Towards Automated Prototyping of Gesture Recognition Systems for Wearable Devices using Inertial Sensors

Johann-P. Wolff, University of Rostock, 18051 Rostock, Germany, johann-peter.wolff@uni-rostock.de
Florian Grützmacher, University of Rostock, 18051 Rostock, Germany, florian.gruetzmacher@uni-rostock.de
Rainer Dorsch, Bosch Sensortec GmbH, Gerhard-Kindler-Straße 9, 72770 Reutlingen/Kusterdingen, Germany, Rainer.Dorsch@bosch-sensortec.com
Rolf Kaack, Bosch Sensortec GmbH, Gerhard-Kindler-Straße 9, 72770 Reutlingen/Kusterdingen, Germany, Rolf.Kaack@bosch-sensortec.com
Lars Middendorf, Bosch Sensortec GmbH, Gerhard-Kindler-Straße 9, 72770 Reutlingen/Kusterdingen, Germany, Lars.Middendorf@bosch-sensortec.com
Christian Haubelt, University of Rostock, 18051 Rostock, christian.haubelt@uni-rostock.de

Abstract

Embedded always-on gesture recognition systems demand for a specific trade-off in terms of computational load, memory footprint, and accuracy. Hence, special care has to be taken in choosing the right recognition algorithms. But even with a suitable recognition algorithm, gesture development is still a manual, labor-intensive, and error-prone task, deteriorating the overall design quality. In this paper, we propose a framework for automated prototyping of embedded gesture recognition systems for mobile and wearable devices using inertial sensors. First results show the viability and scalability of the approach, which might serve as a basis of future automatic design of sensor-based embedded gesture recognition systems.

1 Introduction

Wearable devices are a well suited platform for gesture interaction, allowing users to control their digital neighbourhood with a shake of the arm or a nod of the head. Inertial sensors can capture these movements and poses; their data can be used to recognize gestures. Gesture recognition is a well established research area and has seen various real-life applications. As a specialization of pattern recognition, its goal is to identify certain gestures in available input signals, e.g. inertial sensor data. Looking only at dynamic gestures (as opposed to static gestures, i.e. poses), most proposed systems use probabilistic algorithms to first learn and then recognize gestures [7]. These algorithms provide a large amount of flexibility: implementing new gestures often requires only appropriate recordings of the new gestures and some extra computation time for training.

Developing gesture recognition systems for wearable and mobile computers however does require a different trade-off: as the algorithms have to perform always-on gesture recognition, they will need to run on specialized hardware, like co-processors or sensor hubs [11]. Thus, they have to meet memory, performance, and power constraints. Flexibility becomes a minor issue. As a consequence, the gestures are not user programmable but pre-loaded by device vendors. On the other hand, specialized hardware requires specialized implementation. Developing these specialized implementations is time-consuming and error prone - even more so, when the next generation of a device is equipped with different specialized hardware and the implementation has to be revised. This adaptability of gesture designs to new platforms is especially important for mobile devices.

Inertial sensors are commonly used in mobile devices for always-on gesture recognition. They allow fast and convenient control of a smartwatch display, music playback, and many

ISBN 978-3-8007-4919-5 © VDE VERLAG GMBH · Berlin · Offenbach

other functions. These sensors can provide information on movement, orientation and pose while, unlike camera-based solutions, being mostly environment-independent.
Development for these mobile devices is known for a short time to market and rapidly changing physical attributes between device generations. These physical attributes however have a large impact on the way gestures can be performed: weight and screen size affect the speed and amplitude of gestures; the physical location of the sensor in the mobile device decides whether a rotation around the device center is measured as just a rotation or as a rotation and a linear acceleration. Thus, gesture recognition software development (short: gesture development) needs rapid development and adaptation cycles. Today no automatic development approach for always-on gesture recognition on sensor hubs in mobile devices is available.

Our goal is to automate the development of gesture recognition software for mobile devices. In this paper, we present an automated approach that allows user-centered development. The gesture developer is enabled to develop gestures without detailed knowledge of either recognition algorithms or target platform. Using our approach, the gesture idea can be developed into a gesture recognition algorithm automatically, providing a prototypical implementation. This implementation can then be tested and incrementally refined. The result of this workflow is a gesture prototype that the developer can use for preliminary user acceptance tests, as a start point for a more complex implementation or even as a benchmark for the final gesture recognition system. Central to this idea is an adaptable gesture recognition algorithm and a target platform agnostic gesture description, which we present in the form of the Gesture Description Language (GDL).

2 Related Work

Like the gesture recognition algorithm proposed in this paper, there have been other systems that utilize Finite State Machines (FSM) to recognize hand gestures. In [12] and [9], the dominant 2D hand motion direction is extracted from a video sequence and used to recognize dynamic hand gestures using simple FSM. The speed, with which gestures are performed, is not considered here, only the sequence of movement is relevant to the recognition result. Our approach not only considers specified timings for gestures, but also includes developer knowledge of the gesture in the recognition algorithm.

In [5], 3D gestures performed above a 3D position tracker are recognized similarly to [12], but here Regular Expressions (regex) are used to define and describe gestures. These gesture descriptions however lack timing information, which is necessary to exclude possible too-fast or too-slow false positive recognition results. Our approach addresses this problem. Furthermore, we mitigate a problem apparent in the paper: superpositioned movement like moving up and left at the same time is only recognized as up-left, but not as up *and* left. Given the imprecise way in which humans perform gestures, this reduces recognition accuracy, especially for more than two-dimensional data.

Touchscreen gestures are also in the focus of [6], where these 2D movements are described in regex. A graphical user interface is described to easily design gestures without the need to write regexes themselves. The recognition of the gestures is done using Non-deterministic Finite Automata (NFA), which allows static analysis used to intersect regexes of different gestures and reduce computational effort when recognizing several gestures. This approach to gesture design and implementation is broadly similar to our proposed system. However, as the use of inertial sensors yields a much higher data complexity, the amount of gesture knowledge required from the gesture developer is also larger. Supporting gesture

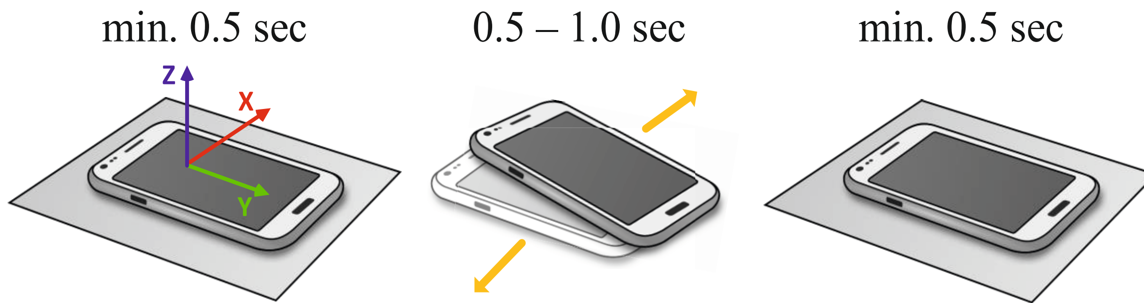

Fig. 1: Phases of a simple Glance-gesture

developers to manage this complexity is a challenge not addressed in [6], which is central to our approach.

Another approach to describe gestures in a formal and automatically evaluatable representation is proposed in [10]. Here, 'functional' FSMs are introduced that are modeled in a functional programming language. Furthermore, the parameters with which raw sensor signals are transformed to the input of the functional FSMs are optimized using cultural algorithms. While this approach is the first one to acknowledge the need for a gesture description language, the implementation is not an intermediate language but executable code. This increases developer effort and slows down prototyping. Also, the naturally sloppy execution of gestures in real life applications is not addressed in [10], while we offer several techniques to include variations in performance at gesture design time.

Our proposed GDL shares many similarities with property languages from the domain of verification. In model checking approaches temporal logics are often used to specify functional properties. Temporal logics extend propositional logics by means of temporal (and sometimes modal) operators. Properties specified in temporal logics can be transformed into monitors, which check at runtime (or during simulation) if the system under verification obeys to the given property. Among others, PSL (Property Specification Language [4]) is one of the prominent representatives of languages used in systems verification. Our proposed GDL is a subset of PSL, which has been adopted to the domain of gesture recognition. As a consequence, our rapid prototyping approach shares some similarities with the automatic generation of monitors from PSL descriptions. But again, our approach is adopted to the domain of signal processing and gesture recognition, thus a reduction of language features in order to streamline language use is reasonable.

3 Gesture Development Kit

As a tool for gesture development and rapid prototyping, we introduce the Gesture Development Kit (GDK). It aims to support development of inertial sensor-based gestures for resource constraint devices, like smartphones, wearables and Internet-of-things devices. The GDK is a modular framework, which supports the gesture developer with visualization and testing tools and enables a development flow using a domain specific language for automated generation of prototypical implementation of recognition software. In this paper, we present the domain specific language and the code generation method derived from it.

3.1 Motivating example

Before we define the GDL more formally, we will use an example of a specific gesture to motivate the proposed structure of the GDL. While not applicable to all scenarios, many gestures can be described as a linear sequence of movements and poses. The *Glance* gesture for example is a smartphone gesture that activates the display and shows

ISBN 978-3-8007-4919-5 © VDE VERLAG GMBH · Berlin · Offenbach

```
1      Sample = SensorSample | DerivedSignal;
2      SampleExpression = Sample, RelOp, Value | Sample, RelOp, Sample;
3      RelOp = "<" | ">" | "=" | "<=" | ">=" | "/=";
4      Symbol = SampleExpression | Symbol, "||", Symbol | Symbol, "&&", Symbol;
5      GestureAtom = Symbol | GestureAtom, "{", minTime, ",", maxTime, "}" |
       GestureAtom, "{",minTime, ",", "}" | GestureAtom, "{", ",", maxTime, "}";
6      GestureExpression = GestureAtom | GestureExpression, GestureExpression;
```

Fig. 2: GDL GestureExpression expressed in Extended Backus-Naur-Form (EBNF)

notifications. It can be implemented as a sequence of non-movement (e.g. laying on a table), some short but significant movement (e.g. shaking the phone) and then non-movement again. This gesture is illustrated in Figure 1.The remainder of this paper uses this simple gesture as an example. It is defined using the proposed GDL in Section 3.3.

3.2 Gesture Description Language

The GDL formalizes a gesture description (like the informal one presented in Section 3.1). It is an easily understandable representation of multi-phase gestures and includes both the signal characteristics used to identify each phase and the timing of the gesture. Goal of the GDL is the transformation of the gesture developer's idea of the gesture to a format that is non-ambiguous and allows code generation for real platforms like mobile devices.
The GDL consists of several aspects, which are used to fully describe gestures. There are three major aspects involved: *What happens? For how long?* And: *In what order?* To describe these three aspects, the gesture is divided into subsequent phases that can be identified through signal characteristics. These phases are henceforth called *GestureAtoms*. They are specified with timing and signal characteristics.

3.2.1 A more formal definition of the GDL

To describe natural movement, the GDL was designed to describe both timings and signal characteristics. The GDL *GestureExpression* is defined as shown in Figure 2.

Line 1 defines the *Sample* as a basic signal information like sensor samples or derived signal information. *DerivedSignal* refers to data derived from sensor data, e.g. filtered or fusioned data. A *SensorSample* in this definition is a single datum that is part of the current sensor data. Both *DerivedSignal* and *SensorSample* are terminal symbols in this definition to ensure readability. In line 2 the *SampleExpression* is defined: A *SampleExpression* is a predicate over one or two samples. It uses a relational operator to compare a *Sample* to a known *Value* or another *Sample*. It can evaluate to *true* or *false*. *Value* is another terminal symbol defined as a real value. Furthermore, line 4 defines the *Symbol*: A *Symbol* is a predicate over *SampleExpressions*. It can evaluate to *true* or *false*. A *GestureAtom* as defined in line 5 is a constrained repetition of a *Symbol*. The parameters of the repetition (*minTime* and *maxTime*) are terminal symbols in this definition. They specify the timing of a gesture and are integers. Finally, line 6 defines the *GestureExpression* is an ordered sequence of *GestureAtoms*.

The GDL is then a text based representation of a *GestureExpression*. As such, it needs to contain all above mentioned information on a gesture. In the GDL, the *GestureExpression* as well as the gesture atoms are formulated as a Regular Expression, while the *Symbols* are defined explicitly. Similar to the use of cultural algorithms in [10], the threshold parameters both in *Symbol*- and *GestureAtom*-generation are obvious targets for later optimization.

 © VDE VERLAG GMBH · Berlin · Offenbach

Relation to PSL: The Property Specification Language formally describes electronic system behaviour and is widely used for specification and verification. The language consists of four layers: Boolean, Temporal, Verification and Modeling Layer. In this paper, we will only look at the first two layers: The Boolean and temporal layers can be used to specify behaviour as properties, which can then be evaluated during simulation. The Boolean layer is used to evaluate conditions in a single evaluation cycle. The temporal layer models temporal behaviour, using either Linear Temporal Logic (LTL) or Sequential Extended Regular Expressions (SEREs). These temporal expressions are evaluated over a series of consecutive evaluation cycles.

The Boolean layer of PSL can be compared to the S*ymbol* definitions in the GDL: both break down conditions on the system state at a single point in time to a Boolean value. The temporal layer of PSL works on independently evaluated terms of the Boolean layer, much like the the regular expression of GDL uses the symbol vector as an input. It could even be said that the regular expressions in GDL are just a more compact nomenclature of SEREs. Each gesture can be interpreted as a property of a sensing device with the fulfillment of the property constituting a recognized gesture.

In the proposed system, only linear, i.e. unbranched regular expressions are currently supported. Furthermore, the definition of gestures through the negation of a described gesture (e.g. a gesture defined as everything but the "Glance" gesture) is not allowed, reducing the feature set of the GDL to that of PSL-FL^{-}. However, this approach of modeling behaviour similar to counterexample-guided abstraction refinement [1] might be beneficial for gesture recognition applications. Future work on GDL will explore this approach. The automated generation of gesture recognition software from GDL descriptions can be compared to the automated generation of monitors from PSL [3]. Instead of the tableau technique [13] for the generation of an NFA, we use a regex transformation based the separation of symbol order and repetition, as described in Section 3.3.

3.3 Recognition Algorithm

With the GDL the developer provides a non-ambiguous definition of the gesture as a sequence of specified movements with a fixed timing.

The task of gesture recognition can then be interpreted as a pattern matching between a series of symbols and a *GestureExpression*, which describes the order and number of repetitions of the same *Symbols*. The *GestureExpression* is the central part of the Gesture Description Language (GDL) that is used in the GDK. The *GestureExpression* itself is realized as a Regular Expression (regex) with a reduced set of operations. The GDL also contains the definitions of all Symbols.

Our example gesture would thus be described in GDL as:

```
1      sym a = (lp(x)<(0.1)) & (lp(y)<(0.1)) & (lp(z)>(0.9))& (lp(z)<(1.1))
2      sym b = lp(sqrt((x'-x)^2 + (y'-y)^2 + (z'-z)^2))
3      regex = a{25,}b{25,50}a{25} ,
```

assuming a sampling rate of 50 Hz, the acceleration unit being *g* and *lp(x)* being a function that implements a low-pass filter.

Regex matching can be done in several ways: Firstly, the regex can be translated into a deterministic finite automaton (DFA). This automaton has states for each and every possible (and relevant) input sequence and is ever only in one of these states. This approach computes very fast, as it only evaluates a single state at a time, but consumes a lot of memory for long sequences. Secondly, backtracking can be employed for regex matching

[2]. This approach consumes less memory but due to its depth-first search can lead to high worst-case execution times. Another approach to regex matching is the use of non-deterministic finite automata (NFA) [8]. A NFA translated from a regex is typically smaller than an equivalent DFA but has a less predictable (and most likely higher) runtime due to concurrent active states. While still being fast and memory-efficient, it does not allow for advanced regex features like backreferences.

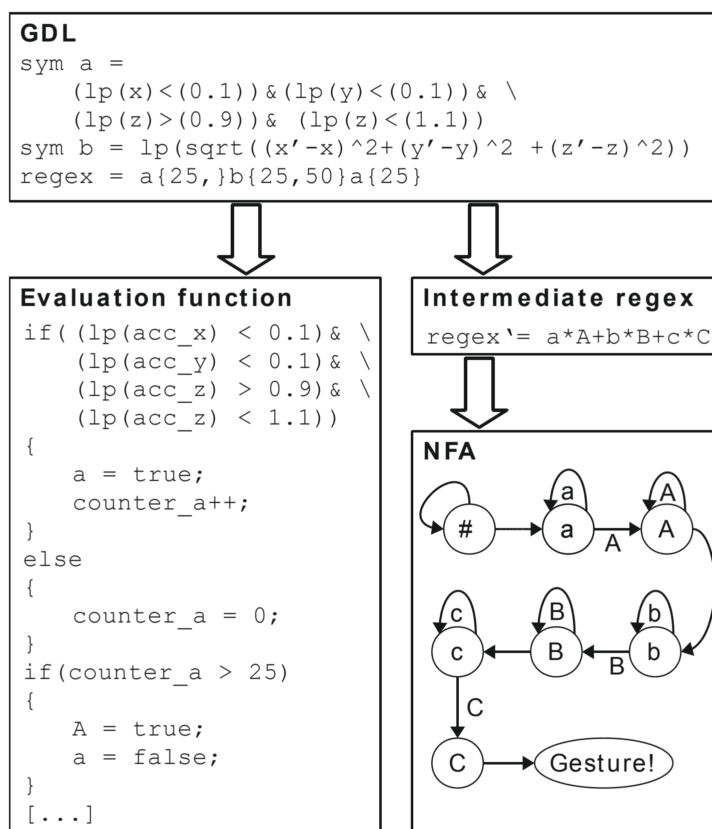

Fig. 3: Processing of the GDL

For the implementation of the regex matching system on embedded systems, we chose an NFA approach based on an implementation provided in [2], but found that the restriction of symbol repetition as a method of time measurement leads to large automata (as do DFAs). While being a good human-readable representation of a gesture, the regex from a GDL is thus unsuitable for NFA generation. We addressed this problem in our GDL processing as depicted in Figure 3: we removed the timing from the regex and included it in the evaluation function. The regex is transformed to a regex without limited repetition, allowing constructed NFAs to be quite small again.

The *Symbols c* and *C* are introduced to express that while they use the same *Symbol*, the first and the second phase of non-movement are distinct in their role in the gesture. The additional *Symbols* can be understood better when looking at the evaluation function presented in Figure 3. This function evaluates both the *SampleExpressions* as well as their timing using simple counters (this will evaluate the *GestureAtoms*), yielding a vector of Boolean values called the symbol vector. In order to enforce the progression from the state associated with the *Symbol* to the state associated with the corresponding *GestureAtom*, the *GestureAtom* deactivates its *Symbol*. This is possible since the GestureAtom already implies the occurrence of the appropriate movement. The results of these *GestureAtoms* as well as the results of the *Symbols* are encoded in the symbol vector. This vector describes which specified movements occur at each sampling time point and whether they are within their specified timing parameters.

The input symbol for the regex matching is the symbol vector. However, the matching does not check for identity between input symbol and literal but whether the bit in the vector corresponding to the literal is *true*. This allows movements, that do not exclude each other (e.g. moving along the x-axis and moving along the y-axis) to be recognized at the same time. The concurrently active states coupled with the concurrent movements allowed by the symbol vector enables a highly flexible and variation tolerant pattern matching algorithm.

4 Evaluation

In order to provide a first assessment of our proposed GDK, we compare recognition software, which has been automatically generated from GDL descriptions, to equivalent

manual C implementations that recognize the same gesture. We use the Glance gesture from before as well as a Pickup gesture, in which the sensing device is picked up and held at a 45° angle. The manual implementations are optimized for the smallest sensor hubs and use a deterministic FSM with a similar counting mechanism for time measurement. In addition, two new GDL gestures are introduced to illustrate the scalability of gesture descriptions and their generated implementations. *Glance 2x* and *Glance 4x* use the regexes $regex = a\{25,\}b\{25{,}50\}a\{25\}b\{25{,}50\}a\{25\}$ and
$regex = a\{25,\}b\{25{,}50\}a\{25\}b\{25{,}50\}a\{25\}b\{25{,}50\}a\{25\}b\{25{,}50\}a\{25\}$,
respectively. Thus, the Glance gesture is extended by additional movement → non-movement phases. Table I shows the length of the different implementations and representations. The manual implementation uses a DFA and similar symbols and timings as the GDL. The GDL consists in both cases of one line for the gesture expression, two lines of symbol declaration and one line to specify the sampling rate. Both implementations describe the used automaton as a series of *if*-statements, in which the appropriate states for the iteration are activated or deactivated. It can be seen that the GDL descriptions are dramatically shorter in terms of Lines Of Code (LOC), as is the purpose of a high-level description language. We argue that this reduction of text, as well as the clearer distinction between gesture progression and the definition of partial movements (i.e. symbols) improves readability and thus development speed. Furthermore, it allows persons unfamiliar with the C programming language to develop gestures for embedded systems. This benefit can be maintained when switching hardware or software architectures if one C programmer adapts the fixed part of the implementation to the new architecture. We argue that the GDL is tailored to the task of gesture description in a way that uses the length reduction to improve understanding.

All four gesture recognition programs were briefly tested on a sensor hub and showed similarly good, but not perfect accuracy. As a second comparison, we compared the compiled code and data memory footprint of the manual and generated implementations for a PC architecture for reproducibility. The results shown in Table I were produced using gcc 8.1.1 for x86_64 and size optimization, i.e. “-Os”. Both implementations use only functions of the standard C library that can typically be found on sensor hubs, namely in *math.h* and *stdint.h*. Neither implementation uses dynamic data memory allocation.

The results show that the generated implementations produce significantly larger programs than the manual implementations, both in terms of code and data memory footprint. However, they are still within the same ballpark - proving that this approach can produce prototype software for small sensor hubs, but through automation at a much lower development effort. The measured memory footprints further show a trend for scalability. The seven states of the original Glance become 11 and 19 states for *Glance 2x* and *Glance 4x*, respectively. As the rest of the implementations of the three Glance variants are identical, we can assume the difference in memory footprint between *Glance 2x* and *Glance 4x* to be

Table I: Lines of code and memory footprint for equivalent gestures in different implementations. *.text* refers to the code memory footprint, *.[ro]data* refers to the sum of *.data* and *.rodata* and thus to the total static data memory footprint

	LOC	LOC Generated Code	.text size [B]	.[ro]data size [B]
Glance (Manual)	127	-	408	18
Pickup (Manual)	143	-	616	18
Glance (GDL)	4	129	801	44
Pickup (GDL)	4	134	743	32
Glance 2x (GDL)	4	187	1016	44
Glance 4x (GDL)	4	301	1473	44

twice the difference between the regular Glance and *Glance 2x*. As can be seen in Table I, the assumption holds at least for this example.

5 Conclusion

In this paper, we propose a framework for gesture development and rapid prototyping and its implementation in the GDK. The major contribution is the formulation of a gesture description language that is expressive in describing sequences of motion yet easier to understand and faster to write than actual code for embedded devices. Through code generation, the GDL description of a gesture can be easily tested on a PC or the actual smart sensor platform. This workflow enables rapid prototyping even for developers unfamiliar with gesture recognition algorithms or embedded systems programming.

This work is supported by BMBF project SAFE4I, grant number 01IS17032O.

References

[1] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In International Conference on Computer Aided Verification, pages 154–169. Springer, 2000.
[2] R. Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...). URL: https://swtch.com/rsc/regexp/regexp1.html, 2007.
[3] M. Gordon, J. Hurd, and K. Slind. Executing the formal semantics of the accellera property specification language by mechanised theorem proving. In Advanced Research Working Conference on Correct Hardware Design and Verification Methods, pages 200–215. Springer, 2003.
[4] IEEE-Commission et al. IEEE standard for property specification language (PSL). Technical report, Technical report, IEEE, 2005. IEEE Std 1850-2005, 2005.
[5] N. Kiliboz and U. Gudukbay. A hand gesture recognition technique for human-computer interaction. Journal of Visual Communication and Image Representation, 28:97–104, 2015.
[6] K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. Proton: Multitouch gestures as regular expressions. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI ’12, pages 2885–2894, New York, NY, USA, 2012. ACM.
[7] A. Sarkar, G. Sanyal, and S. Majumder. Hand gesture recognition systems: a survey. International Journal of Computer Applications, 71(15), 2013.
[8] K. Thompson. Programming techniques: Regular expression search algorithm. Communications of the ACM, 11(6):419–422, 1968.
[9] R. Verma and A. Dev. Vision based hand gesture recognition using finite state machines and fuzzy logic. 2009 International Conference on Ultra Modern Telecommunications and Workshops, 2009.
[10] F. Waris and R. Reynolds. Using cultural algorithms to improve wearable device gesture recognition performance. In Computational Intelligence, 2015 IEEE Symposium Series on, pages 625–633. IEEE, 2015.
[11] J. Wolff, S. Stieber, T. Rankl, R. Dorsch, and C. Haubelt. Improving Always-On Gesture Recognition Power Efficiency for Android Devices UsingSensor Hubs. IEEE International Conference on Embedded and Ubiquitous Computing, 2016.
[12] M. Yeasin and S. Chaudhuri. Visual understanding of dynamic hand gestures. Pattern Recognition, 33(11):1805–1817, 2000.
[13] Jun Yuan, Carl Pixley, and Adnan Aziz. Constraint-based verification. Springer Science & Business Media, 2006.