Software Engineering Principles in the Midas Gesture Specification Language

Thierry Renaux Lode Hoste Christophe Scholliers Wolfgang De Meuter
Vrije Universiteit Brussel
{trenaux, lhoste, cfscholl, wdemeuter}@vub.ac.be

Abstract

While many technologies for gesture-based interaction have been proposed and implemented, few focus on core software engineering principles that are commonplace in traditional programming languages. The lack of such principles restricts the applicability of those technologies when developing large scale gesture enabled systems. This paper describes the software engineering challenges associated with developing multitouch gesture-based interaction, and proposes a solution in the form of the Midas declarative gesture specification language. Midas embeds concepts of logical programming languages and complex event processing to ease the development of gesture based applications. We show how it can be applied to multitouch gesture recognition, and evaluated our solution in real-world applications.

Categories and Subject Descriptors D.2 [*SOFTWARE ENGINEERING*]; H.5.2 [*INFORMATION INTERFACES AND PRESENTATION*]: User Interfaces—Input devices and strategies

Keywords Gesture Recognition, Multitouch, Declarative specification

1. Introduction

Gesture-enabled devices make up the bulk of the computers sold nowadays. Whether through multitouch trackpads, multitouch screens, or depth-sensing cameras, the human-device-interaction is shifting towards a gesture-centric approach. Many gestures are the direct descendant of previous methods of interaction: a 'tap' differs little from a mouse click, and swipe gestures resemble the use of scroll-wheels which have been in use for quite some time. But with increasing sensor capabilities, and with increasing processing power allocated for handling user-interactions, increasingly complex gestures need to be programmed.

As we shall argue further in this paper, the state of the art in software engineering is not keeping up with the pace of improvements in the hardware. Programmers designing user interactions are largely left to their own devices, given the choice between either picking highly restrictive off-the-shelf gesture frameworks, or manually maintaining gesture state. This hampers the ability of software developers to design new ways of interaction, for instance in games with custom, game-setting-specific interaction and UI behavior, or for pioneering developers wishing to create content for novel device types for which no standard, well-defined UI behavior exist.

A gesture is defined as "a movement of the hands, face or other parts of the body in time" [1]. In a computing device, a gesture is reified as a series of *events* specifying those movements. To program gesture-based interaction hence comes down to programming software systems which extract meaningful data from series of events. This task is complicated by a number of issue. For starters, using traditional programming languages the boilerplate code to set up event handlers is tedious to write and error-prone. The same holds true for the code to maintain the "context", i. e., the set of variables representing the state of a gesture as it is being recognized.

The complexity is even exacerbated in the absence of constructs for modularization, abstraction, and composition of gesture specifications. Lacking such constructs, gesture programmers require deep knowledge of the existing gesture definitions when adding new gestures or modifying existing ones, as overlaps in the definitions need to be handled.

Furthermore, to support for instance both tap and double tap gestures, temporal reasoning is required. This reasoning is complicated by the inherent concurrency of multitouch gestures.

Even without temporal aspects, gesture definitions may overlap. This is the case for e. g. two-finger "pinch" and "rotate" gestures.

Finally, identifying which events relate to which finger is difficult, yet necessary to distinguish even simple gestures, or ignore unrelated events like those from hand palms resting on the touch sensitive surface.

This paper explains the software engineering challenges associated with developing multitouch gestures, and proposes a solution in the form of a declarative framework providing software engineering abstractions. This framework and its accompanying declarative domain specific language, Midas [2], has been extended with improved support for abstraction and modularization. The software engineering challenges and the merits of the Midas gesture specification language are presented by first introducing an example of a multi stroke touch gesture (subsection 2.1). Using this example gesture, the general software engineering challenges of programming gesture-based interaction are demonstrated (subsection 2.2). Next, the Midas declarative gesture specification language is proposed as a solution to the problems (section 3). The example gesture introduced earlier is revisited, demonstrating how Midas tackles the software engineering challenges (subsection 3.2). Finally, we discuss how Midas relates to the state of the art (section 5).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SPLASH '14 Companion, Oct 20–24, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3208-8/14/10.
http://dx.doi.org/10.1145/2660252.2662136

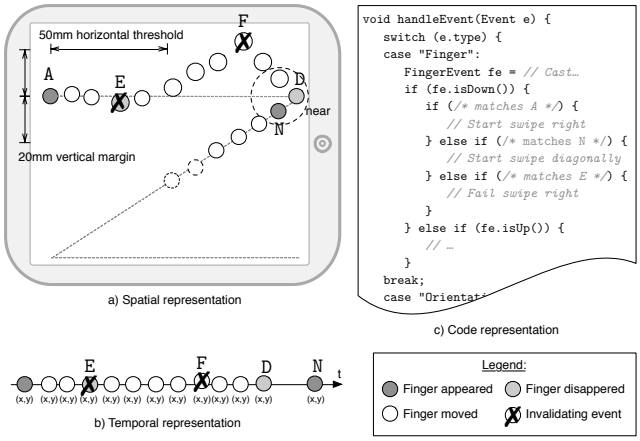

Figure 1: Events of a "swipe z" gesture spatially overlaid over a Z-shape (a) and situated on a temporal axis (b), together with a code snippet depicting the outline of recognition code in a traditional imperative style (c)

2. Motivating Example

2.1 A Three-Stroke Gesture

Consider the task of a programmer to implement a gesture for an application on a touch-enabled mobile device. The requirements document specifies the gesture as follows:

> *"A finger swipes right, swipes down diagonally to the left, and finally swipes right again. Subsequent strokes start near the end of the preceding stroke."*

The gesture forms a "Z" shape, and will hence be called "swipe z". "swipe z" is conceptually defined in terms of two more gestures, "swipe right" and "swipe diagonally". The "swipe right" gesture is specified as follows:

> *"A finger touches the device, and over time progresses at least 50mm to the right, and deviates from the horizontal line at most 20mm vertically. The swipe ends when the finger is removed from the surface."*

Information on primitive events, such as a finger touching, moving on, or leaving the touch-sensitive surface, is assumed to be made available by a low level API.

Figure 1 illustrates the gesture graphically from a spatial (a) and temporal (b) perspective, and as a code snippet (c) in imperative pseudocode. The gray dashed lines in (a) depict the Z-shape to be recognized. The circles represent registered touch events. After the appearance of a finger (marked in the image as `A`), more finger movements may be registered to the right. If a finger is lifted early (`E`), i. e., before it moved 50mm to the right, the gesture should not be recognized. If the finger movements sway too far (`F`) vertically, i. e., more than 20mm up or down, the gesture is invalid as well. If the finger disappears (`D`) and no invalidating events occurred, the "swipe right" gesture is completed.

If the next event (`N`) is the start of a series of events satisfying the "swipe diagonally" gesture, and `N` is near the first stroke's endpoint (`D`), the recognition of the "swipe z" gesture continues. While these requirements may seem reasonably straightforward on the surface, translating this to actual software reveals a number of issues. The following section discusses the challenges software engineers face when developing even simple gesture-based interactions.

2.2 Software Engineering Challenges

Even with simple gestures, handling is not straightforward. Mostly, all information about events' types, locations, timestamps, etc. are squeezed through the funnel of a single event-handler function, which has to unpack and sort through all data. The view the program gets resembles much more to Figure 1 (b) than Figure 1 (a). We categorize the challenges software engineers face when designing gesture-based interaction as overhead, modularization, temporal constraints, concurrency and conflicts, and event identification. Each of these categories is described in more detail below.

Overhead and context maintenance Reacting to (series of) events implies setting up detection code and handlers. The detection code and handlers have to cooperate with each other and with the code for other events. The inversion of control [3] of the "callback hell" associated with event handlers have well-documented nefarious effects on programmer effectivity [4]. For example, to share data between the handlers cooperating to recognize an event, this data has to be stored in the scope of all handlers. In traditional programming languages, this tends to imply the data is stored in class-level or even global variables.

The code snippet in Figure 1 (c) skims over the complexity of this task: the set of events constituting the gestures needs to be stored temporarily as the matching is taking place, so that new events arriving later can take into account what happened before. This data needs to be visible to all gesture handling code, and also needs to be discarded when the gesture is completed or cannot be completed anymore.

Straightforward solutions to clean up this gesture state fail to work even for our simple Z-shape. At two points during the gesture, no fingers are required to be touching the surface, yet the initial two strokes need to be remembered for the entire gesture to be correctly recognized. Resetting the system whenever all fingers are released from the surface, as done in the gesture frameworks libTISCH [5] and SparshUI [6], is hence not sufficient.

Modularization, composition, and abstraction Since gestures are recognized from the combination of multitudes of event states, code that jointly defines one single gesture is easily scattered over the codebase. This becomes clear when implementing even simple gestures. The deep nesting in Figure 1 (c) is archetypical of gesture programming using traditional software languages, yet leads to an unmaintainable codebase. Deep knowledge of existing gestures is required to add new gestures. The clear cut hierarchy and modularity from the specification is lost in translation; with code for the "swipe right" and "swipe diagonally" code intermingled, and code for "swipe z" unintelligibly dissolved into the lower levels.

Temporal constraints A gesture exists out of a number of events. Recognizing the event hence means correlating events in time. Often this is rather straightforward, but more complicated situations arise even with very common gestures. The difference between a tap and a double tap lies in the absence of a second tap after some waiting period. This implies the gesture recognition system needs to maintain timers allowing the tap gesture to trigger if no double tap gesture triggers in a certain timespan.

Furthermore, events which might be relevant for multiple gestures are scattered within a continuous input stream. Determining start- and endpoints of gestures, commonly called "segmentation", needs to be tackled. In the preceding example, this issue does not arise, as strokes have explicit start and endpoints in form of "appear" and "disappear" states of the fingers' motion events.

Concurrency and conflicts The defining property of multitouch gestures is that multiple touches may overlap in time. Multitouch gesture recognition is hence inherently concurrent. The situation is further complicated since multiple gestures may overlap in definition, causing conflicts. A common example is found in "pinch" and "rotate" gestures as executed on a multi-touch surface using two fingers. Both gestures start with two fingers in roughly the same organisation, and differ in whether the inter-finger rotation or distance is modified. In practice, however, most executions of the "rotate" gesture will also slightly scale the inter-finger distance, and most executions of the "pinch" gesture will include a rotation of the fingers' coordinates. If proper care is not taken, both gestures would hence erroneously be recognized at the same time.

The events forming a "swipe z" gesture are explicitly sequential. Still, the gesture description does not explicitly prohibit other gestures to be going on at the same time. Other touch events which do not explicitly invalidate the constraints of the swipe gestures should not interfere with the recognition of the swipes. However, that does not mean those other touches should be ignored. Any gesture performed elsewhere, which does not explicitly prohibit the swipe gestures, should be recognized normally [1].

Identification Events are by design isolated data points. The gesture programmer needs to figure out whether two events originate from the same finger or from two different fingers. For instance, the description of the "swipe diagonally" gesture indicates the gesture should be performed with one finger. This implies the system should track finger ids. Yet, the gesture may be executed with any finger. The "swipe diagonally" gesture should only match when the swipe is performed with a finger that just finished a "swipe right". However, a second "swipe right" gesture might be executed with the same finger – completing the current Z-shape – or be executed with a different finger – possibly starting a different, valid Z-shape. To make matters worse, the second "swipe right" stroke could both be the last stroke of the current Z-shape and the first stroke of another valid "swipe z" gesture. The gesture specification language should be able to describe whether in such situations both gestures should be recognized, only the first, only the second, or none at all.

Finally, elements of the graphical user interface often need to be identified as well. To allow application logic to work with events and gestures, events and gestures require an identity, such that they may be referenced. This is for instance required to allow incremental updates in response to a gesture that is still being executed, as the application logic receives multiple data about the same gesture. The other way around, gestures may depend on the state and bounds of GUI widgets. This entails that there need to be ways to identify components in application logic from the gesture recognition logic.

2.3 Discussion

Scholliers et al. [2] and Richardson et al. [7] argue that most frameworks and libraries have significant shortcomings in their software engineering characteristics. Furthermore, Hammond and Davis [8] demonstrated that domain specific languages lend themselves better to the problem of specifying gestures. However, a paper by Hoste and Signer [1] defines 30 criteria for gesture programming languages, and finds all existing languages come short on a multitude of those criteria. We argue that the most promising approach for specifying touch and, by extension, gesture input is one based on *declarative specification of patterns of events*. A declarative specification of gestures allows the programmer to describe the gesture instead of implement the recognition process. An online, reactive inference engine can then take care of state maintenance, segmentation, and overlap in gesture definition. An approach based on a declarative gesture specification language has the additional advantage of facilitating a separation of concern. The concern of developing the application, of designing the gestures, and of implementing the gesture recognition system can be tackled independently.

Critically to such an approach, however, is that the gesture recognition system does not simply offer a fixed catalogue of hard coded gestures. It should instead offer the gesture designer a rich, extensible set of primitives, e. g. to deal with spatio-temporal reasoning and event identification, and offer means to compose those into new abstractions.

3. Midas and the Mudra Framework

This paper discusses the software engineering principles applied in Midas [2], a declarative gesture specification language. Its runtime is embedded into the Mudra [9] gesture-based interaction framework. Events arrive in the Mudra system, and get matched to patterns by an inference engine as soon as the events arrive. Whenever an entire gesture's patterns are matched, a reaction is automatically executed. The pattern matching phase is optionally aided by a set of temporal and spatial operators and external recognizers such as template matchers or machine learning based recognizers. The useable operators are not limited to the built-in catalog: new operators can be defined by the developer.

3.1 Main Language Constructs

This section introduces the main language constructs in Midas, as well as their syntax. A full specification of the semantic entities of the Midas language can be found in Figure 2.

At the top level, a Midas program consists of "templates", "modules" and "rules". Templates represent a kind of event: a template forms a 'blueprint' which events are instances of. As such, templates have a name, for identification purposes, and specify which "slots" events adhering to that template have. For instance, events adhering to a "FingerTouched" template might have an `x` and a `y` slot.

Where templates can be regarded as equivalent to a class in object oriented programming, modules are similar to mixins. Programmers may for instance declare a module `CarthesianPoint` with slots `x` and `y`, and declare some functions that will work for all events whose template `includes` the `CarthesianPoint` module.

The third and final kind of top-level element, declarative rules, are used in Midas to describe the actual gestures. Gestures are described by listing the events that must be present, and the events that may not be present. When all events required by the rule are found, an automatic reaction takes place. This can take the form of either performing some application logic, or creating new derived events and adding them to the engine's knowledge base. The latter allows for *composing events into higher-level events*.

Within a rule, the description of events is based on two concepts: condition elements and tests. Condition elements specify what kind of event they should be matched to (by naming the event template), and optionally constrain the slots of the event. For instance, a condition element may require that the event's `fingerCount` slot holds the value three:

```
SomeEventTemplate {fingerCount == 3}
```

[1] The same holds for other input modalities, such as speech or device-orientation, which are not relevant to the specific gestures described in subsection 2.1, but in general might be jointly used in a single gesture specification.

Condition elements may be bound to logical variables, allowing to refer to the event captured by the condition element. Capturing of events from condition elements to variables is designated by a single equality sign:

```
var = SomeEventTemplate {fingerCount == 3}
```

In addition to the simple equality constraints on the slots of events captured by condition elements, more involved constraints can be enforced by tests. Tests can enforce (in-)equality of multiple slots, or call domain-specific predicates, e. g., to perform spatio-temporal reasoning. For instance, the following line enforces that event `e1` is located more to the left than event `e2` by calling the `spatiallyLeftOf` domain-specific predicate:

```
test spatiallyLeftOf(e1, e2)
```

Predicates can be stored in the scope of a module. For instance, a module might be created that specifies a `rotation` slot, and bundles it with predicates that act on rotation. Or a higher-level module might deal with the application-level concept of tracking interaction between multiple users on a wide touch enabled surface. Such a module could define slots like `activityCentroid` and `maxDistanceFromCenter`, together with predicates reasoning about the spatial behavior of the group based on those slots.

Modules hence offer a second, complementary form of composition to event-composition. Where event composition has the advantage of creating new 'normal' events, which are treated like any other event adhering to the same event template, they have the drawback of introducing data entanglement: all information captured by the lower-level gesture which might be relevant to any higher-level gesture, must be explicitly packaged into the event template. If for instance any potential consumer of a "swipe" gesture has to be able to find the position at which the swipe began, the coordinates of the start point need to be explicitly stored in all swipe gestures. Predicate composition does not suffer from this drawback, as the predicate code is expanded as if by macro expansion, meaning that the predicate's data is injected in the surrounding codeblock's scope. Hence, e. g., the start point's coordinates need not be stored unless there actually is code present in the lexical scope which uses this data. Predicate composition does however suffer from the inverse drawback that duplication in effort in the runtime cannot necessarily be optimized away. In the event composition case, the starting point's coordinates would be computed and stored once for each swipe regardless of the number of places at which swipes are used. In the case of predicate composition, each such place would compute and store the swipe's starting point independently.

The condition elements in a rule are in an implicit conjunction. An event may be bound to any condition element whose conditions it satisfies. The same event may be bound to multiple (or even all) condition elements in a rule, and an event may be bound to the same condition element in different pattern-matching frames. If the valid candidates for condition element `a` are a_1 and a_2, and similarly b_1 and b_2 satisfy all constraints of condition element `b`, then the set of pattern-matching frames for the conjunction of condition elements `a b` is $\{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2)\}$, i. e., the cross product of all valid candidates.

Midas' gesture specification makes heavy use of what Hoste and Signer [1] describe as "dynamic binding". A gesture specification may require the slots of multiple events to be equal, as follows:

```
test e1.x == e2.x
```

This can be viewed as if all affected slots are bound to one and the same logical variable. The runtime system can hence solve this constraint by *unification* of those variables.

Finally, the *absence* of an event satisfying a condition element may also be specified, by negating the condition element. Negation is done using a closed world assumption: when Midas cannot find a matching event in the knowledge base, it is assumed no such event exists. For instance, when no `DeviceOrientationChanged` event may occur with more than 5 degrees rotation, this can be specified as follows:

```
no { dOC = DeviceOrientationChanged
     test dOC.degrees > 5 }
```

Whenever all condition elements of a rule are satisfied, and the bound values pass the constraints imposed by the test, a gesture is considered recognized. When this happens, the actions forming the consequent of the rule are executed. Once deployed, a Midas system automatically responds to gestures as they are being performed, making Midas a reactive system.

$mp \in$ Program	::=	$\overline{t} \mid \overline{m} \mid \overline{r} \mid \overline{p} \mid \overline{f}$	PROGRAMS
$t \in$ Template	::=	**template** t_{id}	TEMPLATES
		$\{\ \overline{\textbf{include}\ u_{id}}$	
		$\overline{(\gamma) s_{id} = v}\ \overline{p} \mid \overline{f}\ \}$	
$m \in$ Module	::=	**module** m_{id}	MODULES
		$\{\ \overline{\textbf{include}\ u_{id}}$	
		$\overline{(\gamma) s_{id} = v}\ \overline{p} \mid \overline{f}\ \}$	
$r \in$ Rule	::=	**rule** $r_{id}\ \{\ \overline{c} \mid \overline{m}\ \}$	RULES
$p \in$ Predicate	::=	**predicate** $p_{id}\ (\overline{l_{id}})\ \{\ \overline{c}\ \}$	PREDICATES
$f \in$ Function	::=	**function** $f_{id}\ (\overline{l_{id}})\ \{\ \overline{e}\ \}$	FUNCTIONS
$c \in$ Condition	::=	$ce \mid te \mid b \mid sf$	CONDITIONS
$m \in$ Modifier	::=	**assert** $t_{id}\ \{\ \overline{s_{id} \Rightarrow e}\ \}$	MODIFIERS
	|	**modify** $l_{id}\ \{\ \overline{s_{id} \Rightarrow e}\ \}$	
	|	**retract** $\overline{l_{id}}$	
	|	**call** $(t_{id}.) f_{id}$	
$ce \in$ Cond. Element	::=	$t_{id}\ \{\ \overline{s_{id} == cv}\ \}$	CEs
$te \in$ Test	::=	**test** $e <\mid\leq\mid==\mid\geq\mid> e$	TESTS
	|	**test** $(l_{id} \leftarrow) p_{id}(\overline{e})$	
	|	**test** $(t_{id} \leftarrow) p_{id}(\overline{e})$	
$b \in$ Bind	::=	l_{id} **=** $(ce \mid e \mid a)$	BINDS
$sf \in$ Special Form	::=	**no** $\{\ \overline{c}\ \}$	SPECIALFORMS
	|	**async** $(tf_{id} \leftarrow) p_{id}$	
	|	**wait** $l_{id}\ v$	
$e \in$ Expression	::=	$(tf_{id}.) f_{id} \mid l_{id}.s_{id}$	EXPRESSIONS
	|	$v \mid b \mid \delta(\overline{e}) \mid$ **(e)**	
$cv \in$ Constr. Value	::=	$v \parallel cv \mid v\ \&\&\ cv \mid \sim v \mid v$	CVs
$v \in$ Value	::=	*nr* | *string* | *symbol* $\mid l_{id}$	VALUES
$a \in$ Array	::=	**[** $\overline{l_{id}} \mid \overline{v}$ **]**	ARRAYS
$\gamma \in$ Type	::=	*int* | *float* | *string*	TYPES
$\delta \in$ PrimF	::=	+ | - | * | / | *mod* | ...	PRIMITIVES

l_{id} $\in$ *VariableName*
r_{id} $\in$ *RuleName*
s_{id} $\in$ *SlotName*
t_{id}, m_{id}, u_{id} $\in$ *Template- or ModuleName*
tf_{id} $\in$ *Template-, Module- or FactVariableName*

Figure 2: Semantic entities of Midas

3.2 Revisiting the Example

Using the syntax introduced above, the "swipe z" and "swipe right" gestures described in subsection 2.1 can be implemented as demonstrated below. Sensor-readings of finger movements are modelled as events of template `Finger`[2]. Event composition is leveraged to specify the "swipe z" gesture in terms of the "swipe right" and "swipe diagonally" gestures.

[2] The name `Finger` is used for conciseness of the examples. Better names for the event might be `FingerDetected` or `FingerChanged`.

We chose to progress bottom-up, specifying "swipe right" first. Inside the rule for this gesture, we opted to start by specifying the endpoints `A` and `D`[3].

```
1  rule SwipeRightRule {
2   A = Finger {state == "appear"}
3   D = Finger {state == "disappear"}
4   test D.fingerId == A.fingerId
5   test D.x > A.x + 50mm
6   test abs(A.y - D.y) <= 20mm
7   test temporallyAfter(D, A)
```

Dynamic binding is used in line 4 to allow the gesture to be performed with any finger, but to ensure that all matching events originate from the same finger: `A`'s `fingerId` is not restricted. The `fingerId` of `D` is restricted. However, it is not fixed to a specific finger, but only to be equal to the `fingerId` of `A`.

To prohibit multiple small strokes to be erroneously recognized as one long stroke, spurious lifting of the finger has to invalidate the gesture. We employ negation to express this, by adding the keyword `no` to indicate the mandatory absence of an early disappearance event such as `E` in Figure 1. The following lines demonstrate this:

```
8   no { E = Finger {state == "disappear"}
9        test E.fingerId == A.fingerId
10       test temporallyInBetween(E, A, D) }
```

Similar code restricts the intermediate `Finger` positions vertically within the 20mm bound, excluding too far events like `F` in Figure 1:

```
11  no { F = Finger
12       test abs(A.y - F.y) > 20mm
13       test F.fingerId == A.fingerId
14       test temporallyInBetween(F, A, D) }
```

Finally, list-oriented constructs are employed to verify that the `x`-value of the time-ordered intermediate `Finger` events is non-decreasing. We use the `listOf` predicate to construct a list of all `Finger` events temporally in between events `A` and `B`, requiring the `fingerId` to be equal to event `A`'s `fingerId`:

```
15  fs = listOf(Finger).after(A)
16                     .before(D)
17                     .unified("fingerId",
18                              A.fingerId)
19                     .ordered(ASCENDING)
20  test allSpatiallyNonDecreasing(fs, "x")
```

Assuming a definition for "swipe diagonally" is provided, and using the previous definition for "swipe right", the "swipe z" gesture can be implemented in terms of the other two. We distinguish between two ways in which gesture composition and abstraction might take place:

Compile-time composition and abstraction packs the definition of a gesture into a named, user defined predicate that can be expanded into the definition of a more complex gesture.

Run-time composition and abstraction reifies the gesture as an event, allowing complex gestures to be built by matching these derived event types.

Both approaches have their merits and drawbacks, and can coexist in one system. For this example, we employ the second approach. To that effect, the `SwipeRightRule` is extended with a *modifier* action which adds a `SwipeRight` event to the knowledge base:

```
21  assert SwipeRight {
22     startTime ⇒ A.time,
23     endTime ⇒ D.time,
24     ... rest omitted ... }
25 }
```

Notice that, unlike traditional rule-based languages, Midas allows condition elements and modifier actions to be interleaved. A rule does not just trigger some actions when it is completely matched; intermediate actions trigger as soon as all condition elements up to that action are matched. In the absence of such a system, the programmer would have to manually 'lift' the scope built up while matching a partial gesture into a custom event, to carry it over to the rules recognizing the remainder of the gesture.

A rule adding a `SwipeDiagonally` event should be created similarly. Subsequently, the "swipe z" gesture can be defined as follows:

```
26 rule SwipeZRule {
27  stroke1 = SwipeRight
28  stroke2 = SwipeDiagonally
29  test stroke2.deltaX < 0
30  test stroke2.deltaY < 0
31  test spatiallyNear(stroke2.startX,
32                     stroke2.startY,
33                     stroke1.endX,
34                     stroke1.endY)
35  test temporallyNear(stroke2.startTime,
36                      stroke1.endTime)
37  stroke3 = SwipeRight
38  test spatiallyNear(stroke3.startX,
39                     stroke3.startY,
40                     stroke2.endX,
41                     stroke2.endY)
42  test temporallyNear(stroke3.startTime,
43                      stroke2.endTime)
44  assert SwipeZ {
45     startTime ⇒ stroke1.startTime,
46     endTime ⇒ stroke3.endTime,
47     ... rest omitted ... }
48 }
```

3.3 Evaluation Model

By relying on a declarative domain specific language, the gesture specifications need only explain what to recognize, and not how to recognize the gesture. An inference engine can optimize the actual recognition process. The RETE algorithm [10] provides a way of doing this. RETE decomposes the problem of matching events[4] to declarative rules by compiling the rules into a directed acyclic graph (DAG) that guides the recognition. Nodes in that graph represent data filtering or combination; edges represent the flow of data. Individual events are propagated to all entry-nodes of a graph, and subjected to tests to establish whether they satisfy the condition element represented by that entry-node. Valid candidates for one condition element are then combined in a cross product with candidates for the next condition elements. Constraints are enforced by filtering out combinations that violate explicit predicates or whose slots which should be equal fail to unify.

Figure 3 illustrates part of the RETE graph for the example from subsection 3.2 up till line 10. The stream of primitive `Finger` events is duplicated to all nodes representing condition elements capturing `Finger` events. Candidates for `A` are selected by discarding all events whose `state` is not `"appear"`. Similarly, candidates for `D` and `E` are selected by filtering out all events whose `state` is not `"disappear"`. Next, all candidates for `?A?` are combined

[3] The variables are named to coincide with the markings in Figure 1, and are kept short because of the constraints of printed media.

[4] The RETE algorithm normally acts on logical "facts". However, "events" are nothing but facts 'in time', i. e., facts which have a temporal component.

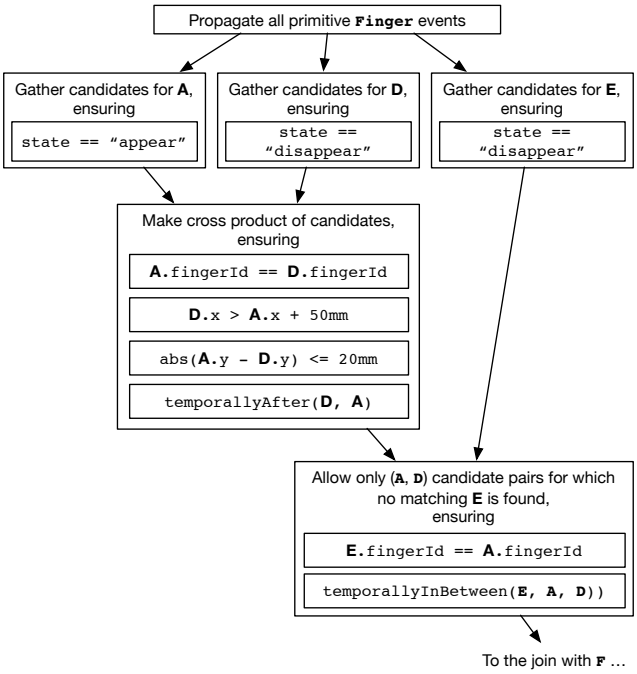

Figure 3: Conceptual illustration of the processing of the "swipe right" gesture, from line 1 to line 10

with all candidates for D. The spatial and temporal constraints on this pair are verified, and pairs meeting the requirements are propagated, to be combined with the candidates for E. This time, a *negated* combination is performed, i. e., only the pairs of candidates for A and D for which *no* triple with a candidate for E can be formed given the constraints, are considered for further processing.

When implemented naively the process's complexity would be exponential in the number of condition elements. This issue is alleviated in RETE by storing intermediate results, so-called "partial matches", with the RETE DAG's nodes. The complexity can be reduced from quadratic in the number of events joined in a rule, to linear in the number of events joined per rule *per time an event is added*. In short, the cost is spread over the individual additions of new events to the knowledge base, instead of fully incurred each time. Gupta and Forgy [11] demonstrated that the tradeoff of storing partial matches to reduce the time it takes to recompute the state pays off in production systems when the state is *stable*, i. e., when the majority of the state remains true when new facts arrive. In the case of gesture recognition this is trivially the case: since events do not become untrue when more events arrive, the entire event history remains stable in the presence of new gesture data.

4. Validation

We presented how the Midas declarative gesture programming language leverages software engineering principles to improve the process of designing gesture-based interactions. By reducing boilerplate code and abstracting over the context maintenance, Midas reduces the amount of tedious to write, yet error-prone code that is required. The tried and tested advantages of modularization are ported to the the gesture programming domain. Midas further offers the means to handle domain-specific requirements such as spatio-temporal reasoning, dealing with concurrency, supporting conflicts in overlapping gestures, and identifying primitive input.

Midas has been fruitfully demonstrated in a number of real-life applications, including during live music performances [12], augmented gaming [13] and powerpoint presentations. During a music performance, many artists rely on projection-based visualisations and light shows to enhance 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. Therefore, expressive control of indirect augmented reality during multiple live music performances was provided to the artists by means of gestures implemented in Midas. Challenging gesture recognition conditions, such as a having single gesture reference point (i. e., no training data), filtering excessive similar movements and dealing with multiple artists, were well handled by our engine. A live performance was showcased in 2012 to an audience of about 1500 people. Midas and Mudra also enabled a novel interactive two-player water game that allows kids and young adults to "fight" in a virtual world with actual physical feedback. Different software policies are provided by the engine in order to control the rule activations such that the water effects reflect the user's intention. Furthermore, Midas has been used as a compilation target for gesture authoring tools [14] which further validates the expressiveness of our approach.

The Mudra framework has been deployed for handling multi-touch gestures, 3D hand gestures using Sun SPOTS [5], as well as full-body skeleton tracking using Microsoft Kinect [6] technology. The declarative nature of Midas, coupled with the modular approach of Mudra has allowed individual parts of the architecture to be replaced, e. g., to increase performance by enabling the rule engine to employ multithreading [15], or to distribute workload over multiple machines [16].

5. Related Work

5.1 Gesture-based Interaction Languages

The "Gesture Definition Language", or "GDL" by Khandkar and Maurer [17] offers some modularity, in the sense that new gestures can be added without knowing about the innards of other gestures, yet composition is not supported. The "Gesture Description Language", also abbreviated to "GDL", by Echtler et al. [18] suffers the same limitation. Furthermore, no support for finger identification or spatio-temporal relations between fingers exists in GDL. This issue is not resolved in their follow-up work GISpL, the Gestural Interface Specification Language [19].

The GeForMT [20] declarative gesture specification language also lacks composition, and makes it impossible to refer to events. This entails there is no way to specify that e. g. two events should originate from the same finger.

The Proton [21] language follows a distinctly different approach, by requiring the gestures to be described as regular expressions over an event stream. Proton necessarily suffers from the restrictions of regular languages, but offers an interesting feature: it is able to detect overlapping gesture specification. Unfortunately, the only way Proton responds to overlaps is to refuse the overlapping gestures. Furthermore, the plain sequence of Kleene operators and disjunctions offer no means of modularisation or composition whatsoever. GestIT [22] extends the Proton language for better handling of partially overlapping gestures. Still, control over temporal constraints is lacking.

[5] *Sun SPOT World*, Oracle, access date: 18 August 2014 http://sunspotworld.com/

[6] *Xbox Kinect: Full Body Gaming and Voice Control*, Microsoft Corp., access date: 18 August 2014 http://www.xbox.com/kinect/

Finally, Interactive Cooperative Objects (ICO) [23] formally describes multi-touch and multimodal interaction based on Petri nets. This greatly improves debugging, reliability, and scalability in terms of complexity. ICO can handle overlapping and concurrent gestures through explicit fork and join operations in the Petri net. Unfortunately, the gesture interaction designer needs to model all possible transitions. ICO offers a visual editor to help keep track of the explosion of cases. As we argued before, a language offers more flexibility. Concretely, ICO's approach does not provide spatial abstractions.

5.2 Other Gesture Specification Approaches

In addition to gesture definition languages, other approaches exist. QuickSet [24] is one example. QuickSet is still text-based, though sacrifice the typability of plain text.

Another category of approaches offers graphical tools to define the gestures. Tablatures [21] is a graphical tool that creates Proton [21] expressions. EventHurdles [25] stands on its own, but offers only limited support for multi-touch and multi-stream gestures.

A significantly different approach can be found in template matching. Techniques such as Rubine [26], Dynamic Time Warping [27], the $1 recognizer [28], or Protractor Li [29] offer "black boxes" which return a distance between a set of events and prerecorded sample sets – the "templates". Template matchers can only decide whether a gesture was sufficiently similar to one of the samples. Distinguishing between two gestures is done by determining which one was closest. No semantic information of the gesture is utilized when recognizing it. This makes debugging false positives and false negatives difficult.

A final category of approaches can be classified as machine learning. Examples include artificial neural networks, support vector machines, and Hidden Markov Models [30]. While they offer the same functionality as template matchers, they differ in the fact that they form a single model from the annotated sample set. Like template matchers, machine learning approaches are largely black boxes. Correct classifications and misclassifications cannot easily be traced to specific samples in the learning set. A semantic description of the gestures is not present.

Acknowledgments

Thierry Renaux and Lode Hoste are supported by a doctoral scholarship of the Agency for Innovation by Science and Technology in Flanders (IWT), Belgium.

References

[1] L. Hoste, B. Signer, Criteria, Challenges and Opportunities for Gesture Programming Languages, in: Proceedings of EGMI 2014, 1st International Workshop on Engineering Gestures for Multimodal Interfaces, Rome, Italy, 22–29, 2014.

[2] C. Scholliers, L. Hoste, B. Signer, W. De Meuter, Midas: A Declarative Multi-Touch Interaction Framework, in: Proceedings of the 5th International Conference on Tangible, Embedded, and Embodied Interaction, TEI '11, Funchal, Portugal, 49–56, 2011.

[3] P. Haller, M. Odersky, Event-Based Programming Without Inversion of Control, in: D. Lightfoot, C. Szyperski (Eds.), Modular Programming Languages, vol. 4228 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, ISBN 978-3-540-40927-4, 4–22, 2006.

[4] J. Edwards, Coherent Reaction, in: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, OOPSLA '09, ACM, New York, NY, USA, ISBN 978-1-60558-768-4, 925–932, 2009.

[5] F. Echtler, G. Klinker, A Multitouch Software Architecture, in: Proceedings of the 5th Nordic Conference on Human-computer Interaction: Building Bridges, NordiCHI '08, ACM, New York, NY, USA, ISBN 978-1-59593-704-9, 463–466, 2008.

[6] P. Ramanahally, S. Gilbert, T. Niedzielski, D. Velázquez, C. Anagnost, Sparsh UI: A Multi-Touch Framework for Collaboration and Modular Gesture Recognition, in: ASME-AFM 2009 World Conference on Innovative Virtual Reality, American Society of Mechanical Engineers, 137–142, 2009.

[7] T. Richardson, L. Burd, S. Smith, Guidelines for supporting real-time multi-touch applications, Software: Practice and Experience ISSN 1097-024X.

[8] T. Hammond, R. Davis, LADDER, A Sketching Language for User Interface Developers, Computers & Graphics 29 (4) (2005) 518–532, ISSN 0097-8493.

[9] L. Hoste, B. Dumas, B. Signer, Mudra: A Unified Multimodal Interaction Framework, in: Proceedings of the 13th International Conference on Multimodal Interfaces, Alicante, Spain, ISBN 978-1-4503-0641-6, 97–104, 2011.

[10] C. L. Forgy, Rete: A fast algorithm for the many pattern/many object pattern match problem, Artificial Intelligence 19 (1) (1982) 17–37, ISSN 0004-3702.

[11] A. Gupta, C. Forgy, Measurements on production systems, CMU-CS-83-167 .

[12] L. Hoste, B. Signer, Expressive Control of Indirect Augmented Reality During Live Music Performances, in: Proceedings of NIME 2013, 3th International Conference on New Interfaces for Musical Expression, Daejeon + Seoul, Korea Republic, 2013.

[13] L. Hoste, B. Signer, 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 '14, Munich, Germany, ISBN 978-1-4503-2635-3, 173–176, 2014.

[14] L. Hoste, B. De Rooms, B. Signer, Declarative Gesture Spotting Using Inferred and Refined Control Points, in: Proceedings of ICPRAM 2013, 2nd International Conference on Pattern Recognition Applications and Methods, Barcelona, Spain, 144–150, 2013.

[15] S. Marr, T. Renaux, L. Hoste, W. De Meuter, Parallel Gesture Recognition with Soft Real-Time Guarantees, Science of Computer Programming .

[16] J. Swalens, T. Renaux, L. Hoste, S. Marr, W. De Meuter, Cloud PARTE: Elastic Complex Event Processing Based on Mobile Actors, in: Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! '13, ACM, New York, NY, USA, ISBN 978-1-4503-2602-5, 3–12, 2013.

[17] S. H. Khandkar, F. Maurer, A Domain Specific Language to Define Gestures for Multi-touch Applications, in: Proceedings of the 10th Workshop on Domain-Specific Modeling, DSM '10, ACM, New York, NY, USA, ISBN 978-1-4503-0549-5, 2:1–2:6, 2010.

[18] F. Echtler, G. Klinker, A. Butz, Towards a Unified Gesture Description Language, in: Proceedings of the 13th International Conference on Humans and Computers, HC '10, University of Aizu Press, Fukushima-ken, Japan, Japan, 177–182, 2010.

[19] F. Echtler, A. Butz, GISpL: Gestures Made Easy, in: Proceedings of the Sixth International Conference on Tangible, Embedded and Embodied Interaction, TEI '12, ACM, New York, NY, USA, ISBN 978-1-4503-1174-8, 233–240, 2012.

[20] D. Kammer, J. Wojdziak, M. Keck, R. Groh, S. Taranko, Towards a Formalization of Multi-touch Gestures, in: ACM International Conference on Interactive Tabletops and Surfaces, ITS '10, ACM, New York, NY, USA, ISBN 978-1-4503-0399-6, 49–58, 2010.

[21] K. Kin, B. Hartmann, T. DeRose, M. Agrawala, Proton: Multitouch Gestures As Regular Expressions, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '12, ACM, New York, NY, USA, ISBN 978-1-4503-1015-4, 2885–2894, 2012.

[22] L. D. Spano, A. Cisternino, F. Paternò, G. Fenu, GestIT: A Declarative and Compositional Framework for Multiplatform Gesture Definition, in: Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS '13, ACM, New York, NY, USA, ISBN 978-1-4503-2138-9, 187–196, 2013.

[23] A. Hamon, P. Palanque, J. L. Silva, Y. Deleris, E. Barboni, Formal Description of Multi-touch Interactions, in: Proceedings of the 5th

ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS '13, ACM, New York, NY, USA, ISBN 978-1-4503-2138-9, 207–216, 2013.

[24] P. R. Cohen, M. Johnston, D. McGee, S. Oviatt, J. Pittman, I. Smith, L. Chen, J. Clow, Quickset: Multimodal interaction for distributed applications, in: Proceedings of the fifth ACM international conference on Multimedia, ACM, 31–40, 1997.

[25] J.-W. Kim, T.-J. Nam, EventHurdle: Supporting Designers' Exploratory Interaction Prototyping with Gesture-based Sensors, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '13, ACM, New York, NY, USA, ISBN 978-1-4503-1899-0, 267–276, 2013.

[26] D. Rubine, Specifying Gestures by Example, SIGGRAPH Comput. Graph. 25 (4) (1991) 329–337, ISSN 0097-8930.

[27] T. Darrell, A. Pentland, Space-Time Gestures, in: Proceedings of CVPR 1993, New York, USA, 335–340, 1993.

[28] J. O. Wobbrock, A. D. Wilson, Y. Li, Gestures Without Libraries, Toolkits or Training: A $1 Recognizer for User Interface Prototypes, in: Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology, UIST '07, ACM, New York, NY, USA, ISBN 978-1-59593-679-0, 159–168, 2007.

[29] Y. Li, Protractor: A Fast and Accurate Gesture Recognizer, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '10, ACM, New York, NY, USA, ISBN 978-1-60558-929-9, 2169–2172, 2010.

[30] A. D. Wilson, A. F. Bobick, Parametric Hidden Markov Models for Gesture Recognition, IEEE Transactions on Pattern Analysis and Machine Intelligence 21 (9).