Using Cultural Algorithms to Improve Wearable Device Gesture Recognition Performance

Faisal Waris
Wayne State University
Dept. of Computer Science
Detroit, USA
fwaris@wayne.com

Robert G. Reynolds, Senior Member IEEE
Wayne State University
Dept. of Computer Science
Detroit, USA
Reynolds@cs.wayne.ed

***Abstract*—Wearable computing devices are now mainstream. Many such devices have capable MEMS sensors that can be exploited for recognizing dynamic, in-the-air gestures. The somewhat limited compute power and battery life of today's devices requires a computationally efficient approach to gesture recognition; one that can be effectively used inside an app running on standard, off-the-shelf hardware, such as an Android Smartwatch. The goal of this project is to test the feasibility of this idea. In a two-phased approach, a class of finite state machines (FSM [1]) for gesture recognition were first constructed which were then tuned for higher accuracy with the help of training data and a suitable optimization method. A novel approach is presented that leverages techniques from functional programming languages to define rich yet compact FSM description. In order to demonstrate effectiveness, a prototype gesture recognition system for an automotive scenario, utilizing an Android Smartwatch app, was developed. Then, the system was tuned using an evolutionary optimization algorithm, Cultural Algorithms, with the help of experimentally derived training data. The blended approach achieved a 77% gesture recognition accuracy. The 'functional' FSM (FnSM) are human defined but machine optimized with Cultural Algorithms. By the blending of the two approaches, an improved balance between computational requirements and recognition accuracy was achieved.**

I. INTRODUCTION

Wearable computing is now in mainstream adoption. By far the most popular wearable computing devices on the market today are fitness bands such as the Nike FitBit [1]. However smartwatches from Apple, Google and others are also gaining rapid adoption. IDC predicts that Apple will ship over 20 million watches in 2015 [2]. Smartwatches are interesting in that it is relatively easy to build and deploy apps to these devices thus opening up the possibility of widespread consumer-grade wearable computing applications.

One such application – a smartwatch app that recognizes motion based gestures to control a companion smartphone app is explored here. The premise is that such an application would be suitable for automotive use, given its potential for reducing driver distraction.

Today's automotive Human Machine Interfaces (HMI) require one to control a rich and complex set of options, settings and functionality – the complexity of which can only grow over time. At this time speech recognition systems are considered to be the state-of-the-art in automotive HMI technology. Speech recognition is an eyes-on-the road and hands-on-the-wheel compatible interface, regarded as ideal for reducing driver distraction. However, speech recognition systems are not perfect and can exhibit many short comings when applied in the real-world automotive domain. The automotive research firm JD Power cited speech recognition as the top consumer complaint in 2014 [3]. The trend now is towards gesture control. New vehicles from Audi, Daimler-Benz, BMW and Cadillac can now be optioned with gesture-based HMI control (Figure 1).

Figure 1: Cadillac CT6 gesture pad – GM Creative Commons License

At the same time, automakers are implementing Application Programming Interfaces (APIs) to allow better smartphone integration with the car's infotainment system, e.g. Apple CarPlay™, Android Auto™ (Figure 2) and Ford AppLink™.

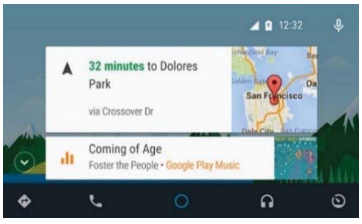

Figure 2: Android Auto – source: android.com/auto

[1] The abbreviations FSM and FnSM are used as both singular and plural, depending on the context

978-1-4799-7560-0/15 $31.00 © 2015 IEEE
DOI 10.1109/SSCI.2015.97

Given the three converging trends of: a) mainstream adoption of wearable computing devices; b) automotive gesture based HMI control ; and c) and automotive APIs for smartphones – *gesture based control of automotive HMI seems like a plausible application of wearable computing technology*.

As a preliminary step towards exploring such an application, a gesture recognition app on the Sony Smartwatch 3 Android smartwatch to control a mock HMI on the companion smartphone is implemented here. The Sony watch has the customary assortment of MEMS (microelectromechanical systems) sensors such as accelerometer, gyroscope, and magnetometer which provide the input data for gesture recognition. The designs are implemented as parameterized Finite State Machines, and Cultural Algorithms are employed to establish the parameters for the systems functionality. Since parameter selection here must support many varying constraints, a data intensive evolutionary learning algorithm, Cultural Algorithms is used within the interface in order to support parameter learning. The advantages of placing an evolutionary computation mechanism in the interface "loop" will be discussed later.

The outline of the paper is as follows. First, section II describes the overall approach taken, given the constraints and challenges foreseen. Then, section III introduces the Finite State Machine design employed. Next in section IV some of the most popular methods used for dynamic gesture recognition are reviewed and contrasted with the prototype gesture recognition technique. Section V provides a background for the specific evolutionary computing approach that was used for the optimization of the gesture recognition engine parameters, Cultural Algorithms. Section VI gives the results of the experiments employed to test the prototype system. Section VII presents the conclusions.

II. GESTURE RECOGNITION STRATEGY

To begin, a gesture vocabulary that balances a number of constraints and challenges to make it viable for the driver of the vehicle must be selected. It should require that the driver make only short and simple forearm and wrist movements in order to expedite the recognition process.

A. Constraints and Challenges

Gesture recognition is a well-understood problem however there are several challenges to consider in relation to the chosen research scenario:

- Limited computational power and battery life of wearable devices.
- Compounding effect of the motion of the car on the device sensor readings.
- Limited space is available to the driver to perform the actual gestures. This is especially acute for left-hand-drive vehicles with the watch worn on the left wrist because of the short distance between the steering wheel and the driver's side window.
- The large number of possible HMI actions to choose from.

Given these constraints / challenges, the following approach was selected as the basis for the gesture recognition system here:

a) Use a vocabulary composed of simple gestures where each individual gesture can be easily performed by the driver in a confined space using mainly the motion of the wrist and the forearm.
b) Use the gesture language to drive a hierarchical menu so as to be able to traverse a large set of choices using a small set of actions.
c) Use computationally efficient Finite State Machines (FSM) rather than a more elaborate scheme such as Hidden Markov Models (HMM), for gesture recognition. Innovative programming techniques and offline optimization of the FSM using a Cultural Algorithm implementation, will compensate for the relative lack of FSM sophistication.

B. Gesture Vocabulary

A driver of a left-hand-driven vehicle, wearing the smartwatch on the left wrist, can only make a limited set of gestures (using his/her left forearm) within the confined available space. After some casual experimentation the gesture vocabulary chosen is a set of 6 simple gestures shown in Table 1.

Table 1: Gesture Vocabulary

Gesture	***Movement***	***Meaning***
Twist	Twist wrist left and right	Enter gesture recognition mode. The driver uses this gesture to tell the app that he/she is interested in using gesture recognition. The sensor input is then tracked more closely to recognize other gestures listed in this table.
Escape	Point forearm downwards (below the horizontal plane)	Exit gesture recognition mode. Sensor input is not tracked except to recognized the Twist gesture (for re-entering gesture recognition mode)
Left	Twist wrist left (counter-clockwise) from the vertical position. The initial position should be where the watch face is relatively perpendicular to the ground.	This is a navigation gesture to move the cursor in the menu hierarchy in the backward or left direction from its current position (if possible)
Right	Twist wrist right (clockwise) from the vertical position	Navigation gesture to move the cursor forward or right from its current position (if possible)
Swipe	Sway arm left or right while keeping the watch face vertical to the ground	Exit current level or go up one level
Tap	Sway arm up and down while keeping the watch face vertical to the ground	Make a selection. If the current item is a sub menu then enter the sub menu. If the item is a leaf then perform the corresponding action.

The performance of a sequence of individual gestures will allow the operator to traverse a relatively large choice set, given an appropriate hierarchical organization. Additional research and experimentation is required to estimate the maximum number of choices that are feasible under such a method.

Although not implemented in the current experimental app, an audio feedback mechanism to announce / acknowledge the traversed menu items, will be required in a production implementation. (Vibration feedback for each recognized gesture is implemented here.)

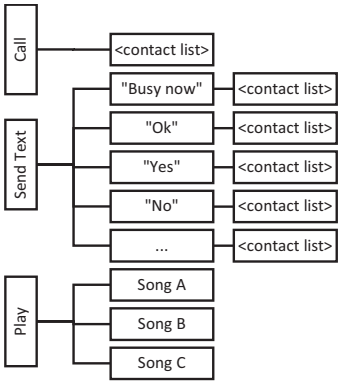

Figure 3: Mock HMI Menu Hierarchy

C. Menu Hierarchy

In order to emulate a fairly typical automotive use scenario, the menu hierarchy shown in Figure 3 was used for the test application. There are three top level menu choices: a) call a contact; b) send a canned text message to a contact; and c) play a song from a list of available songs.

The mock HMI only contains short lists however longer lists, such as a typical contact list, can be further organized, by adding another layer of hierarchy, to enable faster traversal. For example, an A-to-Z letter list can be visited first (instead of moving to the full contact list). The selection of a letter will then only list contacts whose names start with the selected letter.

III. 'FUNCTIONAL' STATE MACHINES

Gesture recognition is essentially a process of identifying patterns in noisy, sequential data. Firstly, sensor data is inherently noisy and secondly limb movements vary slightly each time a gesture is performed. Finite State Machines are computationally efficient and thus are suitable for use in resource constrained devices. However FSM are generally used to detect deterministic patterns as opposed to stochastic ones. Even so FSM have been applied towards gesture recognition. Hong, et al for example constructed gesture recognition FSM mainly through machine learning [4]. The FSM approach presented here also uses machine learning but as a complement to a technique for modeling FSM in a purely functional form.

This novel approach is termed a FnSM[1] or a functional state machine where the states of the Finite State Machine are function calls and event handling is mainly done with pattern matching. In order to illustrate the idea of building an FnSM, let's consider a toy example. Suppose we want to recognize a pattern where exactly 3 'a's are followed by 1 'b'. The graphical depiction of the corresponding FSM is given in Figure 4.

The graphical FSM above can be re-expressed in an appropriate functional programming language. The equivalent FnSM in F# [5] (or ML / OCaml) code is given in Listing 1.

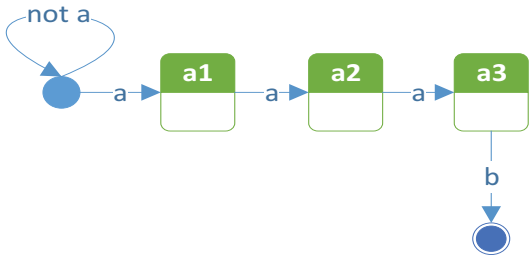

Figure 4: Example FSM for recognizing the "aaab" pattern

FnSM logic consists of pure, side-effect free functions and thus is easier to understand. While not critical to FnSM definition, strong support for pattern matching [6] greatly simplifies the FnSM logic. At their core, functional languages are inspired by mathematics and the roots of such languages can be traced to Lambda Calculus developed by Alonzo Church [7]. While equivalent logic can be expressed in other - notably Object Oriented languages, the resulting code will be more complex, likely will not be side-effect free, and can obscure the FSM logic behind Object Oriented concepts such as classes and interfaces.

Listing 1: F# code for FnSM to recognize the "aaab" pattern

```
type M<'s> = M of ('s -> M<'s>)

let rec start = function
| 'a'  -> M a1
| _    -> M start

and a1 = function
| 'a' -> M a2

and a2 = function
| 'a' -> M a3

and a3 = function
| 'b' -> M ``end``

and ``end`` _ = M ``end``
```

M is a recursively defined type that is required to package returned functions

If input matches 'a' in start state, return the a1 function (wrapped in M type)

The main expressive power comes from "partial application" activity in functional programming [8] which allows for the conveyance of more state information in function calls. If a function takes 2 arguments it can be called with the first argument only. The result is a new function that

takes the 2nd argument – this is partial application. For the FnSM concept to work the last argument of each function must be of the event type which is used in pattern matching. Thus, a function that takes more than one argument acts as template for states – rather than as a single state. Such a function can be transitioned to with the all but the last argument bound to some value. The bound values can be involved in pattern matching or other calculations when the partially applied function is called with the next event.

Partial application allows for the state-space to be organized into "semantic function-states" that group similar states together. This grouping allows one to better control any state-space explosion. Further, the partially applied arguments can be themselves FnSMs. This supports a hierarchical organization and better modularization of the state-space. This is useful for recognizing complex patterns.

By recursively transitioning to the same semantic function-state (until some exit condition is met) data from multiple events can be combined or aggregated. For example, in the case of gesture recognition, a semantic function-state can be used to track average linear acceleration over a number of events before transitioning to a new function-state when the average exceeds some threshold. Or, to go back to the start state if does not exceed the threshold. See Listing 2 for an illustration. The aggregation may be across data from different sensors – which in effect is sensor fusion.

Listing 2: A partial FnSM for the Swipe gesture

```
let rec start = function
| {Snsr=LinearAcceleration; Z=z} when
    abs z > MIN_Z_THRESHOLD
    → side_to_side z 1 |> M
| _ → start             |> M

and side_to_side prev_z count = function
| {Snsr=LinearAcceleration; Z=z} when
     count < COUNT_LIMIT
     →
        let curr_z = updateAvg (prev_z, count, z)
        side_to_side curr_z (count + 1) |> M

| …
//COUNT_LIMIT exceeded,make the transition decision…
```

Compute avg. Z-axis linear acceleration over some count number of

The overall structure of an FnSM – the semantic function-states; pattern matching; event processing; en route computation and sensor fusion etc. – is determined by a human expert. However, within the human designed scaffolding there is much room available for machine driven learning and optimization.

Consider the partial FnSM in Listing 2. Embedded in the event processing logic are 'constant' parameters such as MIN_Z_THRESHOLD, COUNT_LIMIT, etc. The runtime values of such parameters may be determined by the human expert as best guess estimates, or they may be learned from training data. A wide variety of algorithms and methods can be applied towards learning parameter values – e.g. Genetic Algorithms, Particle Swarm Optimization, etc.

Here, an implementation of Cultural Algorithms was used. Firstly, CA are shown to work well with data intensive search over large search spaces [9]. The presented gesture recognition problem involves learning optimal values of about 40 parameters, a mix of reals and integers. Secondly, the training process actually requires 'running' the FnSM which is easily done if the learning algorithm is implemented in the same language as the FnSM. The implementation is also expected to be used in future CA research. Details of the implementation are given in section VI, after a brief overview of Cultural Algorithms in section V.

IV. Gesture Recognition Review

Gesture recognition is not a new area of research so there is plenty of literature already available. This section compares and contrasts the proposed approach with the most commonly used ones, especially with respect to use in constrained devices.

Gesture recognition systems can be classified as vision-based, motion-based or touch-based. This taxonomy is reflected in the cited research. While the research problem here is motion-based, much of the underlying theory of gesture recognition is common across all approaches.

Chahal et al [4] describe the process of gesture recognition as having four stages: Data Acquisition, Gesture Modeling, Feature Extraction and Recognition. An analogous process was followed to create the prototype gesture recognition system presented here:

1. Data Acquisition: Raw sensor output was recorded while performing individual gestures a number of times.
2. Gesture Modeling: The recorded sensor output was graphically analyzed to identify the high-level patterns in sensor outputs.
3. Feature Extraction: The sensor output patterns were studied to extract FSM states and rough transition conditions.
4. Recognition: To determine precise transition conditions, the extracted FSM are 'trained' with the help of the training data and Cultural Algorithm optimization. More details are provided later.

A cursory look at gesture recognition literature will show that the theoretical foundations of dynamic [2] gesture recognition rest mainly with the following approaches:

a) Finite State Machines [4].
b) Probabilistic Graphical Models - Markov Chains / Hidden Markov Models (HMM) [10].
c) Neural Networks (NN) [11] [12].

[2] In the literature, gestures are classified as static (also sometimes called postures) or dynamic; here the word 'gesture' refers to only dynamic gestures unless otherwise stated.

d) Dynamic Time Warping (DTW) [13].
e) Support Vector Machines (SVM) [14].

The above is by no means an exhaustive list but depicts the methods most commonly applied towards dynamic gesture recognition or similar problems. Zhou, et. al. list a number of other techniques as well [15].

Of those listed above, HMM is one of the most commonly used techniques for dynamic gesture recognition, and will be used as the representative technique when evaluating and contrasting the presented approach for the following reasons.

To begin, it is well-known that the running time of an FSM is linear in the size of the input where competing approaches – namely HMM, DTW, NN and SVM – are all quadratic or higher[3]in terms of execution time. Viterbi (HMM) and DTW are $O(n^2)$ dynamic programming algorithms. NN have $O(n^2)$ worst case complexity for dense nets. From the perspective of an app (written in a high level language) running on a constrained device (e.g. a smartwatch), an FSM offers a clear advantage.

While FSM are computationally efficient they cannot easily handle stochastic patterns. A possible approach is to train an FSM to learn to recognize stochastic patterns. As such, Hong, et al [4] first used K-means clustering – without the temporal component – to identify states for each gesture and then manually sequence the states to obtain the FSM for each gesture. The time spent in each state is then learned from training data. Hong et. al. also note the computational efficiency of FSM over HMM [4].

However Hong, et. al. only use output from a single sensor (Kinect) and then only the x, y coordinates. It is not clear how well this approach will work when used with a larger state vector containing additional dimensions and sensors outputs. Are the discovered states truly meaningful for gesture recognition? More research is required to answer this question.

The raw input data from sensors is usually preprocessed and then used for gesture recognition. The preprocessing may be relatively expensive such as the application of a filter (e.g. Kalman, Particle, etc.) or feature extraction (e.g. Fourier Transform). Aditya, et al [16] combined Kalman filter for hand tracking with HMM to improve gesture recognition. Wu, et al [14] used Fourier Transform for feature extraction with SVM for gesture recognition. Akl and Valaee [17] used temporal compression to remove noise with DTW based gesture recognition. In the case of FnSM, by contrast, any required processing can be performed when needed, on a state-by-state basis. For example, in Listing 2, the average linear acceleration is computed only when the FnSM is in a particular state. On constrained devices this is a significant saving. In addition, FnSM do not preclude preprocessing, if that is the best option.

Discrete HMMs require quantization of real valued sensor data. Also, the complexity of an HMM grows exponentially with the number of variables. The transition probability table of a three-variable HMM is exponentially larger than that of a two-variable one. The current system uses 3 sensors - linear acceleration, gravity and gyroscope - each with x, y and z axis values – for a total of 9 variables. Even with coarse quantization the transition probability table (one for each gesture) will end up being very large. In such a scenario an HMM would require feature extraction in order to reduce the state space [18]. An FnSM is not affected in this way – at least as used here. Firstly, FnSM do not use transition probability tables but rather conditional logic that is individually tailored to each semantic function-state and event. Secondly, events from multiple sensors can be handled as independent events – these do not have be fused into a single state label[4]. For example, the Android serializes sensor events in a thread-safe manner. FnSM handle each event individually depending on the state. Pattern matching is used to discriminate between events from different sensors. If the FnSM is tracking linear acceleration for the swipe gesture, it can handle gyroscope events in parallel. If it so happens that the wrist rotation rate is higher than a given threshold, the FnSM can reset itself as there should be little to no wrist rotation in swipe.

In this prototype an Android-based watch is used. An Android device continuously generates sensor events. Recognition of individual gestures in such a data stream is a challenge. Segmentation [18] or framing [14] (sliding windows) is required to carve a chunk of feature vectors that can be used as input to HMM, DTW, NN or SVM. Note that the use of sliding windows will increase the computation burden considerably as the same data is processed multiple times. An FnSM on the other hand can operate in a streaming fashion. The incoming sensor events are piped to multiple FnSM (one for each gesture) running in parallel. Each FnSM can reset itself as soon as the input sequence is found to be incompatible with the gesture being tracked. Whenever a gesture is recognized, all FnSM are reset and gesture recognition resumes after a very short pause.

V. CULTURAL ALGORITHMS REVIEW

This section provides a brief overview of Cultural Algorithms (CA) which is a class of data intensive evolutionary computing algorithms inspired by models of human cultural evolution. Unlike some other nature inspired algorithms e.g., Particle Swarm Optimization, Ant Colony Optimization, and Artificial Bee Colony etc., CA are not just population based and support socially motivated learning via a network of individuals and knowledge sources. CA were first proposed by Reynolds in 1979 [19] and over the years have been further developed and applied towards a wide variety of problems, e.g. [20] [9] [21] [22].

The CA has two main components – the Population Space and the Belief Space. The population of individuals are usually networked in a Social Fabric. The problem solving

[3] SVM complexity further depends on the kernel function used. The popular SVM library "libsvm" estimates complexity at $O(n^3)$.

[4] HMM require the fusing of sensor data to extract a single state label for each discrete time step

experience of individuals from the Population Space are ACCEPTED into the Belief Space as a variety of types of knowledge and used to UPDATE the knowledge source network there. Knowledge sources in the Belief Space can also constitute a knowledge network so that updates to one knowledge source can be transmitted to other knowledge sources. Knowledge from the Belief Space is then used to INFLUENCE future generations, as conditioned by the social structure of the population.

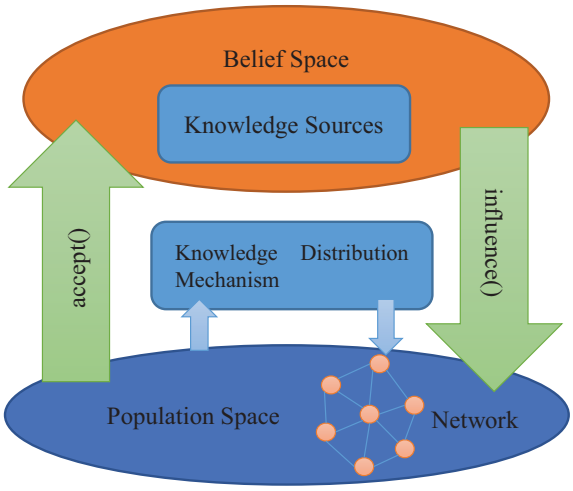

Figure 5: Cultural Algorithms Architecture

The Cultural Algorithms optimization process uses a variety of stochastic search strategies (Knowledge Sources) concurrently and can vary the strategy mix to adapt to the search landscape discovered thus far. Prior research indicates that the CA performs well on problems with large search spaces, and therefore should be well suited to the problem of optimizing FnSM parameter values.

A mathematical view of CA (using notation from OCaml / F#) is given in and the corresponding graphical view is shown in Figure 5. The notation key is provided at the end of the listing. Not all of the details can be captured in the mathematical definition so additional narrative is provide next.

A CA instance (CAinst) is a structure that contains all of the pieces required to (potentially) solve an optimization problem. The data in the CAinst is used in an iterative process to find the optimum solution (or until some termination condition is met). The Population is a collection of Individuals networked into a Social Fabric. An Individual's parameter values represent a point in the problem hyperspace. The fitness is the value of this point wrt the Fitness function. The KS value of the Individual is the type of Knowledge (indirectly the KnowledgeSource) under whose influence the Individual is currently in. During initialization, a Knowledge type is randomly assigned to the Individual. At the beginning of each time step, the population fitness is re-evaluated and stored for further processing. At each time step, the best performing Individuals are extracted from the Population space using the AcceptanceFunction.

Listing 3: Cultural Algorithm Definition in F# / OCaml Notation

```
type CA =
    {
        Population              : Population
        Network                 : Network
        KnowlegeDistribution    : KnowledgeDistribution
        BeliefSpace             : BeliefSpace
        AcceptanceFunction      : Acceptance
        InfluenceFunction       : Influence
        UpdateFunction          : Update
        Fitness                 : Fitness
    }

where:
Topology  = the network topology type of the population e.g. square, ring, global,
etc.
Knowledge   = Situational | Historical | Normative | Topgraphical | Domain
Individual  = {Id:Id; Parms:Parm array; Fitness:float; KS:Knowledge}
Id          = int
Parm        = numeric value such as int, float, long, etc.
Population  = Individual array
Network     = Population → Id → Individual array
Fitness     = Parm array → float
BeliefSpace = KnowledgeSource Tree
Acceptance  = BeliefSpace → Population → Individual array
Influence   = BeliefSpace → Population → Population
Update      = BeliefSpace → Individual array → BeliefSpace
KnowledgeDistribution = Population → Network → Population

KnowledgeSource =
    {
        Type        : Knowledge
        Accept      : Individual array → Individual array * KnowledgeSource
        Influence   : Individual → Individual
    }

Notation key:
type   = basic type e.g. int, string; a record; a union or cross product of types
record = { [field name : type]+ }, is a tuple with named fields
type * type = a cross product of two types e.g. int*int
type | type = union of two types (can be a value of either type)
type → type = function where LHS is the input and RHS is the output
type → type → type = a function that can be partially appied
type array = a 1-dimensional array of the corresponding type
```

These Individuals are then used by the UpdateFunction to update the BeliefSpace. Here knowledge from the best performing Individuals is extracted and stored in the BeliefSpace. The KnowledgeSource type provides two functions Accept and Influence. The Accept function takes an Individual and returns an updated KnowledgeSource plus a list of accepted Individuals. The BeliefSpace consists of KnowledgeSources where different types of knowledge is stored. After the BeliefSpace is updated, the stored knowledge is used to modify the population individuals via the InfluenceFunction. However, before the InfluenceFunction is applied the existing population is modified using the KnowledgeDistribution and Network functions to assign a new Knowledge type to each individual in the population. How the assignment of a Knowledge type to an individual happens depends on the implementations given for the two functions. The Influence function accepts an Individual and returns an updated Individual. Here the Individual's parameter values are modified according to the type of KnowledgeSource the individual gets associated with. The CA framework defines different types of KnowledgeSource such as Situational, History, Domain, Topographical and Normative. Figure 9 is a sample run that shows the change in KS distribution over time. for The reader is referred to other CA papers for the detailed explanation of the various KnowledgeSource types [19] [20] [9].

VI. GESTURE RECOGNITION: IMPLEMENTATION, OPTIMIZATION AND RESULTS

A gesture recognition app based on the FnSM concept was implemented in order to recognize the gesture vocabulary presented in section II. The implementation was done on a *Sony Smartwatch 3* Android smartwatch (Figure 6) using the F# language and the Xamarin toolset for mobile app development. The smartwatch app relays the detected gestures to the companion app on the Android smartphone. The recognized gesture events are then used to drive the mock HMI menu structure presented in section II.C.

The overall FnSM organization is a hierarchical one (Figure 7). The Twist gesture is the trigger to start recognizing the other gestures (and stop recognizing the Twist gesture). The FnSM for the other gestures run concurrently. All sensor events are sent to all of the concurrently running FnSM. The first gesture that is recognized by any of the FnSM is processed, and then all of the FnSM are reset to the start state. If the Escape gesture is recognized, then the high level FnSM switches back to recognizing the Twist gesture (and stops recognizing other gestures).

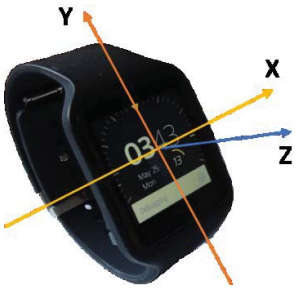

Figure 6: Sony Smartwatch 3 axis orientation

One advantage of using a hierarchical approach is reduced power consumption. In order to recognize the Twist gesture, the FnSM only requires the Gyroscope sensor and therefore the app can turn-off sensor input for all other sensors, such as linear acceleration and gravity. The official documentation on handling Android sensor events suggests that the application should only register for sensors that it needs. Engaging sensor hardware increases power consumption significantly [23]. After the Twist gesture is recognized, the app registers for additional sensors required for the recognition of other gestures; and it unregisters after the Escape gesture is recognized. Note that for a typical drive the Twist gesture recognition will be 'on' for almost the entire time, waiting for the driver's signal to switch to the menu-traversal gestures.

Further power consumption savings are possible by reducing the sampling rate. The Android API allows an app to set the time between successive sensor events. A coarser sampling rate saves power by reducing computation but at the cost of accuracy. We used the standard "GAME" rate of approximately 20ms between events for all sensor input.

The hierarchical structure of the system allows the system to start with a "coarse grained" resolution model, and switch to higher resolution models when needed for other gestures.

The FnSM for each gesture was constructed by first examining the raw sensor data plots (see Figure 8 for an example). This examination revealed the basic structure of the FnSM such as the function-states, transition conditions, extra computation requirements, and parameter values (e.g. limits, thresholds, ranges, etc.).

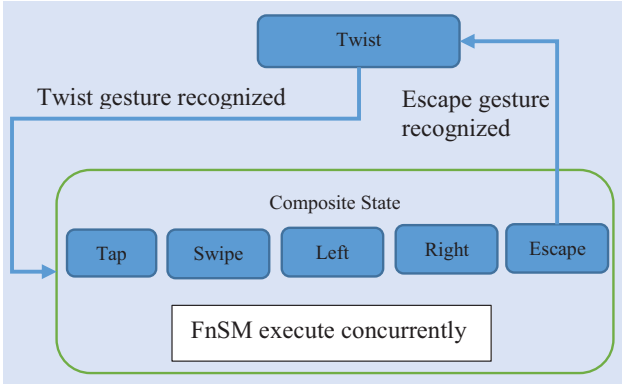

Figure 7: Hierarchical organization of the gesture recognition FSM

The FnSM were then constructed in F# code. The Best guess values for the various parameters (limits, thresholds and ranges) where used as the initial set.

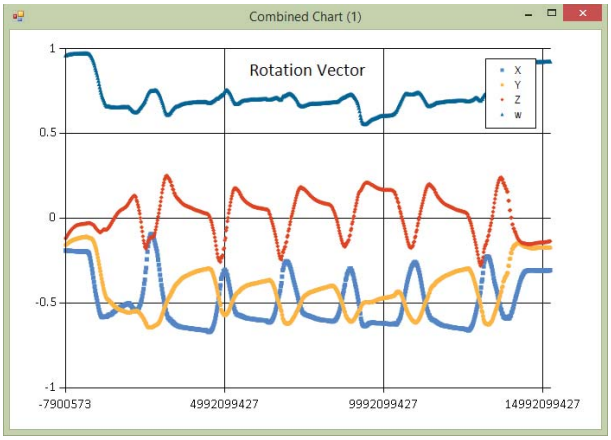

Figure 8: Rotation Vector sensor data plot for repeated Twist gestures

Sensor data was recorded for 10 repetitions of each gesture of the composite state machine, namely Tap, Swipe, Left and Right. These 4 data sets became the positive training examples. Another data set called "Driving" was created by recording sensor data from an emulated drive session. This was used as a negative training set.

The Fitness function – a crucial component for the training/optimization is now described. For each Individual in the population, the CA Fitness function first converts an Individual's Parm array values to a configuration structure. This structure parameterizes the gesture recognition FnSM. The Fitness function then 'runs' the parameterized FnSM over the corresponding training data sets and calculates a fitness score for the Individual. As there are 5 datasets (4 positive and 1 negative) the Fitness function runs the FnSM 5 times, once for each data set. The CA run finds the parameter set to optimize the value of the F(p):

$$\arg max_p F(p) = \text{W1} * \left[\sum_g fit(p,g) - \text{W2} * countD(p)\right] - DF(p)$$

Where:

p = parm array values from a population individual

g ∈ G ={Left, Right, Tap, Swipe}, the action gestures

fit(p, g) = GC(g) – | count(p, g) – countNot (p, g) |
is the fitness function for a single gesture. This function's value is calculated by running the FnSM parameterized with p over the training set for g. (The Ideal score is 0 when all of the 10 gesture instances are recognized and none any other gestures are recognized).

GC(g) = The # of gestures of type g in the training set for g (constant 10 for all gestures).

count(p, g) = The count of gesture of type g recognized when the p parameterized FnSM is run over the training set for g. (The ideal value is 10).

countNot(p,g) = The count of all gestures of type [G – {g}] recognized when the p parameterized FnSM is run over the training set for g. (The ideal value is zero).

countD(p) = The total count of any gesture in G recognized in the negative (driving) training set with a p parameterized FnSM. (The ideal score is zero). This term is used to reduce inadvertent recognition of any gestures.

DF(p) = $\sum_{i,j \in G, i \neq j} |fit(p,i) - fit(p,j)|$
The sum of absolute differences between all individual fitness scores. This term favors p values that result in even gesture recognition performance across all gesture types in G. As an example, for a hypothetical two-gesture system, it is better to have a recognition score of (5, 5) than (10, 0) – assuming 10 iterations of each gesture in the training set.

W1 and **W2** are weighting factors to control the influence of some of the terms.

In summary, the fitness function is constructed so that a perfect score of 120 is achieved when all 10 instances of a gesture type are recognized with the corresponding training set and none of any other type are recognized (for each of the action gesture types). See Figure 9 for a sample CA run (0 is the generation after the first CA step).

The fitness score of the best guess parameter values was calculated and then applied to the CA optimization process. The results are shown below:

Step (before / after CA)	*Fitness Score (max 120)*
Best guess parameter values	-1.5
After CA optimization	92.5

The results show that the CA achieved a dramatic incremental improvements in gesture recognition performance by selectively applying machine optimization. Notice that initially exploratory knowledge source are employed (normative and domain) to make a initial increment in performance. After that normative knowledge continues to drive the search while domain knowledge becomes less important. A second bridging action takes place at step 3. After that normative knowledge begins to drop out as well and the remainder of the search process falls on the shoulders of history and situational knowledge. These knowledge sources are exploitative knowledge sources that conduct a fine turning operation. Such incremental learning is characteristic of complex physical systems where certain functional building blocks must be produced before other innovations can be added. Previous work by Kinniard-Heether and Reynolds [24] on training driver controllers.

In a complex system, the search process can generate emergent building blocks through search guided by domain knowledge. Once a building block is available it can then be exploited by other knowledge sources until they become less productive and control may return to more exploratory knowledge sources. Thus, one sees "knowledge swarms that are attracted to different features during the search process. Here exploratory knowledge sources take charge at first, and their influence gradually is diminished as the exploitative knowledge sources take over.

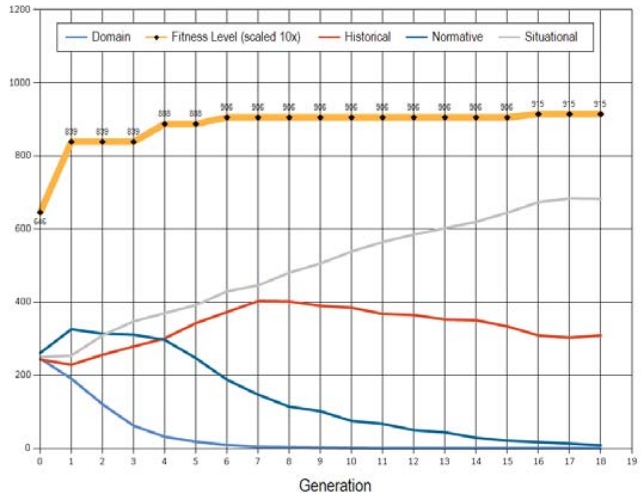

Figure 9: Sample optimization run showing best fitness and change in KS distribution over generations (pop. size = 1000)

VII. CONCLUSIONS

In the context of the three emerging trends of a) mainstream wearable computing adoption; b) the increasing use of gestures based automotive HMI controls; and c) new APIs for automotive-smartphone integration the goal was to answer the question of whether it's viable use a wearable device (specifically smartwatch) for gesture control of automotive HMI.

There were a few challenges to consider a) limited computational power and battery life of wearable devices; b) limited space available to the driver (especially in a left-hand-drive car wearing the smartwatch on the left wrist); and c) the large number of possible HMI menu options in today's automotive HMI systems. In order to test the viability of this approach the following steps were taken in the project:

- A simple gesture vocabulary to drive a hierarchical menu based mock HMI was devised.
- A gesture recognition system as an app that runs on a smartwatch was implemented.
- An innovative, functional programming based approach to compactly model finite state machines for recognizing complex and stochastic patterns was developed.
- The Cultural Algorithm was inserted into the simulation loop in order to optimize the human's initial best guess parameter values.

A 77% (92.5 / 120) recognition accuracy was achieved. The training data was also used for validation.

The preliminary research results show that wearable device gesture recognition is viable for automotive use but the question cannot be fully answered until significant field testing is performed, which is planned for the next phase.

In the future phases of this research the goal is to increase the gesture recognition accuracy by re-examining the FnSM structure especially in light of slack variables; use separate training and validation sets; and use training data recorded in a moving vehicle to better emulate the driver's environment.

In addition, the Cultural Algorithm is a powerful technique and can be applied towards learning the structure of the FnSM – not just optimizing the tunable parameters (which was done for in this research effort). Future plans are to investigate the use of CA to generate better FnSM structures given the training data and the optimization formulation.

ACKNOWLEDGEMENTS

Dr. Reynolds wishes to acknowledge the support of the National Science Foundation through Grant #1451316.

REFERENCES

[1] IDC, "Worldwide Wearable Computing Device Market 2014-2018 Forecast and Analysis," IDC, Framingham, 2014.

[2] IDC, "IDC Worldwide Consumer Technology Predicitons," IDC, Framingham, MA, 2014.

[3] J D Power, "Voice Recognition No.1 Problem With New Vehicles," [Online]. Available: http://www.jdpower.com/press-releases/2014-multimedia-quality-and-satisfaction-study. [Accessed 27 August 2015].

[4] P. Hong, T. S. Huang and M. Turk, "Gesture Modeling and Recognition Using Finite State Machines," in *IEEE International Conference on Automatic Face and Gesture Recognition*, Grenoble, 2000.

[5] Microsoft Corporation, "Visual F#," [Online]. Available: https://msdn.microsoft.com/en-us/library/dd233154.aspx. [Accessed 2015].

[6] Microsoft, "Pattern Matching (F#)," 2013. [Online]. Available: https://msdn.microsoft.com/en-us/library/dd547125.aspx.

[7] P. Andrews, "Church's Type Theory," Spring 2012. [Online]. Available: http://plato.stanford.edu/archives/spr2014/entries/type-theory-church.

[8] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg and B. Yorgey, "Software Foundations (online book)," University of Pennsylvania, Philadelphia, 2015.

[9] N. Rychtyckyj and R. Reynolds, "Using Cultural Algorithms to Re-Engineer Large-Scale Semantic Networks," *International Journal of Software Engineering and Knowledge Engineering,* vol. 15, no. 4, pp. 665-693, 2005.

[10] Y. Gu, H. D, Y. Ou and W. Sheng, "Human Gesture Recognition through a Kinect Sensor," in *Proceedings of the 2012 IEEE International Conference on Robotics and Biomimetics* , Guangzhou, 2012.

[11] M. C. Mozer, "Neural net architectures for temporal sequence processing," University of Colorado , Boulder, 1997.

[12] I. Mocanu and T. Cristea, "HAND GESTURES RECOGNITION USING TIME DELAY NETWORKS," *Journal of Information Systems & Operations Management,* no. Winter 2013, pp. 1-9, 2013.

[13] Y.-L. Hsu, C.-L. Chu, Y.-J. Tsai and J.-S. Wang, "An Inertial Pen With Dynamic Time Warping Recognizer for Handwriting and Gesture Recognition," *Sensors Journal, IEEE,* vol. 15, no. 1, pp. 154 - 163, 2014.

[14] J. Wu1, G. Pan, D. Zhang, G. Qi and S. Li1, "Gesture Recognition with a 3-D Accelerometer," in *Ubiquitous Intelligence and Computing*, Springer Berlin Heidelberg, 2009, pp. 25 - 38.

[15] S. Zhou, F. Fei, G. Zhang, J. Mai, Y. Liu, J. Liou and W. Li, "2D Human Gesture Tracking and Recognition by the Fusion of MEMS Inertial and Vision Sensors," *Sensors Journal, IEEE ,* vol. 14, no. 4, pp. 1160-1170, 2014.

[16] A. Ramamoorthya, N. Vaswania, S. Chaudhurya and S. Banerjeeb, "Recognition of dynamic hand gestures," *Pattern Recognition,* vol. 36, p. 2069 – 2081, 2003.

[17] A. Akl and S. Valaee, "ACCELEROMETER-BASED GESTURE RECOGNITION VIA DYNAMIC–TIME WARPING, AFFINITY PROPAGATION, & COMPRESSIVE SENSING," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, Dallas, 2010.

[18] M. Elmezain, A. Al-Hamadi, J. Appenrodt and B. Michaelis, "A Hidden Markov Model-Based Continuous Gesture Recognition System for Hand Motion Trajectory," in *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, Magdeburg, 2008.

[19] R. Reynolds, "Cultural Algorithms: A Tutorial," [Online]. Available: http://groups.engin.umd.umich.edu/vi/w2_workshops/cultural_alg_reynolds_w2.pdf.

[20] R. G. Reynolds, M. Ali and T. Jayyousi, "Mining the Social Fabric of Archaic Urban Centers with Cultural Algorithms," *IEEE Computer,* vol. 41, no. 1, pp. 64-72, 2008.

[21] M. Ali, P. Suganthan, R. Reynolds and A. Al-Badarneh, "Leveraged Neighborhood-Restructuring in Cultural Algorithms for Solving Real-World Numerical Optimization Problems," *IEEE transactions on evolutionary computation,* 2015.

[22] Z. Kobti, A. Snowdon, S. Rahaman, T. Dunlop and R. Kent, "A Cultural Algorithm to Guide Driver Learning in Applying Child Vehicle Safety Restraint," *IEEE Congress on Evolutionary Computation,* pp. 1111-1118, 2006.

[23] Google, "Sensors Overview," Google, [Online]. Available: http://developer.android.com/guide/topics/sensors/sensors_overview.html. [Accessed 26 4 2015].

[24] R. G. Reynolds and L. Kinniard-Heether, "Networks Do Matter: The Socially Motivated Design of a 3D Race Controller Using Cultural Algorithms," *International Journal of Swarm Intelligence Research,* vol. 1, no. 1, pp. 17-41, 2010.