On the other hand, semantic inferencing architectures provide high-level abstractions but offer inadequate computational processing capabilities for fusing the vast amount of raw data.

7.1 Summary and Contributions

In this dissertation we presented novel programming abstractions to describe multimodal interaction patterns. For the first time, a high-level declarative programming language provides adequate expressiveness to specify fusion across the data, feature and decision levels. This domain-specific language, called Midas, allows developers to focus on the *essential* complexity when describing multimodal interaction patterns. Therefore, developers need to deal with less *accidental* complexity, such as the filtering of irrelevant events, updating intermediate states and garbage collection. Midas uses *declarative rules* to express the conditions of a multimodal interaction pattern. These conditions rely on the existence and the spatio-temporal relation of input events that were obtained from various input modalities. A spatio-temporal relation expresses a spatial or temporal link between two or more events. Midas provides adequate programming abstractions to help developers express these conditions in a modular and composable manner.

Midas programs run on top of Mudra, an optimised unified multimodal interaction architecture. Mudra is centred on a global information storage, called the fact base, which is populated by multimodal input events from various devices. Input events are transformed into facts. Facts represent information that happened at a particular point in time. As these facts arrive in a continuous manner, rules and other fusion processes

actively react to changes in the fact base and enrich it with their own interpretations based on combinations of facts. In order to do this efficiently, Mudra transforms the declarative Midas definitions into a directed acyclic graph that forms the basis for the Rete algorithm. Facts are progressively filtered and joined with other facts in order to infer knowledge. A main characteristic of the Rete network is the ability to cache intermediate results. Therefore, when a fact satisfies only one condition of a multimodal description that consist out of two conditions (i.e. facts a and b need to happen), an intermediate representation of the result is temporarily kept in memory. This information is maintained until the event expires and memory can be freed for newer input events. This allows the Mudra architecture to process low-level data streams as well as high-level semantic inferences in one efficient specification language.

Mudra's extensibility allows developers to incorporate existing fusion processes and contextual application information. Additional support for application integration is provided in the form of shadow facts, which is a novel technique to synchronise state between the fusion and application state. Mudra also enables cross-level fusion, which facilitates fusion of low-level and high-level data.

We analysed Midas and Mudra based on 30 criteria and compared them to the existing work. This analysis highlights important contributions of our work compared to state-of-the-art multimodal abstractions and opens novel opportunities for future work. Besides this, we performed an in-depth comparison between Midas and a representative data-level gesture language and a representative decision-level multimodal language. This comparison revealed that Midas supersedes existing state-of-the-art data-level and decision-level programming languages. Finally, Midas and Mudra were deployed "in the real world", including live programming sessions or challenging environments for live music performances. To summarise, we have successfully used our approach to process input data originating from multi-touch surfaces, depth sensors, piezoelectric sensors, accelerometers, gyroscopes, RFID readers, speech recognisers, digital pens and hand dynamometers. This means we cover various input modes, such as speech, pen, touch, gaze, and head and body movements. In the following we discuss the contributions of this dissertation.

7.1.1 Analysis of Criteria, Challenges and Open Issues in Multimodal Fusion Frameworks

Our first contribution is the formulation of 30 criteria for assessing the state-of-the-art capabilities of multimodal frameworks as well as to highlight open research opportunities (Chapter 2). The goal of this analysis was to categorise and explicitly expose design decisions to provide a better understanding and foster a discussion around the challenges, opportunities and future directions for multimodal frameworks. We observed that data stream-oriented solutions are lacking high-level programming abstractions to adequately describe multimodal interaction patterns. Furthermore, we concluded that semantic inferencing tools are not well-suited for processing vast amounts of raw data originating from novel hardware sensors. In general, the indicative classification of multimodal processing tools shows that it is still challenging to address many important aspects such as segmentation, negation, event expiration, grouping and scalability with the abstractions provided by existing solutions (Chapter 3).

7.1.2 Midas

Midas is a novel declarative domain-specific programming language with a focus on the modularisation and composition of multimodal interaction patterns. A Midas program consists of templates, modules, rules, attempts and functions. Input events from different modalities are translated into facts and stored in a central fact base. Rules can then try to find combinations of facts that match their conditions in a reactive manner. Modules, attempts and functions modularise the multimodal description logic into small reusable parts. This allows composition to form more complex interaction description without requiring a deep knowledge by the developer on all particular details. More concretely, a module groups attempts and functions under a single name. Modules can be inherited by templates in a similar way as mixins in object-oriented languages. This enables greater reusability in comparison to existing multimodal languages where composition is poorly supported. Attempts solve customisation and data entanglement concerns found in related work. Firstly, attempts allow similar conditions to be easily shared amongst rules. Secondly, they provide parameterisation of complex conditions to enable fine-grained customisation. Thirdly, they can exchange resulting details to the consuming rule through computed facts. Computed facts are untyped

facts and replace the need for (over-)generic templates. Functions describe purely functional computations, such as mathematical expressions.

Additionally, the syntactic distinction between attempts, functions and modifiers allows the interlacing of conditions and modifiers within a single rule. Existing rule languages explicitly separate the conditional side from the modifier side, which requires developers to split their logic into multiple rules without proper tools to propagate information. Finally, the Midas language is designed to enable compile-time developer feedback for common mistakes when declaratively implementing multimodal interaction patterns.

7.1.3 Mudra

Mudra is a unified multimodal interaction framework for processing low-level data streams as well as high-level semantic inferences. The shared bus architecture with template-local event expiration blurs the distinction between data, feature and decision level fusion and offers a performant and extensible architecture. Furthermore, the Midas language allows for an easy coordination of the Mudra's architectural components, including the dynamic instantiation of many services according to the various patterns of the input data. Multiple data formats and communication protocols are provided to extend Mudra with services outside the Mudra infrastructure. Mudra provides several powerful features that are not present in other architectures, including out-of-order event processing, non-subsequent event matching and partial overlap and segmentation. The combination of Midas and Mudra forms an adequate platform for authoring tools that want to provide a high-level declarative external representation. Overall, it demonstrates that a high-level declarative language can be used effectively to extract meaningful information from a vast amount of input events.

7.1.4 Shadow Facts

To improve the awareness of multimodal fusion processes to contextual information, such as application state, we introduced the notion of shadow facts. A shadow fact is a fact that exists in the fact base but which replicates the state of an application-level object. For example, when a Java class is annotated with a `@Shadow` annotation, the Mudra library automatically reifies its instances as facts with the classname as type and the fields of the class as slots. In this manner, fusion processes can rely

on the dynamic state of GUI components. For example, a multi-touch gesture can only be activated when it is performed on top of a figure.

Shadow facts provide a solution for the difficult separation of state in fusion engines and applications. Existing solutions provide a two-way exchange of information using topic-based publish/subscribe APIs and remote procedure calls. This results in the complex management of stateless event handlers, which trigger programming code outside the main application thread. Shadow facts avoid this complexity by integrating application state in the fusion engine. This *application symbiosis* reduces the accidental complexity and improves performance as candidate results can be filtered as soon as possible.

7.1.5 Control Point-based Gesture Spotting

We identified a novel design pattern, called *control points*, for describing 2D and 3D gestures in a declarative manner. Control points are based on the relative spatial location of multiple facts compared to a first, underspecified, fact. Subsequent conditional elements then define a relative spatial relation to this initial conditional element. This spatial relation is typically represented as a translation of an ellipse (for 2D gestures) or ellipsoid (for 3D gestures), but many variations are possible. The control points design pattern enables an automated segmentation strategy of a continuous stream of input data. It should be stressed that segmentation (also known as spotting), is one of the main challenges of always-on gesture recognition solutions.

To define control points, a single, well-formed sample is analysed by an expert. The expert defines a number of control points that characterise the gesture. A definition can be described in a declarative rule using spatial and temporal relations. Our evaluation is based on a standard 2D benchmark and that control points are effective to optimise for recall. Additional refinements to increase precision can be manually defined.

7.2 Shortcomings and Future Work

It would be interesting to explore a formal user study where the Midas and Mudra abstractions are compared to traditional imperative programming abstractions. In Chapter 1, we argued that declarative programming style allows the programmer to think about *what* the fundamental conditions are, instead of analysing *how* to process input events one by

one as necessary in an imperative approach. In Chapter 3, related research also highlights why a declarative approach is best suited to reduce the accidental complexity of programming multimodal interaction. In Chapter 6, we have show considerably effective results with our approach. These findings have been replicated by an independent survey, recently performed by Cirelli et al. [26]. Their analysis shows that, in contrast to imperative solutions, Midas is capable of addressing many criteria in a single, uniform approach. However, measuring the learning curve and productivity of our approach when used by a community of developers would be interesting to explore in future work.

As observed in Section 6.1, there is still room for future improvements. In particular, disambiguation criteria such as prioritisation, uncertainty and user profiling need additional future effort. We believe that the fundamental problem lies in a trade-off between latency and certainty of fusion processes, as illustrated in Figure 7.1 by Julià et al. [81]³. It is important for developers to provide a low-latency application. However, the use of novel sensors increases noise and uncertainty. In order to reduce incorrect interpretations from the machine, a wider scope of information is needed. This means reasoning over events originating from multiple input sources over a longer period in time. This increases accuracy, but also latency, which should be avoided. Therefore, we introduce the notion of *forgiving interfaces*, as explained in the next section.

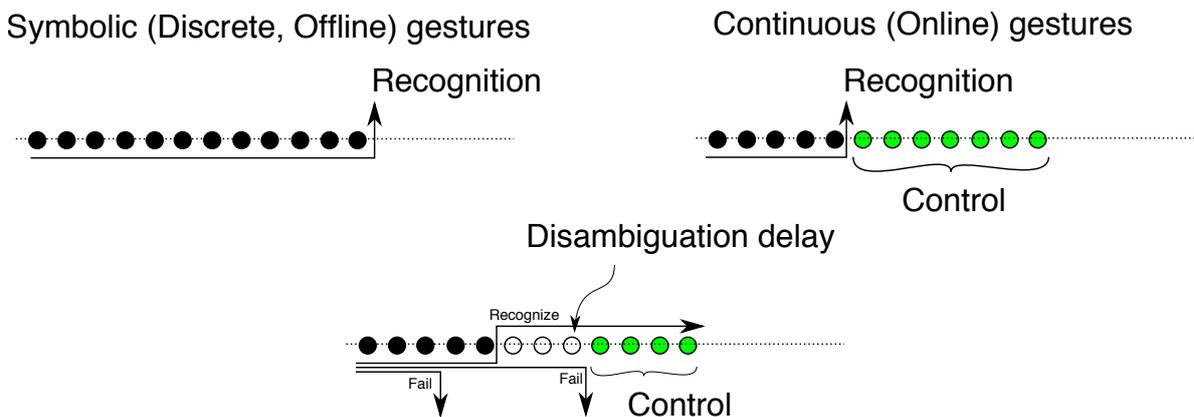


Figure 7.1: Disambiguation delay occurs between the evaluation state (black) and the recognition state (green)

³Figure 7.1 reproduced with permission of Carles Fernández Julià

7.2.1 Forgiving Interfaces

Forgiving interfaces are interfaces that aim for low latency and can undo certain operations when the interpretation of the commands of the user seemed incorrect at a later point in time. Additionally, they assist users in overcoming their own limitations. Moreover, *as long as we are human, we are bound to make mistakes (Anonymous)*.

Current programming languages do not embrace the notion of undo. When a variable is reassigned, its previous value is gone for good. Some programs, such as text editors support undo operators, but at the extent of manually storing intermediate states in a list. Although this requires some implementation, it does work well under the assumption that the input information enters in a predictable way. Input information produced by the user (e.g. mouse click) and the system (e.g. time) are indisputable and will always be true.

However, we observe two problems:

- The GUI might change just before the user clicks a button. This is especially common on mobile devices when browsing a website which is still loading its DOM. Users frequently click a wrong link because the browser just loaded a new image a few milliseconds before.
- There is a new major trend towards multimodal information which allows users to interact with the system using camera vision, multi-touch technology and voice recognition all at the same time. Thus applications will not only have to deal with dynamic, event-driven information but also with the inherent uncertain nature of that information. Consider the scenario presented in Figure 7.2.

In our envisioned forgiving interfaces paradigm, we can allow users to easily undo and correct incorrectly applied behaviour. Not only will these applications accommodate for human mistakes, it will also recover from mistakes made by the machines while still providing a low-latency interface which is preferred by all users.

7.3 Overall Conclusion

In an interview on the new era of Artificial Intelligence by Larry Larsen on 28 Jan 2015, Eric Horvitz, a Director of the Microsoft Research lab

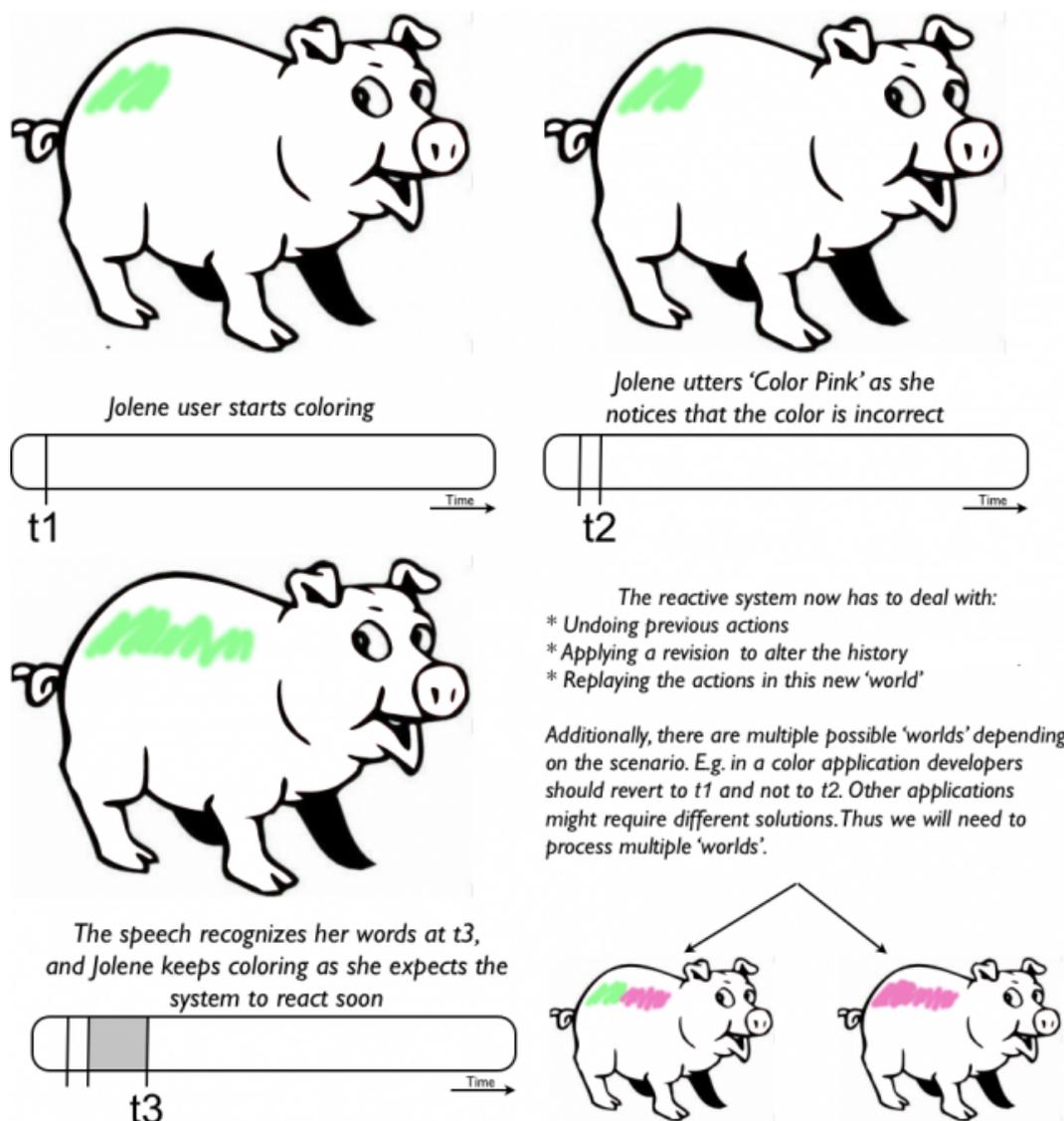


Figure 7.2: A colouring task in a forgiving interface

replied as follows to the question about what developers do today and how they prepare for the future with respect to AI and machine learning:

(...) One whole area that is very important for developers is that what they call multimodal systems. Systems that can take in different streams of information, anyone's calendar, visual signal, speech, pen and touch. How do you coordinate these things, how do you write software that is open to multiple sensory, perceptual input and data input at the same time. It is worth thinking deeply about and worth building skills in this space [101].

In order to fuse events coming from multiple input streams, an intermediate state of “the current progress” needs to be maintained. Unfortunately, the complexity of maintaining such an intermediate state is extremely high because it needs to capture complex spatio-temporal relations between many events. Moreover, the implementation of the update step needs to be efficient such that the system is ready to process the following event in a timely manner. Accidental complexity, such as segmentation (Section 2.3.2) and partially overlapping matches (Section 2.3.2) increases this complexity even further. Therefore, many proposed solutions offer a declarative language to describe complex patterns with a focus on *what* they are and not *how* they should be found. Unfortunately, these declarative approaches were limited to high-level information processing because their interpretation could not cope with the vast amount of raw events produced by sensors.

In this dissertation, we presented Midas and Mudra as a domain-specific solution to ease the implementation of multimodal fusion processes. Our declarative approach allows developers to focus on the essential complexity of fusion, while being relieved from tedious accidental complexity tasks, such as filtering irrelevant events, updating intermediate states and garbage collection. Furthermore, it enables a quick prototyping of promising multimodal interaction ideas.

We believe that Midas and its Mudra execution platform present a major leap forward to ease the development of multimodal interaction patterns. Our effort provides an answer to the overuse of statistical learning-based solutions which sacrifice important properties, such as control over the multimodal description, the ability to verify its results, as well as the ability to move towards a true understanding. Additionally, it allows developers to rapidly prototype complex multimodal interaction patterns, without having to rely on a vast amount of training data.

Our experiments with multi-touch, depth cameras, piezoelectric sensors, accelerometers and gyroscopes, RFIDs, speech and digital pens show the versatility and applicability of the presented approach. Some of them were performed in highly challenging environments where other solutions would fall short, in particularly for the scenarios where data gathering is not possible.

Finally, many criteria, such as composition, customisation, application symbiosis, unbound variables, segmentation, synchronising streams, concurrent interaction, spatial specification, temporal specification and spatio-temporal specification, have been tackled, or at least improved in contrast to related work. Remaining criteria, such as activation pri-

oritisation, prediction and user profiling are interesting candidates to solve in future work. However, we envision that a fundamental solution to properly disambiguate multimodal fusion results can be provided by developing forgiving interfaces, as explained in the future work section.

Midas is a fifth-generation programming language with a focus on providing high-level programming abstractions for multimodal interaction. Its Mudra execution engine caters a soft real-time, efficient incremental processing of a vast amount of input events. The combination of Midas and Mudra greatly simplifies the search for natural multimodal interactions by providing the much needed programming abstractions.

Appendices

A

Terminology in Multimodal Interaction

The following terminology is used throughout this dissertation to establish an unambiguous semantic understanding.

Modality A modality refers to the use of a medium, or channel of communication, as a means to express and convey information [19].

Multimodality A combination of modalities in order to improve transparency, flexibility, efficiency and expressiveness of human-computer interaction [124].

Multimodal Interaction Pattern A particular pattern executed by the human to convey a message to the computer.

Multimodal Interaction Pattern Description An implementation of a multimodal interaction pattern in a particular programming language.

Condition A part of the multimodal interaction pattern which should be satisfied by input events. A condition has a technical implication in the context of this dissertation and is similar to an *if* statement in traditional programming languages. For example, a multimodal interaction pattern description consists out of multiple conditions.

Activation The activation of a multimodal description happens when all conditions are satisfied.

Gesture A gesture in the context of this dissertation is based on the definition of Rhyne et al., namely a configuration of strokes, including handwritten text, pointing, and others [133]. This includes movement of the hands, face and other body parts to communicate a meaning. This includes the gesticulation, manipulations, semaphores, deictic and language gestures defined by Karam et al. [87] which is an extension of Quek et al. [130]

Single-stroke and multi-stroke gesture A gesture executed in a single, unbroken movement is a single-stroke gesture. When a gesture consists out of multiple strokes (i.e. when the pen or finger needs to be lifted to complete the gesture), it is called a multi-stroke gesture.

Segmentation Segmentation is the extraction process of a begin- and end-point from a continuous stream of data [116]. For example, the segmentation of a single step of a walk is described by moving the leg in front of the other leg.

F-measure An F-measure represents the accuracy of a classification. It is based on a harmonic mean of precision and recall.

Precision The amount of true positives divided by all positively classified samples. A high precision means that the algorithm returned more relevant than irrelevant results.

Recall The amount of true positives divided by the actual positive samples. A high recall means that an algorithm returned most of the relevant results.

True/False Positive A true positive sample is a positive sample which is correctly classified. This means that an interaction pattern existed and was correctly extracted. A false positive sample is the detection of an interaction pattern where it did not happen.

True/False Negative A true negative sample is a correctly classified negative sample. This means that no interaction pattern was

executed and nothing was detected. A false negative is the incorrect negative classification of a sample which contained an actual interaction pattern. In this case the multimodal fusion failed to detect a pattern that happened.

Template matching The comparison between an unknown sample and multiple known samples in order to classify the unknown sample.

Sample A segmented list of events, possibly conveying a multimodal interaction pattern.

Classification The translation of a number of events into a meaningful name.

Multimodal Fusion The combination of multiple events from different modalities to reveal new information (e.g. a pattern).

Multimodal Fission Multimodal fission refers to the decomposition of information into several smaller chunks [28].

Multimodal Framework A multimodal framework is a tool, consisting of an architecture and programming API, to ease the description of multimodal interaction patterns.

Multimodal Architecture A multimodal architecture manages the exchange of information between fusion processes.

Multimodal Programming API or Multimodal Language A programming interface for a multimodal framework. This can be provided in the form of a library with a number of preprogrammed functions or as a domain specific programming language.

Multimodal Fusion Process An executable piece of code responsible for a single multimodal fusion task.

Garbage Gesture Model The representation of a non-gesture, typically used in machine learning algorithms to diverge the learning process from other meaningful gestures.

Unification The process of finding a combination two terms with a

suitable substitution. This implies a search to a value which satisfies two or more conditions.

Low-level Data Low-level data is raw information originating from an input modality. Depending on the used sensor, it can contain a lot of noise and operate at a high frequency.

High-level Data High-level data is processed, semantic information.

Frequency Represents the amount of data being generated. A commonly used unit is hertz (Hz), representing the amount of events generated every second.

B

Transcript of the Formal Grammar of Midas

A Midas program mp consists of a set of templates t , modules m , rules r , attempts a , functions f and modifiers x . A template t has a mandatory name t_{id} , an optional mixin inclusion m_{id} and contains a number of slots, followed by attempts and functions. The mixin inclusion embeds a particular module into a template or module. Like templates, modules have a unique name, allow includes and supports slots, attempts and functions. However, modules cannot be instantiated and are meant to group functionality in a reusable namespace.

Slots are represented by a name s_{id} , are optionally typed and can contain a default value. Attempts (a) have a name a_{id} , a parameter list $\overline{l_{id}}$ and a number of conditions. Attempts have access to lexically scoped slots and optionally return computed facts cf . A computed fact is a local, anonymous fact that cannot be asserted or modified. Computed facts provide access to local information, such as conditional elements inside an attempt, to the outer scope, such as a rule or another attempt.

Functions f have a name f_{id} and an argument list $\overline{l_{id}}$. When invoked, they execute a number of expressions e and provide a return value. They can be embedded in modules or templates and have access to their lexical scope. Functions are typically used to express calculations.

A rule r starts with a name r_{id} and lists a number of conditions (c) and/or modifiers (m). Rules form the main execution paradigm of the pro-

gram as they react by invoking modifiers as soon as the conditions are met. Conditions (c) are composed from conditional elements (ce), tests (te), bindings (b) and special forms (sf). A conditional element (ce) expresses the need for a particular fact in the fact base. Conditional elements introduce reactivity, as new facts will be matched against them. A fact matched by such a conditional element can be bound to a local variable (l_{id}) using a bind. Optionally, inline constraint values (cv) can be used to express equality and nonequality conditions on the slot values of the matched fact. More sophisticated filtering can be achieved using tests (te). These tests express relations between expressions ($e <|\leq|==|!=|\geq|> e$) and can invoke global or template-scoped attempts. Template-scoped attempts use the left-arrow symbol (\leftarrow) and can be applied to bound conditional elements, templates and modules.

Bound variables allow developers to refer to facts matched by a conditional element, the result of an attempt or function invocation, a particular fact slot value (by referring to $l_{id}.s_{id}$), a primitive value or an array (a). Such an array is composed out of variables or primitive values. Special forms (sf) are expressions that denote negation (no), an asynchronous test ($async$) or a delayed verification ($wait$). Negation is an advanced construct that inverts the meaning of its nested conditions. When used in combination with conditional elements, the result is a negation by failure (i.e. the negation is true when no fact was found that satisfies the condition). It should be noted that binds from within such a negation are limited to their scope (i.e. a developer cannot reuse ‘negated’ values outside the scope of the special form). Asynchronous tests offer a way to obtain results asynchronously without relying on facts or callbacks. Finally, the *wait* special form is a language feature to delay the matching of the other conditions of a rule by a timeout value (v) which is relative to the time of a matched fact l_{id} .

Developers can use modifiers (m) to react when a number of conditions are matched. New events can be created by *asserting* them in the fact base. Matched facts can be *modified* by changing the slot values of conditional elements, or be removed from the fact base by *retracting* them. The final modifier is a *call* operator that invokes a function with side effects. For example, this function can shadow application-specific procedures.

We conclude the core set of semantic entities in Midas with basic values (v) such as numbers (-1,0,1,2), strings (“hello”), symbols (:sym) and nothing (nil).

C

Positioning and Discussion of Related Work

Listing C.1: Gesture languages

- 1 /*
- 2 *Midas*
- 3 [1] Hoste, L. *Software Engineering Abstractions for the Multi-Touch Revolution*. In *Proc. of ICSE 2010 (Cape Town, South Africa, May 2010)*
- 4 [2] Scholliers, C., Hoste, L., Signer, B., and De Meuter, W. *Midas: A Declarative Multi-Touch Interaction Framework*. In *Proc. of TEI 2011 (Funchal, Portugal, January 2011)*
- 5 [3] Hoste, L., Dumas, B. and Beat Signer, *Mudra: A Unified Multimodal Interaction Framework*, In *Proc. of ICMI 2011 (Alicante, Spain, November 2011)*
- 6 [4] Hoste, L., Rooms, B. D., and Signer, B. *Declarative Gesture Spotting Using Inferred and Refined Control Points*. In *Proc. of ICPRAM 2013 (Barcelona, Spain, February 2013)*
- 7 [5] Hoste, L., and Signer, B. *Expressive Control of Indirect Augmented Reality During Live Music Performances*, In *Proc. of NIME 2013 (Daejeon + Seoul, Korea Republic, May 2013)*
- 8 [6] Swalens, J., Renaux, T., Hoste, L., Marr, S., and De Meuter, W. *Cloud PARTE: Elastic Complex Event Processing based on Mobile Actors*. In *Proc. of AGERE! 2013 (Indianapolis, Indiana, USA, October 2013)*
- 9 [7] Hoste, L., and Signer, B. *Water Ball Z: An Augmented Fighting Game Using Water as Tactile Feedback*, In *Proc. of TEI 2014 (Munich, Germany, February 2014)*
- 10 [8] Marr, S., Renaux, T., Hoste, L., and De Meuter, W. *Parallel Gesture Recognition with Soft Real-Time Guarantees*. *Science of Computer Programming (February 2014)*
- 11 [9] Dox, L., de la Asuncion, J., Sabbe, B., Hoste, L., Baeten, R., Warnaeerts, N., and Morrens, M. (2015). *Effort discounting and its association with negative symptoms in schizophrenia*. *Cognitive Neuropsychiatry*.
- 12 GDL-K

- 13 [10] Khandkar, S. H., and Maurer, F. *A Domain Specific Language to Define Gestures for Multi-Touch Applications*. In *Proc. of DSM Workshop (Reno/Tahoe, Nevada, USA, 2010)*
- 14 *GeForMT*
- 15 [20] Kammer, D., Wojdziak, J., Keck, M., Groh, R., and Taranko, S. *Towards a Formalization of Multi-Touch Gestures*. In *Proc. of ITS 2010 (Saarbrücken, Germany, 2010)*
- 16 [21] Kammer, D., Henkens, D., and Groh, R. *GeForMTjs: A JavaScript Library Based on a Domain Specific Language for Multi-touch Gestures*. In *Proc. of ICWE 2012 (Berlin, Germany, 2012)*
- 17 [22] Kammer, D. *Formalisierung gestischer Interaktion für Multitouch-Systeme*. PhD thesis, Technische Universität Dresden, 2013
- 18 *GDL-E*
- 19 [30] Echtler, F., Klinker, G., and Butz, A. *Towards a Unified Gesture Description Language*. In *Proc. of HC 2010 (Aizu-Wakamatsu, Japan, 2010)*
- 20 [31] Echtler, F., and Butz, A. *GISpL: gestures made easy*. In *Proc. of TEI 2012 (Kingston, Ontario, Canada, 2012)*
- 21 *Proton*
- 22 [40] Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. *Proton: Multitouch Gestures as Regular Expressions*. In *Proc. of CHI 2012 (Austin, Texas, USA, 2012)*
- 23 [41] Kin, K. *Proton++: a customizable declarative multitouch framework*. In *Proc. UIST 2012 (Cambridge, Massachusetts, USA, 2012)*
- 24 *GestureAgents*
- 25 [50] Julia, C. F., Earnshaw, N., and Jorda, S. *Gestureagents: An agent-based framework for concurrent multi-task multi-user interaction*. In *Proc. of TEI'13 (Barcelona, Spain, 2013)*
- 26 *EventHurdle*
- 27 [60] Kim, J.-W., and Nam, T.-J. *EventHurdle: Supporting Designers' Exploratory Interaction Prototyping with Gesture-based Sensors*. In *Proc. of CHI 2013 (Paris, France, 2013)*
- 28 [61] Kim, J.-W., Nam, T.-J., and Park, T. *CompositeGesture: Creating Custom Gesture Interfaces with Multiple Mobile or Wearable Devices*. *International Journal on Interactive Design and Manufacturing (June 2013)*
- 29 *GestIT*
- 30 [70] Spano, L. D., Cisternino, A., Paterno, F., and Fenu, G. *Gestit: a declarative and compositional framework for multiplatform gesture definition*. In *Proc. of EICS 2013 (London, UK, 2013)*
- 31 *ICO*
- 32 [80] Hamon, A., Palanque, P., Silva, J. L., Deleris, Y., and Barboni, E. *Formal description of multi-touch interactions*. In *Proc. of EICS 2013 (London, UK, 2013)*
- 33 **/*
- 34
- 35 *// Data*
- 36 *d = [*
- 37 *[// Midas*
- 38 *{axis: "modularisation", value: 5}, // Each description is contained in a rule*
- 39 *{axis: "composition", value: 4}, // Descriptions can easily be composed by relying on high-level facts, attempts, computed facts and module inheritance*
- 40 *{axis: "customisation", value: 4}, // Rules can be copy/pasted, modified and extended; Attempts further enable this via parameters.*
- 41 *{axis: "negation", value: 5}, // Supports complex negation, with the possibility to wait for future events*
- 42 *{axis: "application", value: 4}, // Supports shadow facts to synchronise application info*
- 43 *{axis: "activation", value: 2}, // Bookkeeping facts (which are cumbersome but powerful) and activation flags (easy but limited)*
- 44 *{axis: "binding", value: 5}, // Provides unbound variables*
- 45

```

46  {axis: "online", value: 4}, // Supports online processing through alternation of conditions
    and modifiers or intermediate progress notifications
47  {axis: "offline", value: 4}, // Supports offline processing although explicit state checks are
    required
48  {axis: "overlapping", value: 3}, // Inherently deals with overlapping matching in the
    processing engine but does not inform the developer
49  {axis: "segmentation", value: 5}, // Optimises for recall to optimally segment candidates
50  {axis: "synchronisation", value: 5}, // Out-or-order, realignment and synchronised
    expiration supported
51  {axis: "expiration", value: 3}, // Offers a template-local relative time event expiration
    mechanism as well as bounded size. However, this needs to be set manually
52  {axis: "long-term", value: 3}, // Offers persistency and DRAM containers to ease long-term
    reasoning. However, this needs to be handled manually
53  {axis: "concurrent", value: 4}, // Requires few effort (i.e. user grouping) to support
    concurrent interaction
54  {axis: "embeddability", value: 3}, // Can be embedded in all kinds of applications (C
    library and networking APIs exists) but requires rules to be developed as a "string" value
55  {axis: "runtime", value: 4}, // Supports adding rules at runtime but does not have an
    automated refinement
56  {axis: "reliability", value: 4}, // Does not limit the event input rate but is sandboxed.
    Extensions have shown to scale in terms of performance to 64 cores however the
    approach is less flexible (no fusion across levels)
57
58  {axis: "spatial", value: 5}, // Offers extensive support for spatial operators
59  {axis: "temporal", value: 5}, // Offers extensive support for temporal operators
60  {axis: "spatio-temporal", value: 5}, // Spatio-temporal features and nesting of spatial and
    temporal operators is not restricted by the engine
61  {axis: "identification", value: 3}, // Uses logical binds to ease the identification problem
    and inherently supports the grouping problem. However, this results into multiple
    candidates that need to be verified later
62  {axis: "prioritisation", value: 2}, // Offers rule-specific prioritisation and uses
    bookkeeping facts to enable fusion. However it is hard to express a conflict resolution
    between multiple rules
63  {axis: "prediction", value: 2}, // An additional condition implies waiting for future events.
    Furthermore, developers can write delay statements but it not easy to manage
64  {axis: "verification", value: 3}, // Embeds other classifiers to verify candidates
65  {axis: "uncertainty", value: 1}, // Uncertainty can be part of a fact attribute but requires
    additional effort
66  {axis: "profiling", value: 1}, // Does not offer user profiling tools, but does provide a
    manual interface to access and store historical data
67
68  {axis: "readability", value: 2}, // Kammer[22] measured a low readability of Midas 1.0.
    Midas 2.0 abstracts a LISP-based syntax with a Ruby-style flavour
69  {axis: "debugging", value: 2}, // Offers no special debugging support
70  {axis: "authoring", value: 3} // An visual editor exists both for multi-touch and for full-body
    gestures
71 ],
72
73 [ // GDL-K
74  {axis: "modularisation", value: 5}, // Separates all gesture descriptions
75  {axis: "composition", value: 2}, // Allows developers to describe gestures in multiple steps
76  {axis: "customisation", value: 3}, // Gestures can be copy/pasted, modified and extended
77  {axis: "negation", value: 0}, // Does not support negation
78  {axis: "application", value: 1}, // Allows bounding boxes
79  {axis: "activation", value: 0}, // No special activation policies
80  {axis: "binding", value: 0}, // No unbound variables allowed

```

```

81
82 {axis: "online", value: 0}, // No support for online gestures
83 {axis: "offline", value: 5}, // Supports offline gestures by default
84 {axis: "overlapping", value: 2}, // Supports overlapping gestures but does not inform the
    developer
85 {axis: "segmentation", value: 0}, // Explicit begin and end points are required
86 {axis: "synchronisation", value: 0}, // No synchronisation of streams supported
87 {axis: "expiration", value: 2}, // State is reset when stroke ends but supports multiple
    strokes by storing the 'steps'
88 {axis: "long-term", value: 0}, // No support for long-term reasoning
89 {axis: "concurrent", value: 0}, // Stepwise finite-state machine is not replicated
    automatically
90 {axis: "embeddability", value: 2}, // .Net-oriented and string-based programming
91 {axis: "runtime", value: 2}, // Probably not supported, unclear
92 {axis: "reliability", value: 3}, // Specifies touch limits, no performance boundaries, only
    returns a set of points (sandboxed)
93
94 {axis: "spatial", value: 3}, // Primitive spatial operators available but not extendible to the
    event level (website offline, no further details)
95 {axis: "temporal", value: 1}, // Simple chained expressions
96 {axis: "spatio-temporal", value: 2}, // Few spatio-temporal operators built-in but not
    extendible (website offline, no further details)
97 {axis: "identification", value: 0}, // No support for identification and grouping
98 {axis: "prioritisation", value: 0}, // No support for prioritisation
99 {axis: "prediction", value: 0}, // No support for future event
100 {axis: "verification", value: 0}, // No support for verification
101 {axis: "uncertainty", value: 0}, // No support for uncertainty
102 {axis: "profiling", value: 0}, // No support for user profiling
103
104 {axis: "readability", value: 3}, // Few syntax rules and simple constructs increase the
    readability
105 {axis: "debugging", value: 1}, // No support for additional debugging
106 {axis: "authoring", value: 0} // No support for editor
107 ],
108
109 [ // GEFORMT
110 {axis: "modularisation", value: 5}, // Separates all gesture descriptions
111 {axis: "composition", value: 2}, // Allows developers to compose events from atomic
    building blocks but not from described gestures
112 {axis: "customisation", value: 2}, // Gestures can be copy/pasted, modified and extended,
    however the order of relations might require some work
113 {axis: "negation", value: 0}, // No support for negation
114 {axis: "application", value: 3}, // Supports object regions via the HTML dom tree but
    lacks expressiveness
115 {axis: "activation", value: 0}, // No special activation policies
116 {axis: "binding", value: 0}, // No unbound variables allowed
117
118 {axis: "online", value: 3}, // Online gestures are supported but only within a specific type of
    gestures (few constructs available)
119 {axis: "offline", value: 5}, // Supports offline gestures by default
120 {axis: "overlapping", value: 2}, // Supports overlapping gestures but does not inform the
    developer
121 {axis: "segmentation", value: 0}, // Explicit begin and end points are required
122 {axis: "synchronisation", value: 1}, // Maps each finger to a preprocessor, resulting in
    limited support for synchronisation

```

```

123  {axis: "expiration", value: 2}, // State is reset when stroke ends but supports multiple
      strokes using a global slack variable 'contiguityInterval'
124  {axis: "long-term", value: 0}, // No support for long-term reasoning
125  {axis: "concurrent", value: 0}, // Requires developers to split the canvas manually, no
      inherent multi-user support
126  {axis: "embeddability", value: 0}, // JavaScript support but via string-based programming
127  {axis: "runtime", value: 4}, // Supports adding gesture definitions at runtime but does not
      have an automated refinement
128  {axis: "reliability", value: 3}, // Specifies touch limits, no performance boundaries, only
      returns a set of points (sandboxed)
129
130  {axis: "spatial", value: 4}, // No path-level spatial operators available, direction
      preprocessors and template matching provide useful build-in constructs
131  {axis: "temporal", value: 2}, // Temporal expressions can be constructed using the build-in
      relations but requires global slack variables
132  {axis: "spatio-temporal", value: 3}, // Spatial and temporal expressions can be combined,
      albeit in a restricted format
133  {axis: "identification", value: 0}, // No support for identification and grouping
134  {axis: "prioritisation", value: 0}, // No support for prioritisation
135  {axis: "prediction", value: 0}, // No support for future event
136  {axis: "verification", value: 0}, // No support for verification
137  {axis: "uncertainty", value: 0}, // No support for uncertainty
138  {axis: "profiling", value: 0}, // No support for user profiling
139
140  {axis: "readability", value: 4}, // Clear and simple constructs make it a highly readable
      approach
141  {axis: "debugging", value: 2}, // Is embedded in a test environment where multiple options
      can be enabled. Can support further incremental matching feedback
142  {axis: "authoring", value: 0} // No graphical editor of the code is provided
143  ],
144
145  [ // GDL-E
146  {axis: "modularisation", value: 5}, // Separates all gesture descriptions
147  {axis: "composition", value: 0}, // Does not support composition
148  {axis: "customisation", value: 3}, // Gestures can be copy/pasted, modified and extended
149  {axis: "negation", value: 0}, // No support for negation
150  {axis: "application", value: 2}, // Inherently supports regions but need to be manually
      synchronised with the GUI
151  {axis: "activation", value: 2}, // Supports a number of flags to guide the activation process
152  {axis: "binding", value: 0}, // No support for unbound variables
153
154  {axis: "online", value: 2}, // Online gestures are supported by offering native filters
155  {axis: "offline", value: 5}, // Supports offline gestures by default
156  {axis: "overlapping", value: 2}, // Supports overlapping gestures but does not inform the
      developer
157  {axis: "segmentation", value: 0}, // Explicit begin and end points are required
158  {axis: "synchronisation", value: 1}, // Maps each finger to a preprocessor, resulting in
      limited support for synchronisation
159  {axis: "expiration", value: 1}, // Unclear how expiration of events is handled. Probably
      there is a slack variable indicating the maximum number of events per filter
160  {axis: "long-term", value: 0}, // No support for long-term reasoning
161  {axis: "concurrent", value: 4}, // Concurrent gestures are supported and can be splitted by
      region
162  {axis: "embeddability", value: 4}, // The format is compatible with JSON which allows for
      native syntax development in Javascript
163  {axis: "runtime", value: 4}, // Gestures can be added and modified at runtime

```

```

164  {axis: "reliability", value: 3}, // Does not limit the event input rate, but only returns a
      set of points (sandboxed)
165
166  {axis: "spatial", value: 4}, // A template path can be used to define the gesture. Adding
      new features are required to provide more spatial constructs. Relative spatial operators
      are available.
167  {axis: "temporal", value: 0}, // No special support for expressing temporal relations
168  {axis: "spatio-temporal", value: 1}, // Spatio-temporal information can be embedded in
      feature extractors (but lacks language support)
169  {axis: "identification", value: 0}, // A form of grouping is provided for GUI objects but it
      is unrelated to the identification and grouping problem of multiple input streams
170  {axis: "prioritisation", value: 1}, // GUI regions can provide a priority level for gestures
171  {axis: "prediction", value: 1}, // A 'delay' feature that expresses the time between events,
      but waiting for future knowledge is not supported on the language level
172  {axis: "verification", value: 0}, // No support for verification
173  {axis: "uncertainty", value: 0}, // No support for uncertainty
174  {axis: "profiling", value: 0}, // No support for user profiling
175
176  {axis: "readability", value: 3}, // Quite some flags and a lot of string quoting reduce the
      readability but the constructs are JSON compatible which is well known to many
      developers
177  {axis: "debugging", value: 1}, // No support for additional debugging
178  {axis: "authoring", value: 0} // No support for editor
179 ],
180
181 [ // Proton
182  {axis: "modularisation", value: 5}, // Separates all gesture descriptions
183  {axis: "composition", value: 0}, // Does not support composition
184  {axis: "customisation", value: 1}, // It is quite hard to extend the gesture due to the
      importance of ordering the regular expression. Might cause an exponential increase in
      complexity when adding a single requirement
185  {axis: "application", value: 3}, // The result of a hit test is returned as an attribute.
      More complex GUI correlations are not supported
186  {axis: "activation", value: 2}, // A confidence calculator is provided
187  {axis: "binding", value: 0}, // No unbound variables supported
188
189  {axis: "negation", value: 0}, // Negation is not supported
190  {axis: "online", value: 3}, // Online gestures are supported, however of limited complexity
191  {axis: "offline", value: 3}, // Offline gestures are also supported, however of limited
      complexity
192  {axis: "overlapping", value: 4}, // Explicitly blocks overlapping gestures to help the
      developer deal with (potentially) problematic definitions
193  {axis: "segmentation", value: 0}, // Explicit begin and end points are required
194  {axis: "synchronisation", value: 0}, // No support for synchronisation, requires strict
      ordering of conditions
195  {axis: "expiration", value: 1}, // State is reset when stroke ends
196  {axis: "long-term", value: 0}, // No long-term reasoning
197  {axis: "concurrent", value: 1}, // Simultaneous gestures are not supported. A possible
      extension is described in the paper but might require quite some work and an extension
      to the language. Furthermore there is no support for concurrent input streams
198  {axis: "embeddability", value: 1}, // It is challenging to describe the regular expressions
      with the optional attributes within a plain ascii text code environment
199  {axis: "runtime", value: 2}, // Probably but unsure about the static analysis of conflicting
      gestures
200  {axis: "reliability", value: 3}, // Does not limit the event input rate, but only returns a
      set of points (sandboxed)

```

```

201
202   {axis: "spatial", value: 2}, // No path-level spatial operators available. A direction
      preprocessor using the last two events is provided
203   {axis: "temporal", value: 2}, // Provides a sequential or parallel construct without support
      for more complex temporal relations
204   {axis: "spatio-temporal", value: 0}, // Preprocessors compare the two latest input events
205   {axis: "identification", value: 0}, // No support for dealing with identification or complex
      grouping problems. The canvas can however be split in to support two groups
206   {axis: "prioritisation", value: 3}, // Explicit prioritisation level required when conflicts
      are detected
207   {axis: "prediction", value: 1}, // A hint to support a 'Z' touch event is specified but
      currently not supported. Additionally, all attributes must be decided on the event level
208   {axis: "verification", value: 0}, // No support for verification
209   {axis: "uncertainty", value: 0}, // No support for uncertainty
210   {axis: "profiling", value: 0}, // No support for user profiling
211
212   {axis: "readability", value: 2}, // Regular expressions are known for most programmers
      however a multi-touch gesture definition rapidly becomes excessively long. It is also not
      fully compatible with ascii based code fragments
213   {axis: "debugging", value: 3}, // Incremental visualisation of the gesture states
214   {axis: "authoring", value: 4} // Tablatures compiles into regular expressions which implies
      the use of a modifiable external representation. Unfortunately it might be limiting for
      certain gestures
215 ],
216
217 [ // GestureAgents
218   {axis: "modularisation", value: 5}, // Every agent is responsible for one gesture
219   {axis: "composition", value: 5}, // Agents can be composed out of other agents
220   {axis: "customisation", value: 0}, // Poor customisation due the lack of abstractions to
      specify gestures themselves. Imperative programming languages are no good fit for
      defining gestures (not really the focus of the paper)
221   {axis: "application", value: 4}, // Allows agents to poll for GUI context
222   {axis: "activation", value: 0}, // No special activation policies
223   {axis: "binding", value: 1}, // Variable bindings are support in a manual manner
224
225   {axis: "negation", value: 1}, // Potentially possible when manually managing the gesture
      state
226   {axis: "online", value: 3}, // The framework supports online gestures but does not aid in
      coding them
227   {axis: "offline", value: 3}, // The framework supports offline gestures but does not aid in
      coding them
228   {axis: "overlapping", value: 3}, // By design, only one input event can be part of one
      gesture. However it is unclear how this can be tested at design time
229   {axis: "segmentation", value: 1}, // Each agent can evaluate a potential gesture candidate
      and potentially acquire a lock to decide it. However segmenting a continuous stream
      without dedicated gesture programming support is very complex
230   {axis: "synchronisation", value: 2}, // Ability to lock gesture activation until later
      information arrives
231   {axis: "expiration", value: 0}, // Each implementation points are required to expire
      temporal state manually
232   {axis: "long-term", value: 0}, // No support for long-term reasoning
233   {axis: "concurrent", value: 4}, // Offers duplication operators for agents to support
      concurrent interaction
234   {axis: "embeddability", value: 1}, // The abstractions are part of a host language and
      therefore hard to exchange

```

```

235 {axis: "runtime", value: 2}, // Agents can be added at runtime but it requires Python meta
    programming
236 {axis: "reliability", value: 3}, // Does not limit the event input rate, but only returns a
    set of points (sandboxed)
237
238 {axis: "spatial", value: 0}, // No abstractions for spatial operators
239 {axis: "temporal", value: 0}, // No abstractions for temporal operators
240 {axis: "spatio-temporal", value: 0}, // No abstractions for spatio-temporal operators
241 {axis: "identification", value: 2}, // No support for expressing identification and grouping
    . However by duplicating recognisers a primitive form of grouping can be supported
242 {axis: "prioritisation", value: 4}, // Supports priorities and multiple explicit conflict
    resolution mechanisms. Furthermore, each agent can poll for additional context
    information. However it might be perceived as slow by users due to waiting periods from
    other agents
243 {axis: "prediction", value: 2}, // Requires to set a waiting time for each gesture, but
    supports a compatibility mechanism
244 {axis: "verification", value: 0}, // No support for verification
245 {axis: "uncertainty", value: 0}, // No support for dealing with uncertain events
246 {axis: "profiling", value: 0}, // No support for user profiling
247
248 {axis: "readability", value: 0}, // Poor readability due the lack of abstractions to specify
    gestures themselves
249 {axis: "debugging", value: 2}, // No support for additional debugging besides logging
    potential conflicts at runtime
250 {axis: "authoring", value: 0} // No support for graphical editor
251 ],
252
253 [ // EventHurdle
254 {axis: "modularisation", value: 5}, // Each gesture has its own hurdle chain
255 {axis: "composition", value: 3}, // Multiple hurdles can be combined into one gesture event
    but it is unclear how this is done
256 {axis: "customisation", value: 4}, // Hurdles can be copy/pasted and easily be modified
257 {axis: "application", value: 0}, // No integration with GUI components
258 {axis: "activation", value: 0}, // No special activation policies
259 {axis: "binding", value: 0}, // No support for unbound variables
260
261 {axis: "negation", value: 4}, // False hurdles are supported but are required for each element
    in the chain
262 {axis: "online", value: 1}, // Unclear but might be supported due to the simple temporal
    state machine
263 {axis: "offline", value: 5}, // Supports offline gestures with simple constructs
264 {axis: "overlapping", value: 2}, // Supports overlapping gestures but does not inform the
    developer
265 {axis: "segmentation", value: 4}, // Inherently supports the segmentation of continuous
    input
266 {axis: "synchronisation", value: 0}, // Multiple streams are not supported
267 {axis: "expiration", value: 2}, // State is reset based on a slack timeout variable
268 {axis: "long-term", value: 0}, // No long-term reasoning
269 {axis: "concurrent", value: 0}, // Each gesture has one hurdle state and needs to be reset
270 {axis: "embeddability", value: 1}, // The generated code can be easily ported but required
    development effort
271 {axis: "runtime", value: 1}, // Adding gestures at runtime depends on the host language
272 {axis: "reliability", value: 3}, // Does not limit the event input rate, but only returns a
    gesture result (sandboxed)
273

```

```

274  {axis: "spatial", value: 4}, // Path-based spatial operators are provided in a graphical
      manner.
275  {axis: "temporal", value: 2}, // Serial and parallel relations are supported
276  {axis: "spatio-temporal", value: 3}, // Works for acceleration data
277  {axis: "identification", value: 0}, // No support for identification and grouping
278  {axis: "prioritisation", value: 0}, // No support for prioritisation
279  {axis: "prediction", value: 0}, // No support for future event
280  {axis: "verification", value: 0}, // No support for verification
281  {axis: "uncertainty", value: 0}, // No support for uncertainty
282  {axis: "profiling", value: 0}, // No support for user profiling
283
284  {axis: "readability", value: 4}, // The hurdles are easy to understand due to their
      graphical nature
285  {axis: "debugging", value: 4}, // Incremental debugging of which hurdles are taken and
      which are not
286  {axis: "authoring", value: 4} // Visual editor is provided to edit the hurdles and it compiles
      into a readable code due to the simple temporal logic
287 ],
288
289 [ // GestIT
290 {axis: "modularisation", value: 5}, // Gestures are described in their own definition
291 {axis: "composition", value: 5}, // Gestures can be composed by their name
292 {axis: "customisation", value: 2}, // Gestures can be copy/pasted, modified and extended,
      however the order of relations might require some work
293 {axis: "application", value: 3}, // Gestures are limited to a single region
294 {axis: "activation", value: 0}, // No special activation policies are provided
295 {axis: "binding", value: 0}, // Not supported. A good example of this issue can be related to
      Table 6 in their paper with a small modification: the left or right hand should be closed
      followed by the opening of the same hand '(cHr[closed] [] cHl[closed]) >> (cHr[open] []
      cHl[open])*'. In this case to distinguish between the right and the left hand requires code
      duplication.
296
297 {axis: "negation", value: 3}, // A disabling operator is provided but offers no additional
      temporal logic
298 {axis: "online", value: 3}, // Online gestures are supported, however of limited complexity
299 {axis: "offline", value: 3}, // Offline gestures are supported, however of limited complexity
300 {axis: "overlapping", value: 4}, // Supports overlapping gestures without delay but requires
      a compensation method
301 {axis: "segmentation", value: 0}, // Not supported. Requires the use of slack variables to
      threshold a virtual screen using the Kinect sensor.
302 {axis: "synchronisation", value: 3}, // Supports an interactive statement to ignore
      repetitive data in addition to order independence
303 {axis: "expiration", value: 1}, // State is reset when it becomes invalid to support the
      gesture
304 {axis: "long-term", value: 0}, // No long-term reasoning supported
305 {axis: "concurrent", value: 0}, // Each gesture has one state and needs to be reset
306 {axis: "embeddability", value: 1}, // It is challenging to describe the GestIT expressions
      with its attributes within a plain ascii text code environment
307 {axis: "runtime", value: 4}, // Supports adding gesture definitions at runtime but does not
      have an automated refinement
308 {axis: "reliability", value: 3}, // Does not limit the event input rate, might cause many
      compensating actions
309
310 {axis: "spatial", value: 3}, // Supports the use of boolean functions implemented in the host
      language with a list of the event history (manual separation)
311 {axis: "temporal", value: 2}, // Limited set of built-in temporal operators

```

```

312 {axis: "spatio-temporal", value: 4}, // Spatial and temporal can be nested but in a
    restricted manner
313 {axis: "identification", value: 0}, // No support for dealing with identification or complex
    grouping problems.
314 {axis: "prioritisation", value: 0}, // The application level needs to decide between gestures
    , potentially requiring to revert actions
315 {axis: "prediction", value: 0}, // No support for future event
316 {axis: "verification", value: 0}, // No support for verification
317 {axis: "uncertainty", value: 0}, // Uncertainty is not supported
318 {axis: "profiling", value: 0}, // No support for user profiling
319
320 {axis: "readability", value: 2}, // The language is dense and is not fully compatible with
    ascii based code fragments which make it hard to read during development
321 {axis: "debugging", value: 4}, // Incremental debugging of the expression evaluation in a
    graphical IDE
322 {axis: "authoring", value: 0} // Textual description of the gestures
323 ],
324
325 [ // ICO
326 {axis: "modularisation", value: 4}, // Modularisation is supported, but becomes a bit
    complex when extending the system
327 {axis: "composition", value: 4}, // Composition is supported, but becomes a bit complex
    when extending the system
328 {axis: "customisation", value: 2}, // Modifying or extending definitions is not trivial as it
    requires developers to check all transitions and be complete
329 {axis: "application", value: 0}, // No support for GUI specific relations
330 {axis: "activation", value: 0}, // No special activation policies
331 {axis: "binding", value: 0}, // No support for variable bindings
332
333 {axis: "negation", value: 3}, // It is unclear if negation is supported and how it can be used,
    but petrinets can provide this functionality via weighted arcs, capacities, coloured tokens
    and reset arcs
334 {axis: "online", value: 3}, // Online gestures are supported but require developer effort
335 {axis: "offline", value: 3}, // Offline gestures are also supported, however of limited
    complexity
336 {axis: "overlapping", value: 5}, // Supports locally overlapping gesture definitions
337 {axis: "segmentation", value: 0}, // Segmentation is not supported
338 {axis: "synchronisation", value: 4}, // Maintains an arbitrary number of tokens until data
    is available. However the global state needs to remain in a valid combination
339 {axis: "expiration", value: 3}, // It is unclear how events expire, but all state transitions
    are required to be described by the developer (i.e. event expiration is tackled but similar
    to other requirements it requires development time)
340 {axis: "long-term", value: 2}, // Can rely on long-term token activations but is limited to
    DRAM memory
341 {axis: "concurrent", value: 5}, // Concurrent interaction is inherently supported
342 {axis: "embeddability", value: 0}, // It is unclear how the resulting code looks like and how
    the models are stored
343 {axis: "runtime", value: 1}, // It is unclear if gestures can be added at runtime
344 {axis: "reliability", value: 4}, // Limits event rates via a 'FingerPool', only triggers an
    event at the application side (sandboxed), verifies all transitions
345
346 {axis: "spatial", value: 1}, // No path-level spatial operators available. Few built-in (simple)
    movement operators
347 {axis: "temporal", value: 3}, // Supports complex temporal relations by writing code on the
    transitions

```

```

348 {axis: "spatio-temporal", value: 3}, // Supports arbitrary combinations of spatial and
    temporal operators
349 {axis: "identification", value: 4}, // Each activation (e.g. a finger) can cause an
    individual token that can be passed through the network. The grouping problem is tackled
    by providing a re-clustering phase
350 {axis: "prioritisation", value: 2}, // Provides a form of simple resolution rules to describe
    relations between gestures
351 {axis: "prediction", value: 0}, // No support for future event
352 {axis: "verification", value: 1}, // Minor support for verification
353 {axis: "uncertainty", value: 0}, // Uncertainty is not supported
354 {axis: "profilng", value: 0}, // No support for user profiling
355
356 {axis: "readability", value: 2}, // The readability is low due to the completeness
    requirements. However all possible transitions can be visualised and verified
357 {axis: "debugging", value: 4}, // Incremental debugging of which hurdles are taken and
    which are not
358 {axis: "authoring", value: 4} // A graphical visualisation is provided to provide a way of
    coding and keep an overview of all transitions
359 ]
360 ];

```

Listing C.2: Multimodal frameworks

```

1 /*
2 QuickSet
3 [1] Cohen, P. R., Johnston, M., McGee, D., Oviatt, S., Pittman, J., Smith, I., Chen, L., and
    Clow, J. (1997). Quickset: Multimodal interaction for distributed applications. In
    Proceedings of the 5th international conference on Multimedia (MULTIMEDIA 1997),
    pages 31--40, Seattle, Washington, USA.
4 [2] Wu, L., Oviatt, S., and Cohen, P. (2002). From members to teams to committee -a robust
    approach to gestural and multimodal recognition. IEEE Transactions on Neural Networks,
    13(4):972--982.
5 [3] Johnston, M., Cohen, P., McGee, D., Oviatt, S., Pittman, J., and Smith, I. (1997).
    Unification-based multimodal integration. In Proceedings of the 35th Annual Meeting of
    the Association for Computational Linguistics (ACL 1997), pages 281--288, Madrid,
    Spain.
6
7 MIML
8 [10] Latoschik, M. E. (2002). Designing transition networks for multimodal vr-interactions
    using a markup language. In Proceedings of the 4th International Conference on
    Multimodal Interfaces (ICMI 2002), pages 411--447, Pittsburgh, PA, USA.
9
10 PATE
11 [20] Pflieger, N. and Schehl, J. (2006). Development of advanced dialog systems with pate. In
    Proceedings of the 9th International Conference on Spoken Language Processing (
    INTERSPEECH ICSLP 2006), Pittsburgh, PA, USA.
12
13 OpenInterface
14 [30] Serrano, M., Nigay, L., Lawson, J., Ramsay, A., Murray-Smith, R., and Deneff, S.
    (2008). The openinterface framework: A tool for multimodal interaction. In Proceedings
    of the 26th SIGCHI
15 Conference on Human Factors in Computing Systems (CHI 2008), Florence, Italy.
16 [31] Lawson, J.-Y. L., Al-Akkad, A.-A., Vanderdonckt, J., and Macq, B. (2009). 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 2009), pages 245--254, Pittsburgh, PA, USA.

```

17

18 *Squidy* (Apr)

19 [40] Koning, W., Radle, R., and Reiterer, H. (2009). *Squidy: A zoomable design environment for natural user interfaces*. In *Proc. of CHI 2009, ACM Conference on Human Factors in Computing Systems*, pages 4561-4566, Boston, MA, USA.

20

21 *HephaistTK* (Nov)

22 [50] Dumas, B., Lalanne, D., and Ingold, R. (2009a). *Hephaistk: a toolkit for rapid prototyping of multimodal interfaces*. In *Proceedings of the 11th international conference on Multimodal interfaces (ICMI-MLMI 2009)*, pages 231-232, Cambridge, MA, USA.

23 [51] Dumas, B., Lalanne, D., and Ingold, R. (2010). *Description languages for multimodal interaction: a set of guidelines and its illustration with smuiml*. *Journal on multimodal user interfaces*, 3(3):237-247.

24 [52] Dumas, B., Signer, B., and Lalanne, D. (2012). *Fusion in multimodal interactive systems : an hmm-based algorithm for user-induced adaptation*. In *Proceedings of the 4th SIGCHI symposium on Engineering interactive computing systems (EICS 2012)*, pages 15-24, Copenhagen, Denmark.

25 [53] Dumas, B., Signer, B., and Lalanne, D. (2014). *A graphical editor for the smuiml multimodal user interaction description language*. *Science of Computer Programming*, 86:30-42.

26

27 *Midas*

28 (See previous listing)

29

30 *DynaMo*

31 [70] Avouac, P.-A., Lalanda, P., and Nigay, L. (2011b). *Service-oriented autonomic multimodal interaction in a pervasive environment*. In *Proceedings of the 13th international conference on multimodal interfaces, ICMI'11*, pages 369-376, New York, NY, USA. ACM.

32 */

33

34 // Data

35 d = [

36 [// QuickSet

37 {axis: "modularisation", value: 5}, // Each description is contained in a rule

38 {axis: "composition", value: 2}, // No rule composition supported but conditions can be nested

39 {axis: "customisation", value: 3}, // Rules can be copy/pasted, modified and extended; However an incremental integer is used for variable bindings

40 {axis: "negation", value: 0}, // Not supported

41 {axis: "application", value: 2}, // One-way information to the application

42 {axis: "activation", value: 5}, // Members to teams to committee (MTC) resolution policy

43 {axis: "binding", value: 5}, // Provides unbound variables

44

45 {axis: "online", value: 1}, // Uses a backtracking algorithm, not suitable for event-driven behaviour

46 {axis: "offline", value: 5}, // Supports offline processing

47 {axis: "overlapping", value: 3}, // Overlapping matches are supported but need to be decided in every turn (MTC)

48 {axis: "segmentation", value: 1}, // Explicit segmentation is required as the approach relies on internal features

49 {axis: "synchronisation", value: 4}, // Complex temporal operators with event tracking

50 {axis: "expiration", value: 4}, // Offers a timeout feature to remove stale data and 'self-destructs' data when new input arrives

```

51     {axis: "long-term", value: 3}, // 'Edges' can be persisted to be used multiple times until
      new data (from the same kind) arrives
52     {axis: "concurrent", value: 4}, // Requires few (e.g. user check) effort to support
      concurrent interaction
53     {axis: "embeddability", value: 3}, // Rules are expressed in a graphical manner and a
      one-way application API is provided
54     {axis: "runtime", value: 1}, // No rules can be added at runtime
55     {axis: "reliability", value: 2}, // Does not limit the event input rate and is not
      sandboxed
56
57     {axis: "spatial", value: 2}, // Relies on internal features which cannot be extended
      through the language
58     {axis: "temporal", value: 3}, // Offers support for built-in temporal operators
59     {axis: "spatio-temporal", value: 4}, // Spatial and temporal operators can be combined
60     {axis: "identification", value: 4}, // Uses logical binds to ease the identification problem
      and inherently supports the grouping problem
61     {axis: "prioritisation", value: 0}, // No support for priorities
62     {axis: "prediction", value: 0}, // No support for future events
63     {axis: "verification", value: 3}, // Embeds other classifiers
64     {axis: "uncertainty", value: 4}, // Handled by MTC but limited to ML classifiers (no
      rules)
65     {axis: "profiling", value: 0}, // Does not offer user profiling tools
66
67     {axis: "readability", value: 4}, // Few keywords and provides a graphical layout
68     {axis: "debugging", value: 1}, // Offers no special debugging support
69     {axis: "authoring", value: 1} // Rules are created in a graphical manner but no constructs
      are provided to integrate with sample data
70 ],
71
72 [ // MIML
73     {axis: "modularisation", value: 5}, // Separates all descriptions
74     {axis: "composition", value: 1}, // Descriptions are nameless and cannot be composed.
      However, by escaping to functions a form of composition might be possible
75     {axis: "customisation", value: 3}, // Can be modified and extended through external
      functions. However, reordering of conditions is problematic
76     {axis: "negation", value: 4}, // Supports negation
77     {axis: "application", value: 1}, // No application integration, except to call functions
78     {axis: "activation", value: 0}, // No special activation policies
79     {axis: "binding", value: 2}, // Variables allowed, but need to be manually filled. These
      variables can only contain a single value for the entire program
80
81     {axis: "online", value: 0}, // No support for online processing
82     {axis: "offline", value: 5}, // Supports offline processing
83     {axis: "overlapping", value: 0}, // No overlapping matching
84     {axis: "segmentation", value: 0}, // Explicit begin and end points are required
85     {axis: "synchronisation", value: 0}, // No support provided by the framework
86     {axis: "expiration", value: 1}, // Single state variables (automated discarding
87     {axis: "long-term", value: 4}, // Access to historical data is possible
88     {axis: "concurrent", value: 3}, // Uses multiple tokens
89     {axis: "embeddability", value: 4}, // Embeddable engine with descriptions provided in
      XML syntax
90     {axis: "runtime", value: 2}, // Probably not supported, unclear
91     {axis: "reliability", value: 3}, // Single value slots, but support for concurrent tokens
92
93     {axis: "spatial", value: 2}, // No spatial operations provided and one needs to escape to
      the host language

```

```

94     {axis: "temporal", value: 4}, // Advanced combinations of temporal operations are possible
      but difficult to encode
95     {axis: "spatio-temporal", value: 3}, // Nesting conditions requires careful design
96     {axis: "identification", value: 1}, // No identification and grouping support except for
      concurrent input
97     {axis: "prioritisation", value: 0}, // No prioritisation support
98     {axis: "prediction", value: 0}, // No future event support
99     {axis: "verification", value: 0}, // No verification support
100    {axis: "uncertainty", value: 1}, // No uncertainty support, but escape to host language is
      possible
101    {axis: "profiling", value: 0}, // No user profiling support
102
103    {axis: "readability", value: 2}, // A lot of syntax characters and keywords
104    {axis: "debugging", value: 1}, // No support for additional debugging
105    {axis: "authoring", value: 0} // No support for editors
106  ],
107
108  [ // PATE
109    {axis: "modularisation", value: 5}, // Separates all rules
110    {axis: "composition", value: 1}, // No composition supported and need to escape to the
      host language
111    {axis: "customisation", value: 3}, // Descriptions can be copy/pasted, modified and
      extended. The order of conditions does not matter
112    {axis: "negation", value: 0}, // No negation support
113    {axis: "application", value: 1}, // No application integration, except to call functions
114    {axis: "activation", value: 3}, // Turn-based activation policy
115    {axis: "binding", value: 5}, // Unbound variables supported
116
117    {axis: "online", value: 0}, // No support for online processing
118    {axis: "offline", value: 5}, // Supports offline processing
119    {axis: "overlapping", value: 4}, // Supports overlapping descriptions but does not inform
      the developer
120    {axis: "segmentation", value: 2}, // Explicit begin and end points are not required, but no
      abstractions are provided to aid the segmentation process
121    {axis: "synchronisation", value: 2}, // Realignment of streams can happen automatically
      but requires proper use of temporal operators (and extensions)
122    {axis: "expiration", value: 2}, // Events discarded at each turn
123    {axis: "long-term", value: 1}, // No dedicated support for long-term reasoning
124    {axis: "concurrent", value: 3}, // Concurrent interaction possible when manually
      separated
125    {axis: "embeddability", value: 5}, // XML Syntax and cross-platform
126    {axis: "runtime", value: 2}, // Single configuration file, but runtime addition of rules is
      probably possible (unclear from the paper)
127    {axis: "reliability", value: 1}, // No reliability aspects
128
129    {axis: "spatial", value: 1}, // No spatial operators provided but can be extended (outside
      the language)
130    {axis: "temporal", value: 4}, // Temporal expressions are provided and can be extended (
      outside the language syntax)
131    {axis: "spatio-temporal", value: 3}, // Spatial and temporal conditions can be combined
      but no internal features are provided
132    {axis: "identification", value: 4}, // Identification and grouping support through
      unification
133    {axis: "prioritisation", value: 3}, // [0..1] range prioritisation levels
134    {axis: "prediction", value: 0}, // No future event support
135    {axis: "verification", value: 0}, // No verification support

```

```

136  {axis: "uncertainty", value: 0}, // No uncertainty support
137  {axis: "profiling", value: 0}, // No user profiling support
138
139  {axis: "readability", value: 4}, // Clear and simple constructs, but uses verbose XML
      syntax
140  {axis: "debugging", value: 5}, // Advanced online debugging with editor support
141  {axis: "authoring", value: 4} // Editor support with external representation
142 ],
143
144 [ // OpenInterface
145  {axis: "modularisation", value: 5}, // Separates implementation in composition boxes
146  {axis: "composition", value: 4}, // Boxes can be composed through pipelining
147  {axis: "customisation", value: 3}, // Routes can be redirected
148  {axis: "negation", value: 3}, // Negation is supported (typically used as an inversion
      function)
149  {axis: "application", value: 2}, // Supported when the application delivers information as
      a stream (or interweaves in a composition box)
150  {axis: "activation", value: 2}, // Each box can be configured autonomously
151  {axis: "binding", value: 0}, // No unbound variables supported
152
153  {axis: "online", value: 5}, // All data is processed in an online fashion. Rather difficult to
      reason over a period of time
154  {axis: "offline", value: 1}, // Manual state management to gather data between begin-
      and end- events
155  {axis: "overlapping", value: 2}, // Supports overlapping matching but requires duplication
      of data (i.e. splitting pipes)
156  {axis: "segmentation", value: 2}, // Poor segmentation support as all logic needs to be
      coded in an imperative host language
157  {axis: "synchronisation", value: 3}, // Pipes can be manually delayed based on time or
      input from another source
158  {axis: "expiration", value: 3}, // Events expire as they are discarded by each step in the
      pipeline.
159  {axis: "long-term", value: 3}, // Persistency can be incorporated (and queried) but no
      details are presented in the paper
160  {axis: "concurrent", value: 1}, // Concurrent interaction need to be split on the event
      level and pipelines need to be replicated manually
161  {axis: "embeddability", value: 4}, // Typically installed between the input source and the
      application as a C++ daemon. Provides extensive TCP/IP support
162  {axis: "runtime", value: 3}, // Redirection of pipes is possible at runtime but no API/
      language abstractions are provided to handle this properly
163  {axis: "reliability", value: 4}, // Pass-through semantics with the potential of limiting
      event rates
164
165  {axis: "spatial", value: 2}, // Ad-hoc implementation of spatial operators. Few threshold
      and averaging filters are provided
166  {axis: "temporal", value: 2}, // No abstractions for temporal logic, each event is handled
      one by one
167  {axis: "spatio-temporal", value: 1}, // No abstractions to express spatio-temporal
      relations
168  {axis: "identification", value: 1}, // No abstractions to perform identification (unless
      ad hoc)
169  {axis: "prioritisation", value: 0}, // No details provided on prioritisation
170  {axis: "prediction", value: 1}, // A 'delay' feature that expresses the time between events,
      but waiting for future knowledge is not supported on the language level
171  {axis: "verification", value: 1}, // Output can be verified using multiple boxes, however
      no details are provided in the paper

```

```

172   {axis: "uncertainty", value: 1}, // Potentially event-based uncertainty handling, no
      details provided
173   {axis: "profiling", value: 0}, // No user profiling support
174
175   {axis: "readability", value: 3}, // For data-level fusion tasks the pipelines are typically
      easy to follow
176   {axis: "debugging", value: 1}, // No support for additional debugging
177   {axis: "authoring", value: 3} // Graphical authoring of pipelines, but without a focus on
      particular modalities
178 ],
179
180 [ // Squidy
181   {axis: "modularisation", value: 5}, // Separates all descriptions
182   {axis: "composition", value: 4}, // Boxes can be composed through pipelining
183   {axis: "customisation", value: 2}, // Routes can be redirected, but extensibility is limited
      to the provided boxes
184   {axis: "negation", value: 0}, // Negation is not supported
185   {axis: "application", value: 1}, // No integration with the application
186   {axis: "activation", value: 2}, // Each box can be configured autonomously
187   {axis: "binding", value: 0}, // No unbound variables supported
188
189   {axis: "online", value: 5}, // All data is processed in an online fashion. Rather difficult to
      reason over a period of time
190   {axis: "offline", value: 1}, // Manual state management to gather data between begin-
      and end- events
191   {axis: "overlapping", value: 2}, // Supports overlapping matching but requires duplication
      of data (i.e. splitting pipes)
192   {axis: "segmentation", value: 2}, // Poor segmentation support, limited to threshold filters
      and not extensible
193   {axis: "synchronisation", value: 1}, // No details in the paper but potentially enabled by
      a delay box with a slack parameter
194   {axis: "expiration", value: 3}, // Events expire as they are discarded by each step in the
      pipeline.
195   {axis: "long-term", value: 0}, // No support for long-term reasoning
196   {axis: "concurrent", value: 1}, // Concurrent interaction need to be split on the event
      level and pipelines need to be replicated manually
197   {axis: "embeddability", value: 2}, // Provides OSC support. No details on the description
      format
198   {axis: "runtime", value: 2}, // Redirection of pipes is potentially possible at runtime
199   {axis: "reliability", value: 3}, // Pass-through semantics with the potential of limiting
      event rates (no details in the paper)
200
201   {axis: "spatial", value: 1}, // No abstractions for spatial relations
202   {axis: "temporal", value: 1}, // No abstractions for temporal logic, each event is handled
      one by one
203   {axis: "spatio-temporal", value: 1}, // No abstractions to express spatio-temporal
      relations
204   {axis: "identification", value: 1}, // No abstractions to perform identification (unless
      ad hoc)
205   {axis: "prioritisation", value: 0}, // No details provided on prioritisation
206   {axis: "prediction", value: 0}, // Waiting for future knowledge is not supported on the
      language level
207   {axis: "verification", value: 1}, // Output can be verified using multiple boxes, however
      no details are provided in the paper
208   {axis: "uncertainty", value: 1}, // Potentially event-based uncertainty handling, no
      details provided

```

```

209     {axis: "profiling", value: 0}, // No user profiling support
210
211     {axis: "readability", value: 3}, // For data-level fusion tasks the pipelines are typically
      easy to follow. Ad hoc code is written in Java
212     {axis: "debugging", value: 4}, // Visual aid to specify properties of composition boxes,
      with an option to query historical data
213     {axis: "authoring", value: 4} // Graphical authoring tool for pipelines with additional aid
      to specify properties
214 ],
215
216 [ // HephaisTK
217     {axis: "modularisation", value: 5}, // Multimodal descriptions are separated
218     {axis: "composition", value: 0}, // No support for composition
219     {axis: "customisation", value: 2}, // Poor customisation support, limited to internal
      triggers and actions
220     {axis: "application", value: 2}, // Dialogue management should be specified in the
      SMUIML language. No synchronisation with the application level is provided
221     {axis: "activation", value: 3}, // Integration committee toggles activations
222     {axis: "binding", value: 0}, // Variable bindings are support in a manual manner
223
224     {axis: "negation", value: 0}, // Negation is not supported
225     {axis: "online", value: 2}, // Triggers can be embedded within transitions
226     {axis: "offline", value: 3}, // Triggers and results are omitted when conditions are met
227     {axis: "overlapping", value: 2}, // Input can be reused across descriptions but the
      committee needs to decide on an per event basis
228     {axis: "segmentation", value: 0}, // No support for segmentation
229     {axis: "synchronisation", value: 2}, // Dialogs provide a lead-time with a slack
      parameter
230     {axis: "expiration", value: 3}, // Events expire as they are overwritten by new events
231     {axis: "long-term", value: 0}, // No support for long-term reasoning
232     {axis: "concurrent", value: 0}, // No support for concurrent interaction
233     {axis: "embeddability", value: 3}, // Interweaving of specifications with the host language
      is required. Cross-platform deployment is possible
234     {axis: "runtime", value: 2}, // Additional descriptions can be added at runtime, but
      updating or removing is not possible
235     {axis: "reliability", value: 3}, // Does not limit the event input rate, but sandboxed
236
237     {axis: "spatial", value: 0}, // No abstractions for spatial operators
238     {axis: "temporal", value: 4}, // Few built-in temporal operations with can be nested
239     {axis: "spatio-temporal", value: 2}, // No spatio-temporal operators but conditions can
      be interweaved with temporal information
240     {axis: "identification", value: 0}, // No support for expressing identification and
      grouping
241     {axis: "prioritisation", value: 1}, // Integration committee can prioritise descriptions,
      but this is outside the scope of the language
242     {axis: "prediction", value: 2}, // Requires to set a waiting time for each event, but
      supports a compatibility mechanism
243     {axis: "verification", value: 0}, // No verification support
244     {axis: "uncertainty", value: 4}, // HMM-based extension allows dealing with uncertain
      events and uncertain results
245     {axis: "profiling", value: 4}, // User profiling provided with an HMM-based extension
246
247     {axis: "readability", value: 3}, // Deep nesting of constructs using a verbose XML
      syntax
248     {axis: "debugging", value: 4}, // Interactive visualisation with live support
249     {axis: "authoring", value: 3} // Graphical definition is possible (in a limited way)

```

```

250 ],
251
252 [ // Midas
253   {axis: "modularisation", value: 5}, // Each description is contained in a rule
254   {axis: "composition", value: 4}, // Descriptions can easily be composed by relying on high-
      level facts, attempts, computed facts and module inheritance
255   {axis: "customisation", value: 4}, // Rules can be copy/pasted, modified and extended; '
      Attempts' further enable this via parameters.
256   {axis: "negation", value: 5}, // Supports complex negation, with the possibility to wait for
      future events
257   {axis: "application", value: 4}, // Supports shadow facts to synchronise application info
258   {axis: "activation", value: 2}, // Bookkeeping facts (which are cumbersome but powerful)
      and activation flags (easy but limited)
259   {axis: "binding", value: 5}, // Provides unbound variables
260
261   {axis: "online", value: 4}, // Supports online processing through alternation of conditions
      and modifiers or intermediate progress notifications
262   {axis: "offline", value: 4}, // Supports offline processing although explicit state checks
      are required
263   {axis: "overlapping", value: 3}, // Inherently deals with overlapping matching in the
      processing engine but does not inform the developer
264   {axis: "segmentation", value: 5}, // Optimises for recall to optimally segment candidates
265   {axis: "synchronisation", value: 5}, // Out-or-order, realignment and synchronised
      expiration supported
266   {axis: "expiration", value: 3}, // Offers a template-local relative time event expiration
      mechanism as well as bounded size. However, this needs to be set manually
267   {axis: "long-term", value: 3}, // Offers persistency and DRAM containers to ease long-
      term reasoning. However, this needs to be handled manually
268   {axis: "concurrent", value: 4}, // Requires few effort (i.e. user grouping) to support
      concurrent interaction
269   {axis: "embeddability", value: 3}, // Can be embedded in all kinds of applications (C
      library and networking APIs exists) but requires rules to be developed as strings
270   {axis: "runtime", value: 4}, // Supports adding rules at runtime but does not have an
      automated refinement
271   {axis: "reliability", value: 4}, // Does not limit the event input rate but is sandboxed.
      Extensions have shown to scale in terms of performance to 64 cores however the
      approach is less flexible (no fusion across levels)
272
273   {axis: "spatial", value: 5}, // Offers extensive support for spatial operators
274   {axis: "temporal", value: 5}, // Offers extensive support for temporal operators
275   {axis: "spatio-temporal", value: 5}, // Spatio-temporal features and nesting of spatial
      and temporal operators is not restricted by the engine
276   {axis: "identification", value: 3}, // Uses logical binds to ease the identification problem
      and inherently supports the grouping problem. However, this results into multiple
      candidates that need to be verified later
277   {axis: "prioritisation", value: 2}, // Offers rule-specific prioritisation and uses
      bookkeeping facts to enable fusion. However it is hard to express a conflict resolution
      between multiple rules
278   {axis: "prediction", value: 2}, // An additional condition implies waiting for future
      events. Furthermore, developers can write delay statements but it not easy to manage
279   {axis: "verification", value: 3}, // Embeds other classifiers to verify candidates
280   {axis: "uncertainty", value: 1}, // Uncertainty can be part of a fact attribute but requires
      additional effort
281   {axis: "profiling", value: 1}, // Does not offer user profiling tools, but does provide a
      manual interface to access and store historical data
282

```

```

283     {axis: "readability", value: 2}, // Kammer[22] measured a low readability of Midas 1.0.
      Midas 2.0 abstracts a LISP-based syntax with a Ruby-style flavour
284     {axis: "debugging", value: 2}, // Offers no special debugging support
285     {axis: "authoring", value: 3} // An visual editor exists both for multi-touch and for full-
      body gestures
286 ],
287
288 [ // DynaMo
289     {axis: "modularisation", value: 5}, // Focus on proxy models
290     {axis: "composition", value: 4}, // Focus on device and output abstraction with the ability
      to use alternatives on the fly. Thereby providing many layers of composition
291     {axis: "customisation", value: 3}, // Outsources customisation purposes to external tools
      based on a Java implementation
292     {axis: "application", value: 4}, // Two-way communication between application and
      fusion framework. Requires manual linking
293     {axis: "activation", value: 4}, // Uses an autonomic manager and administration layer
      to make decisions. However no API/language abstractions are provided in the paper
294     {axis: "binding", value: 0}, // No (unbound) variables supported
295
296     {axis: "negation", value: 0}, // Negation is not supported in their models
297     {axis: "online", value: 2}, // Potentially possible to trigger in the middle of an execution,
      but unclear from paper. Requires ad hoc implementation in Java
298     {axis: "offline", value: 3}, // The fusion process is outsourced to native services thereby
      relying on processed data for offline decisions
299     {axis: "overlapping", value: 2}, // Supports overlapping fusion but does not inform the
      developer
300     {axis: "segmentation", value: 1}, // No segmentation support except when outsourced to
      services
301     {axis: "synchronisation", value: 2}, // Mentioned as a concern in the paper but unclear
      how it works
302     {axis: "expiration", value: 3}, // Events expire are discarded by the following event
303     {axis: "long-term", value: 0}, // No support for long-term reasoning
304     {axis: "concurrent", value: 0}, // No support for concurrent interaction
305     {axis: "embeddability", value: 4}, // Deep integration based on OSGi for Java
306     {axis: "runtime", value: 5}, // DynaMo focuses on runtime adaptation and discovery
307     {axis: "reliability", value: 5}, // Does not limit the event input rate. Covers networking
      problems with dynamic adaptation
308
309     {axis: "spatial", value: 0}, // No spatial abstractions provided
310     {axis: "temporal", value: 0}, // No temporal abstractions provided
311     {axis: "spatio-temporal", value: 0}, // No spatio-temporal abstractions provided
312     {axis: "identification", value: 0}, // No identification and grouping support
313     {axis: "prioritisation", value: 4}, // Prioritisation on fusion and fission level to provide
      the best combination
314     {axis: "prediction", value: 0}, // No future event support
315     {axis: "verification", value: 0}, // No verification support
316     {axis: "uncertainty", value: 0}, // No uncertainty support
317     {axis: "profiling", value: 0}, // No user profiling support
318
319     {axis: "readability", value: 3}, // Graphical layout with illustrations and connected
      actions. XML-base coding of proxy models
320     {axis: "debugging", value: 4}, // Incremental debugging
321     {axis: "authoring", value: 4} // Visual editor to link input sources with actions
322 ]
323 ];

```

D

ANTLR Specification of Midas

Listing D.1: ANTLR 4 specification of Midas

```
1 grammar Midas2;
2
3 @header {
4   package midas2;
5 }
6
7 TEMPLATE: 'template';
8 FUNCTION: 'function';
9 ATTEMPT: 'attempt';
10 MODULE: 'module';
11 INCLUDE: 'include';
12 RULE: 'rule';
13 TEST: 'test';
14 ASSERT: 'assert';
15 MODIFY: 'modify';
16 RETRACT: 'retract';
17 DISPLAY: 'display';
18 PRINTOUT: 'printout';
19 CALL: 'call';
20 SELF: 'self';
21 SUPER: 'super';
22 NO: 'no';
23 ASYNC: 'async';
```

```
24 WAIT: 'wait';
25 GROUP: 'group';
26 SALIENCE: 'salience';
27 RETURN: 'return';
28 WITH: 'with';
29 NIL: 'nil';
30 TYPE_INTEGER: 'int';
31 TYPE_FLOAT: 'float';
32 TYPE_STRING: 'string';
33 BECOMES: '=>';
34 APPL_ATTEMPT: '<-';
35 NEQUALS: '!=';
36 EQUALS: '==';
37 PIPE: '|';
38 AMP: '&';
39 TILDE: '~';
40 DOLLAR: '$';
41 LT: '<';
42 LTE: '<=';
43 GT: '>';
44 GTE: '>=';
45 PLUS: '+';
46 MINUS: '-';
47 TIMES: '*';
48 EXP: '**';
49 DIV: '/';
50 BINDS: '=';
51 CHECK: 'check';
52 END: 'end';
53 LPAREN: '(';
54 RPAREN: ')';
55 QUESTION: '?';
56 FIRST: 'first';
57 LAST: 'last';
58 DOT: '.';
59
60 /* Reserved keywords for Ruby */
61 OBJECT: 'Object';
62 CLASS: 'Class';
63 DEF: 'def';
64 NOT: 'not';
65
66 program
67   : entity* EOF
68   ;
69
70 entity
```

```
71 : declaration | constant | classe | module | function
    | attempt | rulee | modifier | global_function_call
72 ;
73
74 declaration
75 : (DOLLAR l_name BINDS)? CALL? c_name DOT L_ID
    arguments?
76 ;
77
78 classe
79 : TEMPLATE c_name inheritance? open_block include*
    class_field* (attempt|function)* close_block
80 ;
81
82 module
83 : MODULE c_name open_block include* class_field* (
    attempt|function)* close_block
84 ;
85
86 inheritance
87 : LT c_name
88 ;
89
90 include
91 : INCLUDE c_name (',' c_name)*
92 ;
93
94 class_field
95 : class_field_multiple
96 | class_field_single
97 | constant
98 ;
99
100 class_field_multiple
101 : type? l_name (',' l_name)+
102 ;
103
104 class_field_single
105 : type? l_name primitive_value?
106 ;
107
108 constant
109 : type? C_ID BINDS expression
110 ;
111
112 function
113 : FUNCTION (SELF DOT)? l_name parameters? open_block
    expression+ close_block
```

```
114 ;
115
116 parameters
117   : LPAREN parameter (',' parameter)* RPAREN
118   ;
119
120 parameter
121   : l_name (BINDS primitive_value)?
122   ;
123
124 attempt
125   : ATTEMPT (SELF APPL_ATTEMPT)? l_name parameters?
126     open_block attempt_condition* attempt_return?
127     close_block
128   ;
129
130 attempt_condition
131   : condition
132   | (l_name BINDS)? computed_fact
133   ;
134
135 attempt_return
136   : returne
137   | RETURN computed_fact
138   ;
139
140 rulee
141   : RULE l_name open_block rulee_declarations*
142     rulee_statement+ close_block
143   ;
144
145 rulee_declarations
146   : SALIENCE INT
147   | GROUP l_name
148   ;
149
150 rulee_statement
151   : condition | modifier
152   ;
153
154 condition
155   : conditional_element | test | bind | special_block
156   ;
157
158 conditional_element
159   : c_name literal_constraint?
160   | c_name DOT l_name value
161   ;
```

```
159
160 bind
161   : l_name BINDS (conditional_element | expression |
162     attempt_inline | array)
163   ;
164 test
165   : TEST? expression (LT|LTE|EQUALS|NEQUALS|GTE|GT)
166     expression
167   | TEST? attempt_inline
168   ;
169 expression
170   : special_form
171   | returne
172   | infix_expression
173   | MINUS? function_call
174   | LPAREN MINUS? expression RPAREN
175   | l_name BINDS expression
176   | MINUS? value
177   ;
178
179 special_form
180   : 'if' LPAREN expression RPAREN then_clause
181     elsif_clause* else_clause?
182   ;
183 then_clause
184   : '{' expression* '}'
185   ;
186
187 elsif_clause
188   : 'elif' expression '{' expression* '}'
189   ;
190
191 else_clause
192   : 'else' '{' expression* '}'
193   ;
194
195 global_function_call
196   : l_name ( DOT l_name )? arguments?
197   | c_name DOT l_name arguments?
198   ;
199
200 function_call
201   : l_name ( DOT l_name )? arguments?
202   | c_name DOT l_name arguments?
203   | SUPER DOT l_name arguments?
```

```
204 | SELF DOT l_name arguments?
205 | DOLLAR l_name ( DOT l_name )? arguments?
206 ;
207
208 function_call_paren
209 : l_name ( DOT l_name )? arguments_paren?
210 | c_name DOT l_name arguments_paren?
211 | SUPER DOT l_name arguments_paren?
212 | SELF DOT l_name arguments_paren?
213 | DOLLAR l_name ( DOT l_name )? arguments_paren?
214 ;
215
216 attempt_inline
217 : l_name ( APPL_ATTEMPT l_name )? arguments?
218 | c_name APPL_ATTEMPT l_name arguments?
219 | SUPER APPL_ATTEMPT l_name arguments?
220 | SELF APPL_ATTEMPT l_name arguments?
221 ;
222
223 infix_expression
224 : infix_expression1
225 ;
226
227 infix_expression1
228 : infix_expression2 (infix_operator1 infix_expression2
229   )*
230 ;
231 infix_operator1
232 : PLUS | MINUS
233 ;
234
235 infix_expression2
236 : infix_expression3 (infix_operator2 infix_expression3
237   )*
238 ;
239 infix_operator2
240 : TIMES | DIV
241 ;
242
243 infix_expression3
244 : infix_expression4 (infix_operator3 infix_expression4
245   )*
246 ;
247 infix_operator3
248 : EXP
```

```
249 ;
250
251 infix_expression4
252 : LPAREN expression RPAREN
253 | value
254 | function_call_paren
255 ;
256
257 computed_fact
258 : attribute_values computed_fact_extends?
259 ;
260
261 computed_fact_extends
262 : WITH '[' c_name (',' c_name )* ']'
263 ;
264
265 returne
266 : RETURN expression
267 ;
268
269 arguments
270 : expression (',' expression)*
271 | LPAREN expression (',' expression)* RPAREN
272 ;
273
274 arguments_paren
275 : LPAREN expression (',' expression)* RPAREN
276 ;
277
278 literal_constraint
279 : '{' constraint_inline (',' constraint_inline)* '}'
280 ;
281
282 modifier
283 : ASSERT c_name attribute_values?
284 | MODIFY l_name attribute_values
285 | RETRACT l_name (',' l_name)*
286 | DISPLAY arguments
287 | PRINTOUT arguments
288 | (l_name BINDS)? CALL function_call
289 ;
290
291 special_block
292 : negation_block
293 | async_block
294 | wait_block
295 ;
296
```

```
297 negation_block
298   : (NO|NOT) open_block condition+ close_block
299   ;
300
301 async_block
302   : ASYNC DOLLAR attempt_inline
303   | ASYNC open_block expression+ close_block
304   ;
305
306 wait_block
307   : WAIT variable ',' primitive_number
308   ;
309
310 attribute_values
311   : '{' l_name BECOMES expression (',' l_name BECOMES
312     expression)* '}'
313   ;
314
314 constraint_inline
315   : l_name (EQUALS|NEQUALS) constraint_value
316   ;
317
318 constraint_value
319   : value PIPE PIPE? constraint_inline
320   | value AMP AMP? constraint_inline
321   | TILDE value
322   | value
323   ;
324
325 value
326   : MINUS? logical_variable
327   | MINUS? primitive_value
328   | array
329   | MINUS? array_value
330   | MINUS? variable
331   | MINUS? (c_name DOT)? C_ID
332   | MINUS? SELF
333   ;
334
335 variable
336   : slot_variable
337   | array_variable
338   | l_name
339   ;
340
341 logical_variable
342   : QUESTION l_name
343   ;
```

```
344
345 slot_variable
346   : l_name DOT l_name
347   ;
348
349 array_variable
350   : (l_name|array) DOT (FIRST|LAST) DOT l_name
351   | (l_name|array) '[' MINUS? INT ']' DOT l_name
352   ;
353
354 array
355   : '[' value (',' value)* ']'
356   ;
357
358 array_value
359   : l_name '[' MINUS? INT ']'
360   ;
361
362 primitive_value
363   : primitive_number | primitive_string | NIL
364   ;
365
366 primitive_number
367   : MINUS? (INT|FLOAT) (DOT (TIME_HELPER|SPACE_HELPER))
368     ?
369   ;
370
371 primitive_string
372   : STRING | SYMBOL
373   ;
374
375 type
376   : TYPE_INTEGER | TYPE_FLOAT | TYPE_STRING
377   ;
378
379 open_block
380   : ('{' | CHECK)?
381   ;
382
383 close_block
384   : '}' | END
385   ;
386
387 c_name
388   : C_ID
389   ;
390
391 l_name
```

```

391 : L_ID
392 | TIME_HELPER | SPACE_HELPER
393 | FIRST | LAST
394 | FUNCTION
395 ;
396
397 TIME_HELPER
398 : 'd'|'day'|"days"|"h"|"hour"|"hours"|"m"|"min"|"
    minute"|"minutes"
399 | 's'|"second"|"seconds"|"ms"|"millisecond"|"
    milliseconds"
400 ;
401
402 SPACE_HELPER
403 : 'dam'|"decameter"|"decameters"|"meter"|"meters"|"dm"
    |"decimeter"|"decimeters"
404 | 'cm'|"centimeter"|"centimeters"|"mm"|"millimeter"|"
    millimeters"
405 | 'px'|"em"|"pt"|"in"|"ex"|"pc"|"rem"
406 ;
407
408 C_ID
409 : ('A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* (':'('a
    '..'z'|'A'..'Z'|'0'..'9'|'_')+)*
410 ;
411
412 L_ID
413 : ('a'..'z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* (':
    ('a'..'z'|'A'..'Z'|'0'..'9'|'_')+)*
414 ;
415
416 INT
417 : ('0'..'9')+
418 ;
419
420 FLOAT
421 : ('0'..'9')+ (DOT ('0'..'9')+)? EXPONENT?
422 | DOT ('0'..'9')+ EXPONENT?
423 | ('0'..'9')+ EXPONENT
424 ;
425
426 BLOCK_COMMENT
427 : '/*' .*? '*/' -> channel(HIDDEN)
428 ;
429
430 LINE_COMMENT
431 : ('//'|'#') ~[\r\n]* -> channel(HIDDEN)
432 ;

```

```
433
434 WS
435   : [ \t\r\n\f]+ -> channel(HIDDEN)
436   ;
437
438 SYMBOL
439   : ':' (C_ID | L_ID | TIME_HELPER | SPACE_HELPER |
440         FIRST | LAST | TIMES)
441   ;
442 STRING
443   : '"' ( ESC_SEQ | ~('\\"'|"'') )* '"'
444   ;
445
446 CHAR: "'" ( ESC_SEQ | ~('"'|'\\') ) "'"
447   ;
448
449 fragment
450 EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;
451
452 fragment
453 HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;
454
455 fragment
456 ESC_SEQ
457   : '\\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
458   | UNICODE_ESC
459   | OCTAL_ESC
460   ;
461
462 fragment
463 OCTAL_ESC
464   : '\\\' ('0'..'3') ('0'..'7') ('0'..'7')
465   | '\\\' ('0'..'7') ('0'..'7')
466   | '\\\' ('0'..'7')
467   ;
468
469 fragment
470 UNICODE_ESC
471   : '\\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
472   ;
```

E

Reused Attempts and Functions

Listing E.1: Reusable attempts and functions

```
1 module Time
2   time
3   attempt beforeF(f, eps = 0)
4     time + eps < f.time
5   end
6   attempt afterF(f, eps = 0)
7     time + eps > f.time
8   end
9   attempt withinF(f, min, max)
10    f.time + min < time
11    f.time + max > time
12  end
13 end
14 module TimeInterval
15   time_begin, time_end
16   attempt duringF(f)
17     time_begin < time_end
18     f.time_begin < f.time_end
19     f.time_begin < time_begin
20     f.time_end > time_end
21   end
22 end
23 module Space2D
24   x, y
25   attempt self←align(x1, y1, x2, y2, x_diff, y_diff)
26     equal x1, x2, x_diff
27     equal y1, y2, y_diff
```

```
28 end
29 function self.euclidean_distance(x1, y1, x2, y2)
30   Math.sqrt ((x2-x1)**2) + ((y2-y1)**2)
31 end
32 function euclidean_distance(x2, y2)
33   Space2D.euclidean_distance x, y, x2, y2
34 end
35 attempt self←near(x1, y1, x2, y2, r)
36   r > euclidean_distance x1, y1, x2, y2
37 end
38 attempt near(x2, y2, r)
39   Space2D←near x, y, x2, y2, r
40 end
41 attempt self←away(x1, y1, x2, y2, r)
42   r < euclidean_distance x1, y1, x2, y2
43 end
44 attempt away(x2, y2, r)
45   Space2D←away x, y, x2, y2, r
46 end
47 attempt awayF(f, r)
48   Space2D←away x, y, f.x, f.y, r
49 end
50 attempt self←translated_near(x1, y1, x2, y2, x_offset, y_offset, r)
51   near x2, y2, x1 + x_offset, y1 + y_offset, r
52 end
53 attempt translated_near(x2, y2, x_offset, y_offset, r)
54   Space2D←translated_near(x, y, x2, y2, x_offset, y_offset, r)
55 end
56 attempt translated_nearF(f, x_offset, y_offset, r)
57   Space2D←translated_near(x, y, f.x, f.y, x_offset, y_offset, r)
58 end
59 end
60 module Space2DInterval
61   include Space2D
62   x_begin, x_end
63   y_begin, y_end
64   attempt near_beginF(f, r)
65     Space2D←near x_begin, y_begin, f.x_begin, f.y_begin, r
66   end
67   attempt align_beginF(f, x_diff, y_diff)
68     Space2D←align x_begin, y_begin, f.x_begin, f.y_begin, x_diff, y_diff
69   end
70 end
```

F

Built-in Mudra Templates

Listing F.1: Built-in Mudra templates

```
1 template Joint
2   sensor,user,joint,x,y,z,confidence
3   HEAD = 1
4   NECK = 2
5   TORSO = 3
6   WAIST = 4
7   COLLAR_LEFT = 5
8   SHOULDER_LEFT = 6
9   ELBOW_LEFT = 7
10  WRIST_LEFT = 8
11  HAND_LEFT = 9
12  FINGERTIP_LEFT = 10
13  COLLAR_RIGHT = 11
14  SHOULDER_RIGHT = 12
15  ELBOW_RIGHT = 13
16  WRIST_RIGHT = 14
17  HAND_RIGHT = 15
18  FINGERTIP_RIGHT = 16
19  HIP_LEFT = 17
20  KNEE_LEFT = 18
21  ANKLE_LEFT = 19
22  FOOT_LEFT = 20
23  HIP_RIGHT = 21
24  KNEE_RIGHT = 22
25  ANKLE_RIGHT = 23
26  FOOT_RIGHT = 24
27 end
28 template RelativeJoint
29   sensor,user,index,parent_muid,parent,parent_time,child_muid,child,child_time
30   x,y,z,time,distance,distance_x,distance_y,distance_z
```

```
31  previous_muid,previous_time,previous_x,previous_y,previous_z
32  end
33  template Hand
34  sensor,hand,x,y,z,confidence
35  end
36  template Face
37  sensor,user,x,y,z,pitch,yaw,roll
38  end
39  template FaceAnimation
40  sensor,user,lip_raise,lip_stretcher,lip_corner_depressor,jaw_lower,brow_lower,brow_raise
41  end
42  template Tuio2DCur
43  id,i,state,x,y,vx,vy,m
44  end
45  template Tuio2DObj
46  id,i,state,x,y,a,vx,vy,ra,m,r
47  end
48  template Touch2D
49  DOWN = 1
50  MOVE = 2
51  UP = 3
52  end
53  template Accelerometer
54  x # The x acceleration
55  y # The y acceleration
56  z # The z acceleration
57  sensor # Sensor providing the value
58  end
59  template Speech
60  word # uttered word(s)
61  probability # probability value [0..1]
62  user # unique identifier for the user
63  origin # name of the voice recogniser
64  end
```

G

Compatibility of Criteria Defined by Cirelli et al.

Cirelli et al. recently conducted a survey on multi-touch gesture recognition and multi-touch frameworks. As shown in Figure G.1, they analysed related data-level frameworks on 14 criteria [26]. Their criteria focus on the user of a framework, while our criteria analyse fine-grained functionality provided by a particular framework. In the following, we show that these 14 criteria are compatible with the criteria we defined in Chapter 2:

1. **Be Flexible and Extensible** This criterium analyses whether developers can easily extend the existing gesture set with a new gesture. This corresponds to the modularisation, composition and customisation.
2. **Be Fast** In order to reduce the recognition latency, algorithms “must be fast”. In our work this corresponds to the reliability and online processing criteria.
3. **Be accurate** In the survey, this criteria analyses the potential for false positives. In our work, we discuss the fundamental trade-off between recall and precision.
4. **Support Multi-Touch** Multi-touch gestures rely on the simultaneous input from multiple fingers. Therefore, the detection of

Table 3: Frameworks and Recognizers Analysis

Criteria	Midas	Proton++	\$N-Protractor	MT4J
1	●	●	●	●
2	●	—	●	—
3	—	—	●	—
4	●	●	●	●
5	●	○	○	○
6	○	○	●	○
7	●	●	○	●
8	○	●	●	○
9	●	○	●	○
10	●	●	○	●
11	●	○	○	●
12	●	●	○	●
13	●	●	○	○
14	●	○	○	○

Figure G.1: Frameworks and recognisers analysis by Cirelli et al.

multi-touch gestures can be seen as a data-level fusion process. In our work, multi-touch support is analysed through the spatial, temporal and spatio-temporal specification criteria. Additionally, support for partial overlapping matches and segmentation criteria is also relevant to overcome the fingers permutation problem the authors mentioned.

5. **Support Multi-User** Support for multiple users corresponds to the concurrent interaction criteria we defined.
6. **Support Spatial Invariance** Scale, translation and rotation invariance is important for developing multi-touch applications. In our work, scale invariance depends on the spatial operators used and defined by the developer.
7. **Provide Continuous Feedback** This criteria corresponds to the online processing criteria we defined.
8. **Allow Easy Prototyping** Easy prototyping is important for optimally experimenting with novel sensors. In our work, this is analysed by the readability, debugging and authoring criteria.

9. **Support Symbolic Gestures** This criteria corresponds to the offline processing criteria we defined.
10. **Allow time-constrained Gestures** This criteria corresponds to the temporal specification criteria we defined.
11. **Support Territories** The support for spatial territories is important to distinguish gestures before delegation to the application level. In our work, territories are part of the spatial specification and application symbiosis criteria.
12. **Recognize Free-form Gestures** This criteria corresponds to the spatial specification criterion we defined.
13. **Support Sequential Gestures** A well-defined sequence of gestures can be performed to trigger a single more important action. In our work, this corresponds to the temporal and composition criteria.
14. **Support Cooperative Gestures** Cooperative gestures are supported when frameworks support the unification, grouping and concurrent interaction criteria we defined.

H

SMUIML XPaint Implementation

Listing H.1: A SMUIML implementation of a paint application

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <smuiml>
3   <integration_description client="xpaint_client">
4     <recognizers>
5       <recognizer name="speech" modality="speech"/>
6       <recognizer name="reactivision" modality="reactivision">
7         <variable name="posx" value="xpos"/>
8         <variable name="posy" value="ypos"/>
9         <variable name="fiducial" value="sourceId"/>
10      </recognizer>
11     <recognizer name="phidgetrfid" modality="rfid">
12       <variable name="shape" value="source"/>
13       <variable name="oper" value="source"/>
14       <translate_value from="2342111" to="filled circle"/>
15       <variable name="rfid_value" value="tagID"/>
16     </recognizer>
17     <recognizer name="phidget_ikit" modality="phidget_ikit"/>
18     <recognizer name="xpaint_client" modality="xpaint_client">
19       <variable name="background_selected" value="data" type="string"/>
20     </recognizer>
21   </recognizers>
22   <triggers>
23     <trigger name="select shape">
24       <source modality="speech" value="select shape"/>
25     </trigger>
26     <trigger name="return">
27       <source modality="speech" value="return"/>
```

```

28     </trigger>
29     <trigger name="begin draw">
30         <source modality="speech" value="begin draw"/>
31     </trigger>
32     <trigger name="end draw">
33         <source modality="speech" value="end draw"/>
34     </trigger>
35     <trigger name="select background">
36         <source modality="speech" value="select background"/>
37     </trigger>
38     <trigger name="operation">
39         <source modality="speech" value="erase shape | rotate shape | move shape"/>
40     </trigger>
41     <trigger name="selected shape">
42         <source modality="xpaint_client" value="$shape_selected"/>
43     </trigger>
44     <trigger name="position">
45         <source variable="fiducial" value="1 | 2" condition="valid"/>
46     </trigger>
47     <trigger name="position1">
48         <source variable="fiducial" value="1" condition="valid"/>
49     </trigger>
50     <trigger name="position2">
51         <source variable="fiducial" value="2" condition="valid"/>
52     </trigger>
53     <trigger name="tools_one_hand">
54         <source modality="rfid" value="select | line | freehand"/>
55     </trigger>
56     <trigger name="tools_two_hands">
57         <source modality="rfid" value="filled circle | empty circle | filled rectangle |
58             empty rectangle | filled polygon | empty polygon"/>
59     </trigger>
60     <trigger name="color">
61         <source modality="phidget_ikit" condition="valid"/>
62     </trigger>
63     <trigger name="thickness">
64         <source modality="phidget_ikit" condition="valid"/>
65     </trigger>
66     <trigger name="background_type">
67         <source variable="fiducial" value="1 | 2" condition="valid"/>
68     </trigger>
69     <trigger name="background_selected">
70         <source variable="data" value="$background_selected"/>
71     </trigger>
72 </triggers>
73 <actions>
74     <action name="draw_operation">
75         <target name="xpaint_client" message="draw $oper $shape $posx $posy"/>
76     </action>
77     <action name="selection_of_background">
78         <target name="xpaint_client" message="background selected"/>
79     </action>
80     <action name="modif_operation">
81         <target name="xpaint_client" message="modify $oper $shape $posx $posy"/>
82     </action>
83 </actions>

```

```

84 <dialog leadtime="2100">
85   <context name="start">
86     <transition leadtime="2200">
87       <trigger name="select shape"/>
88       <result context="modification"/>
89     </transition>
90     <transition>
91       <trigger name="begin draw"/>
92       <result context="drawing"/>
93     </transition>
94     <transition>
95       <trigger name="select background"/>
96       <result context="background"/>
97     </transition>
98   </context>
99   <context name="drawing">
100    <transition name="drawing_one_hand">
101      <par_and>
102        <!-- others : par_or, seq_and, seq_or -->
103        <trigger name="tools_one_hand"/>
104        <trigger name="color"/>
105        <trigger name="thickness"/>
106        <trigger name="position"/>
107      </par_and>
108      <result action="draw_operation"/>
109    </transition>
110    <transition name="drawing_two_hands">
111      <par_and>
112        <trigger name="tools_two_hands"/>
113        <trigger name="color"/>
114        <trigger name="thickness"/>
115        <trigger name="position1"/>
116        <trigger name="position2"/>
117      </par_and>
118      <result action="draw_operation"/>
119    </transition>
120    <transition>
121      <trigger name="end draw"/>
122      <result context="start"/>
123    </transition>
124  </context>
125  <context name="background">
126    <transition name="background_frame">
127      <trigger name="background_type"/>
128      <result action="selection_of_background"/>
129      <result context="start"/>
130    </transition>
131  </context>
132 </dialog>
133 </integration_description>
134 </smuiml>

```

Listing H.2: A Midas translation of the multimodal paint application

```

1 rule translateReactivation
2   r = Reactivation
3   assert Touch { source => "reactivation", xpos => r.posx,ypos => r.posy, fudicial => r.sourceId }
4 end
5 rule translatePhidgetRFID
6   r = PhidgetRFID
7   assert RFID { source => "phidgets", shape => r.shape,
8                 operation => r.operation, value => r.tagID }
9 end
10 rule translateRFID2342111
11   r = RFID { value == "2342111" }
12   modify r { value => "filled circle" }
13 end
14 # Add @Shadow annotation to XPaint class in Java
15 attempt operation
16   Speech { value == "erase shape" || value == "rotate shape" || value == "move shape" }
17 end
18 attempt position
19   Touch { fiducial == 1 || fiducial == 2 }
20 end
21 attempt toolsOneHand
22   RFID { value == "select" || value == "line" || value == "freehand" }
23 end
24 attempt toolsTwoHands
25   RFID { value == "filled circle" || value == "empty circle" ||
26         value == "filled rectangle" || value == "empty rectangle" ||
27         value == "filled polygon" || value == "empty polygon" }
28 end
29 attempt backgroundType
30   Touch { fiducial == 1 || value == 2 }
31 end
32 rule startModification
33   XPaint { context == "start" }
34   Speech { value == "select shape" }
35   call XPaint.beginModification
36 end
37 rule startDrawing
38   XPaint { context == "start" }
39   Speech { value == "begin draw" }
40   call XPaint.beginDraw
41 end
42 rule startBackground
43   XPaint { context == "start" }
44   Speech { value == "select background" }
45   call XPaint.beginBackground
46 end
47 rule drawingOneHand
48   XPaint { context == "drawing" }
49   t = toolsOneHand
50   c = color
51   t = thickness
52   p = position
53   Time←equal4F t, c, t, p, 2100.ms
54   call XPaint.draw t.operation, t.shape, p.fudicial
55 end

```

Context is managed by XPaint

```
56 rule drawingTwoHands
57   XPaint { context == "drawing" }
58   t = toolsTwoHands
59   c = color
60   t = thickness
61   p1 = position1
62   p2 = position2
63   Time←equal5F t, c, t, p1, p2, 2100.ms
64   call XPaint.draw t.operation, t.shape, p1.fudicial
65 end
66 rule drawingEnd
67   XPaint { context == "drawing" }
68   Speech { value == "end draw" }
69   call XPaint.endDraw
70 end
71 rule backgroundFrame
72   XPaint { context == "background" }
73   b = backgroundType
74   call XPaint.selectBackground b.fudicial           # Context is managed by XPaint
75 end
```

Bibliography

- [1] ALLEN, J. F. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM* 26, 11 (November 1983).
- [2] ALON, J., ATHITSOS, V., YUAN, Q., AND SCLAROFF, S. A Unified Framework for Gesture Recognition and Spatiotemporal Gesture Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31 (September 2009).
- [3] AMSTUTZ, R., AMFT, O., FRENCH, B., SMILAGIC, A., SIEWIOREK, D., AND TRÖSTER, G. Performance analysis of an HMM-based gesture recognition using a wristwatch device. In *Proceedings of the 17th International Conference on Computational Science and Engineering (CSE 2009)* (Vancouver, Canada, August 2009), pp. 303–309.
- [4] ANTHONY, L., AND WOBROCK, J. O. \$ N-Protractor: A Fast and Accurate Multistroke Recognizer. In *Proceedings of 28th International Conference on Graphics Interface (GI 2012)* (Toronto, Ontario, Canada, 2012).
- [5] AVOUAC, P.-A., LALANDA, P., AND NIGAY, L. Service-Oriented Autonomic Multimodal Interaction in a Pervasive Environment. In *Proceedings of the 13th International Conference on Multimodal Interfaces (ICMI 2011)* (Alicante, Spain, November 2011).
- [6] AVOUAC, P.-A., LALANDA, P., AND NIGAY, L. Autonomic management of multimodal interaction: DynaMo in action. In *Proceedings of the 4th SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2012)* (Copenhagen, Denmark, 2012).
- [7] BALI, M. *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing Ltd, 2009.

- [8] BALLMER, S. CES 2010: A Transforming Trend – the Natural User Interface. The Huffington Post, 2010. http://www.huffingtonpost.com/steve-ballmer/ces-2010-a-transforming-t_b_416598.html, Retrieved November 25, 2014.
- [9] BARCHUNOVA, A. *Manual Interaction: Multimodality, Decomposition, Recognition*. PhD thesis, Bielefeld University, Germany, February 2014.
- [10] BARRY, M., GUTKNECHT, J., KULKA, I., LUKOWICZ, P., AND STRICKER, T. Multimedial Enhancement of A Butoh Dance Performance - Mapping Motion to Emotion with A Wearable Computer System. In *Proceedings of the 2nd International Conference on Advances in Mobile Multimedia (MoMM 2004)* (Bali, Indonesia, 2004).
- [11] BOLT, R. A. “Put-That-There”: Voice and Gesture at the Graphics Interface. In *Proceedings of the 7th International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1980)* (Seattle, USA, July 1980).
- [12] BOUCHET, J., AND NIGAY, L. ICARE: A Component-Based Approach for the Design and Development of Multimodal Interfaces. In *Proceedings of the 22nd SIGCHI Conference on Human Factors in Computing Systems (CHI 2004)* (Vienna, Austria, 2004).
- [13] BOURGUET, M.-L. A Toolkit for Creating and Testing Multimodal Interface Designs. In *Proceedings of the 15th ACM Symposium on User Interface Software and Technology (UIST 2002)* (Paris, France, 2002).
- [14] BRACHA, G., AND COOK, W. Mixin-Based Inheritance. *ACM SIGPLAN Notices* 25, 10 (1990).
- [15] BRANTON, C., ULLMER, B., WIGGINS, A., ROGGE, L., SETTY, N., BECK, S. D., AND REESER, A. Toward Rapid and Iterative Development of Tangible, Collaborative, Distributed User Interfaces. In *Proceedings of the 5th SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2013)* (London, United Kingdom, 2013).
- [16] BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., MALER, E., AND YERGEAU, F. Extensible Markup Language (XML) 1.0 (Fifth

- Edition), 2008. <http://www.w3.org/TR/2008/REC-xml-20081126>, Retrieved November 25, 2014.
- [17] BROOKE, N. M., AND PETAJAN, E. D. Seeing Speech: Investigation Into the Synthesis and Recognition of Visible Speech Movement Using Automatic Image Processing and Computer Graphics. In *Proceedings of the International Conference on Speech Input and Output* (March 1986), Inspec/Iee.
- [18] BROOKS, JR., F. P. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer* 20, 4 (April 1987).
- [19] CARBONELL, N. Towards the design of usable multimodal interaction languages. *Universal Access in the Information Society* 2, 2 (2003), 143–159.
- [20] CERF, P. H., HURTON, M., AND COLOMIETS, P. 15/ZMTP - ZeroMQ Message Transport Protocol, 2009. <http://rfc.zeromq.org/spec:15>, Retrieved November 25, 2014.
- [21] CERF, V., DALAL, Y., AND SUNSHINE, C. SPECIFICATION of INTERNET TRANSMISSION CONTROL PROGRAM, 1974. <https://tools.ietf.org/html/rfc675>, Retrieved November 25, 2014.
- [22] CHAI, J., HONG, P., AND ZHOU, M. A Probabilistic Approach to Reference Resolution in Multimodal User Interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interfaces (IUI 2004)* (Funchal, Madeira, Portugal, 2004).
- [23] CHESHIRE, S., AND KROCHMAL, M. DNS-Based Service Discovery, 2013. <https://tools.ietf.org/html/rfc6763>, Retrieved November 25, 2014.
- [24] CHESHIRE, S., AND KROCHMAL, M. Multicast DNS, 2013. <https://tools.ietf.org/html/rfc6762>, Retrieved November 25, 2014.
- [25] CHUI, T. L.-K. Real-Time Computer Recognition of Handprinted Characters. Master's thesis, University of British Columbia, Canada, July 1976.
- [26] CIRELLI, M., AND NAKAMURA, R. A Survey on Multi-Touch Gesture Recognition and Multi-Touch Frameworks. In *Proceedings*

- of the 9th International Conference on Interactive Tabletops and Surfaces (ITS 2014)* (Dresden, Germany, 2014).
- [27] COHEN, P. R., JOHNSTON, M., MCGEE, D., OVIATT, S., PITTMAN, J., SMITH, I., CHEN, L., AND CLOW, J. Quickset: Multimodal Interaction for Distributed Applications. In *Proceedings of the 5th International Conference on Multimedia (MULTIMEDIA 1997)* (Seattle, USA, 1997).
- [28] COUTAZ, J., NIGAY, L., AND SALBER, D. Taxonomic Issues for Multimodal and Multimedia Interactive Systems. In *Proceedings of the 1st Workshop on Multimodal Human-Computer Interaction (ERCIM 1993)* (Nancy, France, November 1993).
- [29] COUTAZ, J., NIGAY, L., SALBER, D., BLANDFORD, A., MAY, J., AND YOUNG, R. M. Four Easy Pieces for Assessing the Usability of Multimodal Interaction: the CARE Properties. In *Proceedings of the Fifth International Conference on Human-Computer Interaction (INTERACT 1995)* (Lillehammer, Norway, June 1995), vol. 95.
- [30] CROCKFORD, D. The Application/JSON Media Type for JavaScript Object Notation (JSON), 2006. <https://www.ietf.org/rfc/rfc4627.txt>, Retrieved November 25, 2014.
- [31] CUENCA, F., CONINX, K., VANACKEN, D., AND LUYTEN, K. Graphical Toolkits for Rapid Prototyping of Multimodal Systems: A Survey. *Interacting with Computers* (2014).
- [32] DARRELL, T., AND PENTLAND, A. Space-Time Gestures. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 1993)* (New York, USA, June 1993).
- [33] DE BOECK, J., VANACKEN, D., RAYMAEKERS, C., AND CONINX, K. High-Level Modeling of Multimodal Interaction Techniques Using Nimmit. *Journal of Virtual Reality and Broadcasting* 4 (2007).
- [34] DEERING, S. E., AND HINDEN, R. M. Internet Protocol, Version 6 (IPv6) Specification, 1998. <http://tools.ietf.org/html/rfc2460>, Retrieved November 25, 2014.

- [35] DEMŠAR, J., ZUPAN, B., LEBAN, G., AND CURK, T. Orange: From Experimental Machine Learning to Interactive Data Mining. In *Knowledge Discovery in Databases (PKDD 2004)*, vol. 3202 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004.
- [36] DIETZ, P., AND LEIGH, D. DiamondTouch: A Multi-User Touch Technology. In *Proceedings of the 14th ACM Symposium on User Interface Software and Technology (UIST 2001)* (Orlando, Florida, 2001).
- [37] DOCX, L., DE LA ASUNCION, J., SABBE, B., HOSTE, L., BAETEN, R., WARNAERTS, N., AND MORRENS, M. Effort Discounting and Its Association with Negative Symptoms in Schizophrenia. *Cognitive Neuropsychiatry* (January 2015).
- [38] DUMAS, B. *Frameworks, Description Languages and Fusion Engines for Multimodal Interactive Systems*. PhD thesis, University of Fribourg, Switzerland, November 2010.
- [39] DUMAS, B., LALANNE, D., AND INGOLD, R. HephaisTK: A Toolkit for Rapid Prototyping of Multimodal Interfaces. In *Proceedings of the 11th International Conference on Multimodal Interfaces (ICMI-MLMI 2009)* (Cambridge, USA, November 2009).
- [40] DUMAS, B., LALANNE, D., AND INGOLD, R. Description Languages for Multimodal Interaction: A Set of Guidelines and its Illustration with SMUIML. *Journal on Multimodal User Interfaces* 3, 3 (July 2010).
- [41] DUMAS, B., LALANNE, D., AND OVIATT, S. Multimodal Interfaces: A Survey of Principles, Models and Frameworks. *Human Machine Interaction* (March 2009).
- [42] DUMAS, B., SIGNER, B., AND LALANNE, D. Fusion in Multimodal Interactive Systems: An HMM-Based Algorithm for User-Induced Adaptation. In *Proceedings of the 4th SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2012)* (Copenhagen, Denmark, 2012).
- [43] DUMAS, B., SIGNER, B., AND LALANNE, D. A Graphical Editor for the SMUIML Multimodal User Interaction Description Language. *Science of Computer Programming* 86 (2014).

- [44] ECHTLER, F., AND BUTZ, A. GISpL: Gestures Made Easy. In *Proceedings of the 6th International Conference on Tangible, Embedded and Embodied Interaction (TEI 2012)* (Kingston, Canada, February 2012).
- [45] ECHTLER, F., HUBER, M., AND KLINKER, G. Shadow Tracking on Multi-Touch Tables. In *Proceedings of the International Conference on Advanced Visual Interfaces (AVI 2008)* (Napoli, Italy, 2008).
- [46] ECHTLER, F., KAMMER, D., VANACKEN, D., HOSTE, L., AND SIGNER, B. Engineering Gestures for Multimodal User Interfaces. In *Proceedings of the 6th SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2014)* (Roma, Italy, June 2014).
- [47] ECHTLER, F., KLINKER, G., AND BUTZ, A. Towards a Unified Gesture Description Language. In *Proceedings of the 13th International Conference on Humans and Computers (HC 2010)* (Aizu-Wakamatsu, Japan, December 2010).
- [48] ELMEZAIN, M., AL-HAMADI, A., AND MICHAELIS, B. Hand Gesture Spotting based on 3D Dynamic Features Using Hidden Markov Models. *Communications in Computer and Information Science* 61 (2009).
- [49] EUGSTER, P. T., FELBER, P. A., GUERRAOUI, R., AND KERMARREC, A.-M. The Many Faces of Publish/Subscribe. *ACM Computing Surveys* 35, 2 (June 2003).
- [50] FANELLI, G., DANTONE, M., GALL, J., FOSSATI, A., AND VAN GOOL, L. Random Forests for Real Time 3D Face Analysis. *International Journal of Computer Vision* 101, 3 (February 2013).
- [51] FETTE, I., AND MELNIKOV, A. The WebSocket Protocol, 2011. <https://tools.ietf.org/html/rfc6455>, Retrieved November 25, 2014.
- [52] FIELDING, R. T., AND RESCHKE, J. F. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, 2014. <https://tools.ietf.org/html/rfc7230>, Retrieved November 25, 2014.
- [53] FLIPPO, F., KREBS, A., AND MARSIC, I. A Framework for Rapid Development of Multimodal Interfaces. In *Proceedings of the 5th*

- International Conference on Multimodal Interfaces (ICMI 2003)* (2003).
- [54] FORGY, C. L. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence* 19, 1 (1982).
- [55] FRIEDMAN-HILL, E. *Jess in Action: Java Rule-Based Systems*. Manning Publications, July 2003.
- [56] GIARRATANO, J. C. CLIPS User's Guide, Version 6.30, 2014. <http://clipsrules.sourceforge.net/documentation/v630/ug.pdf>, Retrieved November 25, 2014.
- [57] GIARRATANO, J. C., AND RILEY, G. *Expert Systems*. PWS Publishing Co., 1998.
- [58] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography* (Berkeley, USA, 1996), vol. 6.
- [59] HALL, M., FRANK, E., PFAHRINGER, G. H. B., REUTEMANN, P., AND WITTEN, I. H. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter* 11, 1 (June 2009).
- [60] HAMILTON, G. JavaBeans(TM) Specification 1.01, 1996. <http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec>, Retrieved November 25, 2014.
- [61] HAMON, A., PALANQUE, P., SILVA, J. L., DELERIS, Y., AND BARBONI, E. Formal Description of Multi-Touch Interactions. In *Proceedings of the 5th SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2013)* (London, United Kingdom, 2013).
- [62] HAMON, A., PALANQUE, P., SILVA, J. L., DELERIS, Y., AND BARBONI, E. Formal Description of Multi-Touch Interactions. In *Proceedings of the 5th SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2013)* (London, United Kingdom, 2013).

- [63] HOSTE, L. Software Engineering Abstractions for the Multi-Touch Revolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)* (Cape Town, South Africa, May 2010).
- [64] HOSTE, L., DUMAS, B., AND SIGNER, B. Mudra: A Unified Multimodal Interaction Framework. In *Proceedings of the 13th International Conference on Multimodal Interaction (ICMI 2011)* (Alicante, Spain, November 2011).
- [65] HOSTE, L., DUMAS, B., AND SIGNER, B. SpeeG: A Multimodal Speech- and Gesture-Based Text Input Solution. In *Proceedings of the International Conference on Advanced Visual Interfaces (AVI 2012)* (Capri Island, Italy, May 2012).
- [66] HOSTE, L., ROOMS, B. D., AND SIGNER, B. Declarative Gesture Spotting Using Inferred and Refined Control Points. In *Proceedings of the International Conference on Pattern Recognition (ICPRAM 2013)* (Barcelona, Spain, February 2013).
- [67] HOSTE, L., AND SIGNER, B. Expressive Control of Indirect Augmented Reality During Live Music Performances. In *Proceedings of the 3th International Conference on New Interfaces for Musical Expression (NIME 2013)* (Daejeon + Seoul, Korea Republic, May 2013).
- [68] HOSTE, L., AND SIGNER, B. SpeeG2: A Speech- and Gesture-Based Interface for Efficient Controller-free Text Input. In *Proceedings of the 15th International Conference on Multimodal Interaction (ICMI 2013)* (Sydney, Australia, December 2013).
- [69] HOSTE, L., AND SIGNER, B. Criteria, Challenges and Opportunities for Gesture Programming Languages. In *Proceedings of the 1st International Workshop on Engineering Gestures for Multimodal Interfaces (EGMI 2014)* (Rome, Italy, June 2014).
- [70] HOSTE, L., AND SIGNER, B. Water Ball Z: An Augmented Fighting Game Using Water as Tactile Feedback. In *Proceedings of the 8th International Conference on Tangible, Embedded and Embodied Interaction (TEI 2014)* (Munich, Germany, February 2014).

- [71] INFORMATION SCIENCES INSTITUTE OF THE UNIVERSITY OF SOUTHERN CALIFORNIA. INTERNET PROTOCOL, 1981. <http://tools.ietf.org/html/rfc791>, Retrieved November 25, 2014.
- [72] JAFARPOUR, H., MEHROTRA, S., AND VENKATASUBRAMANIAN, N. Dynamic Load Balancing for Cluster-Based Publish/Subscribe System. In *Proceedings of the 9th Annual International Symposium on Applications and the Internet (SAINT 2009)* (July 2009).
- [73] JOHNSON, R. E., AND FOOTE, B. Designing Reusable Classes. *Object-Oriented Programming 1, 2* (June 1988).
- [74] JOHNSTON, M. Unification-Based Multimodal Parsing. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics (ACL and COLING 1998)* (Montreal, Quebec, Canada, 1998).
- [75] JOHNSTON, M. Building Multimodal Applications with EMMA. In *Proceedings of the 11th International Conference on Multimodal Interfaces (ICMI-MLMI 2009)* (Cambridge, USA, November 2009).
- [76] JOHNSTON, M., AND BANGALORE, S. Finite-State Methods for Multimodal Parsing and Integration. In *Proceedings of the 13th European Summer School in Logic, Language and Information (ESSLLI 2001)* (Helsinki, Finland, August 2001).
- [77] JOHNSTON, M., BANGALORE, S., VASIREDDY, G., STENT, A., EHLEN, P., WALKER, M., WHITTAKER, S., AND MALOOR, P. MATCH: An Architecture for Multimodal Dialogue Systems. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL 2002)* (Philadelphia, USA, 2002).
- [78] JOHNSTON, M., COHEN, P., MCGEE, D., OVIATT, S., PITTMAN, J., AND SMITH, I. Unification-Based Multimodal Integration. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics (ACL 1997)* (Madrid, Spain, July 1997).
- [79] JORDÀ, S., GEIGER, G., ALONSO, M., AND KALTENBRUNNER, M. The ReacTable: Exploring the Synergy between Live Music Performance and Tabletop Tangible Interfaces. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction (TEI 2007)* (Baton Rouge, USA, February 2007).

- [80] JULIÀ, C. F., EARNSHAW, N., AND JORDÀ, S. GestureAgents: An Agent-Based Framework for Concurrent Multi-Task Multi-User Interaction. In *Proceedings of the 7th International Conference on Tangible, Embedded and Embodied Interaction (TEI 2013)* (Barcelona, Spain, 2013).
- [81] JULIÀ, C. F., AND JORDÀ, S. Towards Concurrent Multi-Tasking in Shareable Interfaces. *Journal of Computer Supported Collaborative Work* 25, Special Issue “Collaboration meets Interactive Surfaces - Walls, Tables, Tablets and Phones” (March 2015).
- [82] KADOUS, M. W. Learning Comprehensible Descriptions of Multivariate Time Series. In *Proceedings of 16th International Conference on Machine Learning (ICML 1999)* (Bled, Slovenia, June 1999).
- [83] KALTENBRUNNER, M., BOVERMANN, T., BENCINA, R., AND COSTANZA, E. TUIO: A Protocol for Table-Top Tangible User Interfaces. In *Proceedings of the 6th International Workshop on Gesture in Human-Computer Interaction and Simulation (GW 2005)* (Ile de Berder, France, May 2005).
- [84] KAMMER, D. *Formalisierung Gestischer Interaktion für Multitouch-Systeme*. PhD thesis, Technische Universität Dresden, September 2013.
- [85] KAMMER, D., HENKENS, D., AND GROH, R. GeForMTjs: A JavaScript Library based on a Domain Specific Language for Multi-Touch Gestures. In *Proceedings of Web Engineering, the 12th International Conference (ICWE 2012)* (Berlin, Germany, July 2012).
- [86] KAMMER, D., WOJDZIAK, J., KECK, M., GROH, R., AND TARANKO, S. Towards a Formalization of Multi-Touch Gestures. In *ACM International Conference on Interactive Tabletops and Surfaces* (2010).
- [87] KARAM, M., AND M. C. SCHRAEFEL. A Taxonomy of Gestures in Human Computer Interactions. Tech. rep., University of Southampton, Southampton, United Kingdom, August 2005.
- [88] KATSURADA, K., KIRIHATA, T., KUDO, M., TAKADA, J., AND NITTA, T. A Browser-based Multimodal Interaction System. In

- Proceedings of the 10th International Conference on Multimodal Interaction (ICMI 2008)* (Chania, Greece, 2008), ICMI '08, ACM, pp. 195–196.
- [89] KATSURADA, K., NAKAMURA, Y., YAMADA, H., AND NITTA, T. XISL: A Language for Describing Multimodal Interaction Scenarios. In *Proceedings of the 5th International Conference on Multimodal Interaction (ICMI 2003)* (Vancouver, Canada, 2003), ACM, pp. 281–284.
- [90] KHANDKAR, S. H., AND MAURER, F. A Domain Specific Language to Define Gestures for Multi-Touch Applications. In *Proceedings of the 10th Workshop on Domain-Specific Modeling (DSM 2010)* (Reno/Tahoe, USA, 2010).
- [91] KHANDKAR, S. H., AND MAURER, F. A Domain Specific Language to Define Gestures for Multi-Touch Applications. In *Proceedings of the 10th Workshop on Domain-Specific Modeling (DSM 2010)* (Reno-Tahoe, USA, October 2010).
- [92] KIM, J.-W., AND NAM, T.-J. EventHurdle: Supporting Designers' Exploratory Interaction Prototyping with Gesture-Based Sensors. In *Proceedings of the 31st SIGCHI Conference on Human Factors in Computing Systems (CHI 2013)* (Paris, France, 2013).
- [93] KIM, J.-W., NAM, T.-J., AND PARK, T. CompositeGesture: Creating Custom Gesture Interfaces with Multiple Mobile or Wearable Devices. *International Journal on Interactive Design and Manufacturing (IJIDeM)* (2014).
- [94] KIN, K. Proton++: A customizable declarative multitouch framework. In *Proceedings of the 25th Annual Symposium on User Interface Software and Technology (UIST 2012)* (Cambridge, USA, 2012).
- [95] KIN, K., HARTMANN, B., DEROSE, T., AND AGRAWALA, M. Proton: Multitouch Gestures as Regular Expressions. In *Proceedings of the 30th SIGCHI Conference on Human Factors in Computing Systems (CHI 2012)* (Austin, USA, November 2012).
- [96] KINECT FOR WINDOWS TEAM. Visual Gesture Builder: A Data-Driven Solution to Gesture Detection. Tech.

- rep., Microsoft, Redmond, Washington, Verenigde Staten, July 2014. <https://onedrive.live.com/view.aspx?resid=1A0C78068E0550B5!77743&app=WordPdf>, Retrieved November 25, 2014.
- [97] KÖNIG, W., RÄDLE, R., AND REITERER, H. Squidy: A Zoomable Design Environment for Natural User Interfaces. In *Proceedings of CHI 2009, ACM Conference on Human Factors in Computing Systems* (Boston, USA, 2009).
- [98] KUIJPERS, E., AND WILSON, M. A Multi-Modal Interface for Man Machine Interaction with Knowledge based Systems - MMI². *Knowledge based Systems 2001* (1992).
- [99] LAETHEM, B. V. Recognition of Deictic and Iconic Gestures for Home Automation. Master's thesis, Vrije Universiteit Brussel, June 2012.
- [100] LALANNE, D., NIGAY, L., ROBINSON, P., VANDERDONCKT, J., AND LADRY, J.-F. Fusion Engines for Multimodal Input: A Survey. In *Proceedings of the 11th International Conference on Multimodal Interfaces (ICMI-MLMI 2009)* (Cambridge, USA, November 2009).
- [101] LARSEN, L. Eric Horvitz on the New Era of Artificial Intelligence, January 2015. <https://channel9.msdn.com/Series/Microsoft-Research-Luminaries/Eric-Horvitz-on-the-new-era-of-Artificial-Intelligence>, Retrieved January 29, 2015.
- [102] LATOSCHIK, M. E. Designing Transition Networks for Multimodal VR-Interactions Using a Markup Language. In *Proceedings of the 4th International Conference on Multimodal Interfaces (ICMI 2002)* (Pittsburgh, USA, 2002).
- [103] LAWSON, J.-Y. L., AL-AKKAD, A.-A., VANDERDONCKT, J., AND MACQ, B. 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 2009)* (Pittsburgh, USA, July 2009).
- [104] LEE, H.-K., AND KIM, J.-H. An HMM-Based Threshold Model Approach for Gesture Recognition. *Transactions on Pattern Analysis and Machine Intelligence* 21, 10 (October 1999).

- [105] LI, M., LIU, M., DING, L., RUNDENSTEINER, E. A., AND MANI, M. Event Stream Processing with Out-Of-Order Data Arrival. In *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops (ICDCSW 2007)* (Washington, USA, June 2007).
- [106] LI, Y. Protractor: A Fast and Accurate Gesture Recognizer. In *Proceedings of the 28th SIGCHI Conference on Human Factors in Computing Systems (CHI 2010)* (Atlanta, USA, 2010).
- [107] LONG JR, A. C., LANDAY, J. A., AND ROWE, L. A. Implications for A Gesture Design Tool. In *Proceedings of the 17th SIGCHI Conference on Human Factors in Computing Systems (CHI 1999)* (Pittsburgh, Pennsylvania, USA, 1999).
- [108] LÜ, H., FOGARTY, J. A., AND LI, Y. Gesture Script: Recognizing Gestures and Their Structure Using Rendering Scripts and Interactively Trained Parts. In *Proceedings of the 32nd SIGCHI Conference on Human Factors in Computing Systems (CHI 2014)* (Toronto, Ontario, Canada, 2014).
- [109] LÜ, H., AND LI, Y. Gesture Coder: A Tool for Programming Multi-Touch Gestures by Demonstration. In *Proceedings of the 30th SIGCHI Conference on Human Factors in Computing Systems (CHI 2012)* (Austin, USA, 2012).
- [110] MACKENZIE, I. S., AND WARE, C. Lag As a Determinant of Human Performance in Interactive Systems. In *Proceedings of the 11th SIGCHI Conference on Human Factors in Computing Systems (CHI 1993)* (Amsterdam, The Netherlands, 1993).
- [111] MAIER, I., ROMPF, T., AND ODERSKY, M. Deprecating the Observer Pattern. Tech. Rep. EPFL-REPORT-148043, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, April 2010.
- [112] MARQUARDT, N. Proxemic Interactions with and Around Digital Surfaces. In *Proceedings of the 8th International Conference on Interactive Tabletops and Surfaces (ITS 2013)* (St. Andrews, United Kingdom, 2013).
- [113] MARR, S., RENAUX, T., HOSTE, L., AND DE MEUTER, W. Parallel Gesture Recognition with Soft Real-Time Guarantees. *Science of Computer Programming* (February 2014).

- [114] MCCARTHY, J. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM* 3, 4 (1960).
- [115] MEIJER, E., BECKMAN, B., AND BIERMAN, G. Linq: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the International Conference on Management of Data (SIGMOD 2006)* (2006), ACM.
- [116] MENDONÇA, H., LAWSON, J.-Y. L., VYBORNOVA, O., MACQ, B., AND VANDERDONCKT, J. A Fusion Framework for Multimodal Interactive Applications. In *Proceedings of the 11th International Conference on Multimodal Interfaces (ICMI-MLMI 2009)* (Cambridge, USA, November 2009).
- [117] MIRANKER, D. P. Treat: A Better Match Algorithm for AI Production Systems. In *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI 1987)* (Seattle, USA, 1987), pp. 42–47.
- [118] MIRANKER, D. P., BRANT, D. A., LOFASO, B., AND GADBOIS, D. On the Performance of Lazy Matching in Production Systems. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI 1990)* (Boston, USA, 1990), pp. 685–692.
- [119] NACENTA, M. A., BAUDISCH, P., BENKO, H., AND WILSON, A. Separability of Spatial Manipulations in Multi-Touch Interfaces. In *Proceedings of Graphics interface 2009* (British Columbia, Canada, 2009).
- [120] NAVARRE, D., PALANQUE, P., LADRY, J.-F., AND BARBONI, E. ICOs: A Model-Based User Interface Description Technique Dedicated to Interactive Systems Addressing Usability, Reliability and Scalability. *ACM Transactions on Computer-Human Interaction (TOCHI)* 16, 4 (November 2009).
- [121] NIEWIADOMSKI, R., MANCINI, M., DING, Y., PELACHAUD, C., AND VOLPE, G. Rhythmic Body Movements of Laughter. In *Proceedings of the 16th International Conference on Multimodal Interaction (ICMI 2014)* (Istanbul, Turkey, 2014).

- [122] NIGAY, L. Design space for multimodal interaction. In *Building the Information Society*, IFIP International Federation for Information Processing. Springer, 2004, pp. 403–408.
- [123] NIGAY, L., AND COUTAZ, J. A Design Space for Multimodal Systems: Concurrent Processing and Data Fusion. In *Proceedings of the 11th Conference on Human Factors in Computing Systems (INTERACT 1993 and CHI 1993)* (Amsterdam, the Netherlands, 1993).
- [124] OVIATT, S. Multimodal Interfaces. *The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications* (2003).
- [125] OWENS, M., AND ALLEN, G. *The Definitive Guide to SQLite*, vol. 1. Springer, 2006.
- [126] PARR, T. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.
- [127] PEFFERS, K., TUUNANEN, T., ROTHENBERGER, M. A., AND CHATTERJEE, S. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems* 24, 3 (2007), 45–77.
- [128] PETAJAN, E., BISCHOFF, B., BODOFF, D., AND BROOKE, N. M. An Improved Automatic Lipreading System to Enhance Speech Recognition. In *Proceedings of the 6th SIGCHI Conference on Human Factors in Computing Systems (CHI 1988)* (Washington, USA, June 1988).
- [129] PFLEGER, N., AND SCHEHL, J. Development of Advanced Dialog Systems with PATE. In *Proceedings of the 9th International Conference on Spoken Language Processing (INTERSPEECH ICSLP 2006)* (Pittsburgh, USA, 2006).
- [130] QUEK, F., MCNEILL, D., BRYLL, R., DUNCAN, S., MA, X.-F., KIRBAS, C., MCCULLOUGH, K. E., AND ANSARI, R. Multimodal Human Discourse: Gesture and Speech. *ACM Transaction on Computer-Human Interaction* 9, 3 (September 2002), 171–193.
- [131] RENAUX, T., HOSTE, L., MARR, S., AND DE MEUTER, W. Parallel Gesture Recognition with Soft Real-Time Guarantees. In

- Proceedings of the 2nd International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE! 2012)* (Tucson, USA, October 2012).
- [132] RENAUX, T., HOSTE, L., MARR, S., AND DE MEUTER, W. Software Engineering Principles in the Midas Gesture Specification Language. In *Proceedings of the 2nd International Workshop on Programming for Mobile and Touch (PRoMoTo 2014)* (Portland, USA, October 2014).
- [133] RHYNE, J. Dialogue Management for Gestural Interfaces. *SIGGRAPH Computer Graphics* 21, 2 (April 1987).
- [134] RIVEST, R. S-Expressions, draft-rivest-sexp-00.txt. Network Working Group, Internet Draft, 1997. <http://people.csail.mit.edu/rivest/Sexp.txt>, Retrieved November 25, 2014.
- [135] ROOMS, B. D. VolTra: A Development Environment for Extracting and Defining 3D Full-Body Gestures. Master's thesis, Vrije Universiteit Brussel, August 2012.
- [136] RUBINE, D. Specifying Gestures by Example. In *Proceedings of the 18th International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1991)* (Las Vegas, USA, August 1991).
- [137] RUBINE, D. Specifying Gestures by Example. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1991)* (Las Vegas, USA, August 1991).
- [138] SALOJÄRVI, J., PUOLAMÄKI, K., SIMOLA, J., KOVANEN, L., KOJO, I., AND KASKI, S. Inferring relevance from eye movements: Feature extraction. Tech. rep., Aalto University, Espoo, Finland, March 2005.
- [139] SALVANESCHI, G., EUGSTER, P., AND MEZINI, M. Programming with Implicit Flows. *Software* 31, 5 (September 2014).
- [140] SARNA-STAROSTA, B., AND SCHRIJVERS, T. Transformation-based indexing techniques for Constraint Handling Rules. In *Proceedings of the 5th Workshop on Constraint Handling Rules* (July 2008).

- [141] SCHOLLIERS, C., HOSTE, L., SIGNER, B., AND DE MEUTER, W. Midas: A Declarative Multi-Touch Interaction Framework. In *Proceedings of the 5th International Conference on Tangible, Embedded, and Embodied Interaction (TEI 2011)* (Funchal, Portugal, January 2011).
- [142] SERRANO, M., NIGAY, L., LAWSON, J., RAMSAY, A., MURRAY-SMITH, R., AND DENEFF, S. The OpenInterface Framework: A Tool for Multimodal Interaction. In *Proceedings of the 26th SIGCHI Conference on Human Factors in Computing Systems (CHI 2008)* (Florence, Italy, April 2008).
- [143] SHAFRANOVICH, Y. Common Format and MIME Type for Comma-Separated Values (CSV) Files, 2005. <https://tools.ietf.org/html/rfc4180>, Retrieved November 25, 2014.
- [144] SHARMA, R., PAVLOVIC, V., AND HUANG, T. Toward Multimodal Human-Computer Interface. *Proceedings of the IEEE* 86, 5 (August 1998).
- [145] SIGNER, B., KURMANN, U., AND NORRIE, M. C. iGesture: A General Gesture Recognition Framework. In *Proceedings of the 9th International Conference on Document Analysis and Recognition (ICDAR 2007)* (Curitiba, Brazil, September 2007).
- [146] SOTTET, J.-S., CALVARY, G., COUTAZ, J., FAVRE, J.-M., VANDERDONCKT, J., STANCIULESCU, A., AND LEPREUX, S. A Language Perspective on the Development of Plastic Multimodal User Interfaces. *Journal on Multimodal User Interfaces* 1, 2 (2007).
- [147] SPANO, L. D. Defining CARE Properties Through Temporal Input Models. In *Proceedings of the 1st International Workshop on Engineering Gestures for Multimodal Interfaces (EGMI 2014)* (Rome, Italy, June 2014).
- [148] SPANO, L. D., CISTERMINO, A., PATERNÒ, F., AND FENU, G. GestIT: A declarative and compositional framework for multiplatform gesture definition. In *Proceedings of the 5th SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2013)* (London, United Kingdom, 2013).
- [149] STANCIULESCU, A., LIMBOURG, Q., VANDERDONCKT, J., MICHOTTE, B., AND MONTERO, F. A Transformational Approach for

- Multimodal Web User Interfaces based on UsiXML. In *Proceedings of the 7th International Conference on Multimodal Interfaces (ICMI 2005)* (Toronto, Italy, 2005).
- [150] STANCIULESCU, A., AND VANDERDONCKT, J. Design Options for Multimodal Web Applications. In *Computer-Aided Design Of User Interfaces V*. Springer, 2007, pp. 41–56.
- [151] SWALENS, J., RENAUX, T., HOSTE, L., MARR, S., AND DE MEUTER, W. Cloud PARTE: Elastic Complex Event Processing based on Mobile Actors. In *Proceedings of the 3rd International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE! 2013)* (Indianapolis, USA, October 2013).
- [152] SWIGART, S. Easily Write Custom Gesture Recognizers for Your Tablet PC Applications, 2005. <https://msdn.microsoft.com/en-us/library/aa480673.aspx>, Retrieved November 25, 2014.
- [153] THOMAS, D., HANSSON, D., BREEDT, L., CLARK, M., DAVIDSON, J. D., GEHTLAND, J., AND SCHWARZ, A. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2006.
- [154] VAN CUTSEM, T. Why Programming Languages?, Januari 2011. <http://soft.vub.ac.be/~tvcutsem/invokedynamic/node/11>, Retrieved November 25, 2014.
- [155] VAN SEGHBROECK, G., VERSTICHEL, S., DE TURCK, F., AND DHOEDT, B. WS-Gesture, A Gesture-Based State-aware Control Framework. In *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA 2010)* (December 2010).
- [156] VAN WEERT, P. Efficient Lazy Evaluation of Rule-Based Programs. *IEEE Transactions on Knowledge and Data Engineering* 22, 11 (2010), 1521–1534.
- [157] VATAVU, R.-D. The Effect of Sampling Rate on the Performance of Template-Based Gesture Recognizers. In *Proceedings of the 13th International Conference on Multimodal Interfaces (ICMI 2011)* (2011).
- [158] VO, M. T., AND WOOD, C. Building an Application Framework for Speech and Pen Input Integration in Multimodal Learning

- Interfaces. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 1996)* (Atlanta, USA, May 1996).
- [159] WILLIS, N. Multi-Touch Support Landing in X, 2012. <https://lwn.net/Articles/475886/>, Retrieved November 25, 2014.
- [160] WILSON, A., AND BOBICK, A. Parametric Hidden Markov Models for Gesture Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 21 (September 1999).
- [161] WINER, D. XML-RPC Specification, 1999. <http://xmlrpc.scripting.com/spec>, Retrieved November 25, 2014.
- [162] WOBROCK, J. O., WILSON, A. D., AND LI, Y. Gestures Without Libraries, Toolkits or Training: A \$1 Recognizer for User Interface Prototypes. In *Proceedings of the 20th ACM Symposium on User Interface Software and Technology (UIST 2007)* (Newport, USA, October 2007).
- [163] WRIGHT, M., AND FREED, A. Open Sound Control: A New Protocol for Communicating with Sound Synthesizers. In *Proceedings of the 23rd International Computer Music Conference (ICMC 1997)* (Thessaloniki, Greece, September 1997).
- [164] WU, L., OVIATT, S., AND COHEN, P. From Members to Teams to Committee - A Robust Approach to Gestural and Multimodal Recognition. *IEEE Transactions on Neural Networks* 13, 4 (2002).
- [165] YERGEAU, F. UTF-8, A Transformation Format of ISO 10646, 2003. <http://tools.ietf.org/html/rfc3629>, Retrieved November 25, 2014.