Evaluation of RSL History as a Tool for Assistance in the Development and Evaluation of Computer Vision Algorithms

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Matthew Fluet, Supervisor

Dr. Richard Zanibbi, Supervisor

Dr. Hans-Peter Bischof, Observer

Evaluation of RSL History as a Tool for Assistance in the Development and Evaluation of Computer Vision Algorithms

by

Ben Holm, B.S.

THESIS

Presented to the Faculty of the Galisano College of Computer and Information Sciences
Rochester Institute of Technology
in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

Rochester Institute of Technology

August 2011

Acknowledgments

I would like to thank the many people who helped me during this project. In particular I wish to thank Dr. Matthew Fluet and Dr. Richard Zanibbi for their guidance, correction, and patience throughout. Sudeep Sarkar and Ruiduo Yang responded quickly and kindly to many questions, and I would not have completed this work without them. Ron Male agreed to proofread this paper, and any correct punctuation or grammar can be attributed to his patience and careful eyes.

Finally, I thank my family. My children, Adam, Matthew, and Anna, for doing without a playroom and living with a grumpy, busy dad. And Mary, my wife, for her support and encouragement throughout this project and the coursework that came before. Without her I would never have finished.

Abstract

A revision of Recognition Strategy Language (RSL), a domain-specific language for pattern recognition algorithm development, is in development. This language provides several tools for pattern recognition algorithm implementation and analysis, including composition of operations and a detailed history of those operations and their results. This research focuses on that history and shows that for some problems it provides an improvement over traditional methods of gathering information.

When designing a pattern recognition algorithm, bookkeeping code in the form of copious logging and tracing code must be written and analyzed in order to test the effectiveness of procedures and parameters. The amount of data grows when dealing with video streams; new organization and searching tools need to be designed in order to manage the large volume of data. General purpose languages have techniques like Aspect Oriented Programming intended to address this problem, but a general approach is limited because it does not provide tools that are useful to only one problem domain. By incorporating support for this bookkeeping work directly into the language, RSL provides an improvement over the general approach in both development time and ability to evaluate the algorithm being designed for some problems.

The utility of RSL is tested by evaluating the implementation process of a computer vision algorithm for recognizing American Sign Language (ASL). RSL history is examined in terms of its use in the development and evaluation stages of the algorithm, and the usefulness of the history is stated based on the benefit seen at each stage. RSL is found to be valuable for a portion of the algorithm involving distinct steps that provide opportunity for comparison. RSL was less beneficial for the dynamic programming portion of the algorithm. Compromises were made for performance reasons while implementing the dynamic programming solution and the inspection at every step of what amounts to a brute-force search was less informative. We suggest that this investigation could be continued by testing with a larger data set and by comparing this ASL recognition algorithm with another.

Table of Contents

Acknowledgments **iii**

Abstract **iv**

List of Tables **viii**

List of Figures **ix**

Chapter 1. Introduction **1**
- 1.1 Thesis objective . . . 1
- 1.2 Problem statement . . . 1
- 1.3 Solution approach . . . 3
  - 1.3.1 Development . . . 5
  - 1.3.2 Evaluation . . . 6
- 1.4 Contributions . . . 8

Chapter 2. Background **10**
- 2.1 RSL . . . 10
- 2.2 Program tracing and logging . . . 14
- 2.3 Domain-specific languages . . . 16
- 2.4 Recognition of continuous American sign language . . . 19
  - 2.4.1 Hand detection . . . 20
  - 2.4.2 Sign-matching dynamic programming . . . 22
  - 2.4.3 Level building dynamic programming . . . 24
- 2.5 Summary . . . 27

Chapter 3. Evolving RSL **28**

3.1 Syntax and semantics 28
3.2 RSL Annotations 30
3.3 RSL Style and Organization 33
3.4 Extension 37
3.5 CVSL 37

Chapter 4. Evaluation of RSL History **41**

4.1 Algorithm Development Stage 41
4.1.1 Representing the Hand Detection algorithm as RSL 41
4.1.2 Representing the Dynamic Program as RSL 42
4.1.3 Debugging RSL Programs 47
4.1.4 Results of development stage 48
4.2 Algorithm evaluation stage 50
4.2.1 Algorithm evaluation criteria 50
4.2.2 Evaluating hand detection 51
4.2.3 Results of evaluating hand detection 60
4.2.4 Evaluating the dynamic program 63
4.2.5 Results of evaluating the dynamic program 74
4.3 Algorithm implementation difficulties 78

Chapter 5. Conclusion **81**

5.1 Summary of RSL evaluation 81
5.1.1 Contributions 83
5.2 Future work 84
5.2.1 Future RSL experiments 84
5.2.2 Possible changes to RSL 84
5.2.3 Expansion of the ASL recognition algorithm implementation 85
5.3 Summary 86

Appendices **88**

Appendix A. RSL Implementation Languages **89**
A.1 Use of Standard ML 89
A.2 Calling C from SML 93

Appendix B. RSL source listings **97**

List of Tables

2.1 DSL Evaluation Criteria 18

3.1 Changes to RSL 29

4.1 Sentences tested 64
4.2 Level building results 68

5.1 Criteria to test the hypothesis in the development stage 82
5.2 Criteria to test the hypothesis in the evaluation stage 82
5.3 Hypothesis evaluation criteria for each algorithm stage. Only *Creation* of history for the Dynamic Programming was not easier. . . . 86

List of Figures

2.1 Hand candidate boundary detection 22

2.2 Mahalanobis distances for frames 24

2.3 Best signs 25

3.1 Layers of an RSL program 33

4.1 Maximum number of hand candidates across all frames after each algorithm step for the sign *why* 55

4.2 Euclidean distances for the hand candidate with the lowest Mahalanobis distance in every frame of the sign *why* 56

4.3 Hand candidate quality for values of T_1 for all frames of the sign *why* 59

4.4 Probability mask function of match vs. Non match for all trained signs 67

4.5 Average edit errors for all interpretations and the percent of interpretations that represent ground truth across all levels for sentence ”i need that i”. Edit error is the percent of the frame labeling that needs to change to make the interpretation match ground truth. Levels with no bar for interpretations that represent ground truth have no correct interpretations. 75

4.6 Average edit errors for correct interpretations versus incorrect. Edit error is the percent of the frame labeling that needs to change to make the interpretation match ground truth. 76

Chapter 1

Introduction

1.1 Thesis objective

Hypothesis Recognition Strategy Language (RSL) history will provide information useful for evaluating decisions that would be more difficult to obtain using traditional logging methods.

1.2 Problem statement

In ”White-Box Evaluation of Computer Vision Algorithms through Explicit Decision-Making”, Zanibbi et al. [27] show that a decision history from table recognition algorithms can be used to evaluate and improve those computer vision algorithms. The history was created using a domain specific language (DSL) called RSL to script the table recognition algorithms and grab history data at each decision point. This research examines how historical information gathered from an updated RSL can be used for another computer vision algorithm [1].

Historical information from a computer vision algorithm decision point may require large datasets of images, videos, keyframes, object boundaries, etc. Each interpretation of an image or series of frames often requires its own version of all this data. After running the algorithm, examining this historical data for accuracy,

precision, historical accuracy, and historical precision [26] may reveal useful information about the algorithm that was run. It is important that the history be captured, but managing the history (its creation, storage, and exploration) is not the primary concern of the computer vision algorithm developer. In addition to being a distraction, this logging code can be complex and repetitive and have bugs of its own. Complex data structures need complex storage and access. It would be better to let this bookkeeping work be handled automatically by a tool set. To the best of our knowledge, there are presently no tools for recording history to examine fitness or usefulness of particular steps. The closest work is related to logging and tracing of code for debugging purposes.

Debugging tools and approaches are an active area of research, and many ideas can be found [20, 14, 6, 15, 5]. Unfortunately, when using the tools, much of the work still remains with the programmer to manage logging information, and that management is time consuming. A common problem is that the general-purpose languages require general-purpose logging, so any work specific to decisions, images, video, or other domain-specific concepts must be added by the developer. In addition, tools commonly used in the computer vision field have not embraced the logging technologies and instead rely on developers to implement their own approach. This approach is generally the ad hoc tracing or breakpoint-based examination of state, as seen in the original implementation of the American Sign Language algorithm examined in this research [22]. Here, nested loops have special cases as seen in Listing 1.1 in order to mark a point of interest for examination, likely using debugger breakpoints.

Listing 1.1: Special case code to examine state

```
if (v==5 && i==7 && l==1) {
    // ppp is a write-only variable (never used)
    // it enables setting a breakpoint to examine
    // algorithm state
    int ppp=0;
}
```

By providing history specific operations, a DSL may have an advantage over general-purpose languages and logging. This research examines RSL to test if RSL's history is beneficial in evaluating computer vision algorithms.

1.3 Solution approach

RSL provides several tools for pattern recognition algorithm implementation and analysis:

- Composition of operations on interpretations.
- History of those operations - what operations were applied and what results were returned.
- Annotation of decisions - Information not included in the result of a decision is kept and examined at a later date.

A computer vision algorithm is implemented using RSL to drive the algorithm decisions. Decision history is captured and used to debug and evaluate the algorithm. The evaluation of the RSL language along with generation and usefulness of the history is based on guidelines provided by Mernik et al. [17], Bentley

[2], Hudak [12], and Elliot [7]. This research focuses on criteria laid out by Mernik et al. [17]:

- Data structure representation - History information tends to be full of special cases and conditionals in traditional logging based implementations. RSL should avoid this problem.
- Domain-specific constructs - Special attention is paid to the automatic traversal of interpretations and history.

Data structure representation and domain-specific constructs are examined to understand how history is created, traversed, communicated, stored, and examined. For some problem types, RSL support for these operations provides significant improvement over the existing logging or tracing methods by doing all the common domain-specific work.

RSL history is considered across the operations of two stages of computer vision strategy implementation:

- *Development* - the implementation of the algorithm, scripted from RSL, instrumented for recording the history
- *Evaluation* - examination of the history across the data sets and steps to develop insight into the algorithm and its implementation

Use of history is judged in comparison to general-purpose logging currently used in computer vision algorithm development. Comparisons are made by writing general-purpose logging code to get similar or identical information.

1.3.1 Development

Development and debugging are inseparable steps and are considered together here. During algorithm development the RSL operations provided for history are *creation* (how history is generated), *communication* (how history is moved from source to destination), and *storage* (how history is persisted for later examination). Since development is not central to the use of the history, supporting this thesis hypothesis only requires that developing with history is roughly equivalent to traditional logging methods. Refuting the hypothesis requires that development with history is prohibitively difficult since a slight increase in difficulty could be worth extra effort if the evaluation stage shows improvement over general-purpose logging methods. For example, the historical precision and historical accuracy data that RSL was designed to collect is gathered in the evaluation phase [26].

History is	*Creation*	*Communication*	*Storage*
Easier	Support	Support	Support
Same	Support	Support	Support
Harder			
Much Harder than logging	Refute	Refute	Refute

There is no quantitative measurement of *easier* or *harder* for a programming language, but there are guidelines available. In ”When and How to Develop Domain-Specific Languages”, Mernik et al. [17] extract from their survey a set of reasons to create domain-specific languages. Those reasons for creating languages become the goals of those languages and the criteria by which they are judged. For

the development stage I considered these goals [17]:

- ”substantial gains in expressiveness and ease of use” - RSL development should substantially improve how the creation and examination of history is expressed over general purpose logging tools.

- ”task automation” - RSL should eliminate the tedious, repetitive work of creating and examining history, leaving only the problem specific work.

- complex ”data structure representation” - History information may be complex, and dealing with this complexity should be handled by RSL. The algorithm developer should have to do little work to generate and navigate history other than the work specific to the problem.

1.3.2 Evaluation

Evaluation is the stage where RSL history should show some benefit, and the operations for *examination* (how the history is viewed) and *traversal* (how the history is searched or explored) are key to this stage. The evaluation stage must see improvement over traditional logging to support fully the hypothesis. If examining the history is no easier, the thesis hypothesis is not refuted. Much of the examination operation may be specific to a particular problem: for example, interpreting a hand in an image as a sign. However, the traversal operation must improve in order to support the hypothesis that RSL history is valuable to algorithm evaluation. Traversal is important because computer vision algorithms may be considering several possible interpretations for a given video or image. The ability to record the

decisions for each interpretation is a key requirement of the language. The traversal concept can also be applied to a range of values such as thresholds. Computer vision algorithms will sometimes search over thresholds or parameters to find the best solution, and RSL aims to simplify that work.

History is	*Examination*	*Traversal*
Easier	Support	Support
Same		Refute
Harder than logging	Refute	Refute

The evaluation stage is considered using criteria from Mernik et al. [17] as well:

- complex ”data structure representation” - Evaluation will require access to the history generated during development. The initialization of this history should be easier than with general logging tools.
- ”data structure traversal” - Accessing the items of history should be done more naturally than it would be with general purpose tools.
- ”interaction” - According to Mernik et al. [17], interaction with the application should be made easier. In this case, that means interacting with history and extracting information. Questions addressed during the evaluation stage are:

- Is new information readily apparent that would not be noticed with general purpose tools?
- Is it possible to produce new types of analysis that would be too difficult or time consuming with general purpose tools?
- Are methods for visualizing the data accessible and informative?

1.4 Contributions

This research considers two subproblems: hand-detection and nested dynamic programming of an ASL recognition algorithm. For the hand-detection algorithm, a clear advantage in using RSL is demonstrated according to our evaluation criteria. For the dynamic programming algorithm, the advantage of using RSL for history storage and traversal is shown, but the utility is not.

Additionally, this research provides insight to the designers of RSL with regard to the usability of RSL history. As noted by Mernik et al. [17], ”DSL development is not a simple sequential process”, and each stage of language development may provide new insight or questions into previous stages. By implementing computer vision algorithms in RSL, this research becomes an integral step in the domain-specific language development process [17]. Specifically, the language is changed in the following ways at least in part through the feedback from this work. The annotations language feature and supporting functions are included in RSL as a result of observations made during this work. Minor compiler changes were made to allow recursive functions. RSL extensibility is demonstrated through the creation

of a computer vision specific API. A design structure for RSL is proposed and implemented to create a clear distinction in responsibilities between RSL scripts, and the called decision functions.

Through replicating the ASL recognition algorithm, several steps not called out in the original paper are made clearer, in addition to the identification and partial correction of errors in the dataset.

Chapter 2

Background

This chapter will cover the background and related material for this research. The history and use of RSL is explained in Section 2.1, and alternatives to generating history information are discussed in Section 2.2. Criteria for evaluating domain specific languages such as RSL are discussed in Section 2.3. Section 2.4 describes the algorithm implemented as part of this research to evaluate RSL.

2.1 RSL

RSL is presented by Zanibbi, Blostein, and Cordy in ”White-Box Evaluation of Computer Vision Algorithms through Explicit Decision-Making” [27] and ”Decision-Based Specification of Table Recognition Algorithms” [26]. Both papers deal specifically with table recognition algorithms and note that the black-box nature of table recognition algorithms [27] make it difficult to compare algorithms or evaluate individual decisions [26]. RSL is presented as a tool for solving these problems by providing a mechanism to ”measure the accuracy of individual recognition decisions, and the accuracy of sequences of recognition decisions” [26]. Using these measures, Zanibbi et al. [26] make new observations about an algorithm under analysis [10] and introduce the concepts of historical recall and historical

precision. ”Recall is the percentage of ground truth hypotheses present in an interpretation, whereas precision is the percentage of accepted hypotheses that match ground truth.” [27]. The historical versions of recall and precision take into account the hypothesis created and rejected throughout the algorithm, not just the final hypothesis at algorithm completion.

RSL is a domain specific scripting language designed to provide a tool set and common approach to the task of developing algorithms for pattern recognition. It is important to remember that the domain of this scripting language is not pattern recognition but pattern recognition strategy development. Therefore, the work done by the language is meant to help inform the analysis and refinement of a pattern recognition algorithm.

An interpretation represents a single possible understanding by the algorithm of the data to be tested. RSL accomplishes its goals by organizing algorithms into traceable decision functions about interpretations. For example, in the hand detection portion of the American Sign Language recognition algorithm used in this research, an interpretation may contain the processed images for a frame. Listing 2.1 shows an RSL program fragment that finds keyframes for a series of video frames as an early first step in detecting hands. During hand detection, keyframes represent a series of frames that have little change between them. The first frame is selected as a keyframe automatically. Each successive frame is then compared to the last keyframe, and, when the difference crosses a threshold, a new keyframe is selected. These keyframes are compared to every frame again later to help determine where the motion in the frame is, since the signer’s hands are moving more

than other things in the frames. The `GetFrames` decision uses the `munge` operation to generate an initial set of interpretations, one for each frame. Subsequent decisions labeled `GetSkinMask` and `GetGrayScale` perform image processing operations on each frame to generate new images. Finally the `KeyFrames` decision determines the keyframes of the video and marks only those interpretations for the key frames.

Listing 2.1: RSL call to find keyframes

```
interp:
    frameId : int
    keyframe : bool
    frame : char vector * int * int * int
    gray : char vector * int * int * int
    skin : char vector * int * int * int

fn main(testDir) {
    [GetFrames]
    munge: getFramesImages(testDir)
    [GetSkinMask]
    update skin observing frame: skinMasks
    [GetGrayScale]
    update gray observing frame: grayScales
    [KeyFrames]
    update all keyframe observing gray, skin: keyframes(300)
}
```

The input and output of each decision function is, along with any additional arguments, an interpretation from the set that is under consideration by the algorithm at this point. Iteration over the input set of interpretations is handled transparently by the RSL environment. The decision function will process the input interpretation and may accept or reject it or create a new set of interpretations to be considered. Set operations on these interpretations are performed by the RSL en-

vironment so that each decision function only needs to manage the data specific to the algorithm. For example, while implementing the hand detection portion for the ASL recognition algorithm, an interpretation holds the hand candidates for a single frame. The set of interpretations holds an interpretation for each frame. An RSL programmer should not need to iterate manually over the set, sending each frame to the decision functions, nor should it be necessary for the programmer to filter out irrelevant fields and then reconstruct the new interpretation. This data manipulation and inclusion of the interpretation into history should all be handled by the language.

In addition to managing the operations over the sets of interpretations, RSL also records the history of those operations. Each interpretation is kept along with the information about which predecessor interpretation and decision generated the interpretation. This history may be examined and used to determine the accuracy or usefulness of particular steps in the algorithm. For example, in Listing 2.1 the result of the `GetSkinMask` decision can be compared with ground truth to determine if the generated mask actually represents the skin area in that interpretation's image. It is also possible to see which interpretations were selected as keyframes in the `KeyFrames` decision, and, when this is added to the rest of the hand detection algorithm, see if keyframe selection has any impact on algorithm accuracy.

In its original implementation [27, 26] RSL worked entirely with text and by passing set deltas to and from the decision functions. New research is being done to extend the work presented in these RSL papers [27, 26] and update the language. The results of this work and our contributions are described in Chapter 3.

2.2 Program tracing and logging

Although no work related to evaluating history for usability or fitness of an algorithm could be found, there is a lot of work dealing with general logging or tracing for correctness (i.e., debugging). Much of the work around debugging is not relevant because it seeks to isolate a code change that caused an error. The idea of a set of code changes and their effect on the code does not map well onto RSL or its history. The more relevant work has to do with creating a record of events in a program and analyzing that record.

In ”Aspect-Oriented Programming and Modular Reasoning”, Kiczales and Hilsdale [14] present a way to inject logging (as well as other ’aspects’) into code without modifying the algorithm. The logging code is kept in external classes, and the algorithm developer enables or changes logging for decisions whenever different information is required. This allows a computer vision developer to create history information and change it without modifying the decision, but Steimann [23] claims that even this capability is limited because ”aspects are not domain level abstractions and thus lack a significant source of diversity” (Steimann [23] admits to overstating his case to make a point, which is that Aspect-Oriented Programming (AOP) is limiting compared to other technologies).

RSL’s domain level abstraction is to treat the algorithm as a series of decision functions applied to a set of interpretations. AOP is applied at a lower level, likely inside the decision functions, preventing the abstraction that RSL provides. Additionally, while logging work is moved outside the algorithm to external code, AOP certainly does not lighten the burden of the developer in designing and imple-

menting logging mechanisms. Eaddy et al. [6] point out that "aspect functionality can drastically change the behavior and control flow of the base program, leading to unexpected behavior and resulting in the same complexity that multi-threaded programs are notorious for." AOP does not offer the simplification or abstraction that RSL is pursuing

Another debugging technique that offers some promise is Omniscient Debugging [20]. This technique uses an instrumented runtime environment that can log every single event, assignment, function call, etc., to a database for later replay and examination. Pothier et al. [20] demonstrate that omniscient debugging allows some of the same root cause analysis that RSL hopes to achieve (e.g., what was the origin of this interpretation). The storage demand of omniscient debugging is enormous, but it can be reduced by carefully specifying what to store. While root cause analysis is useful, omniscient debugging does not have much to offer when we want to understand the accuracy of a vision algorithm decision. For example, it would difficult to determine the accuracy of the skin mask generation or keyframe selection decisions discussed in Section 2.1.

Query-Based Debugging [15] offers the ability to track and record events in instrumented libraries (similar to the environment instrumentation of omniscient debugging, but much less extensive) and query the data of those events postmortem. Duca et al. [5] make good use of this technique to debug graphics routines using the OpenGL pipeline. In that case, the OpenGL library is instrumented to record state changes in the various parts of the pipeline which can later be queried using the domain specific query language GQL to help identify bugs. This is, in a

way, an inversion of RSL. In GQL, history is recorded at the lowest level to measure correctness; in RSL, history is recorded at programmer-designated decision boundaries, the highest level, to measure accuracy. It is interesting to note that in both cases, history-based analysis goes hand-in-hand with the creation of a domain-specific query language [5, 15].

While all of the above options are good for debugging in various environments, they all have shortcomings that limit their use for tracing accuracy or usability of a decision in an algorithm. Looking at the current libraries and tools in the computer vision field, like OpenCV [3] and Matlab [9], leads one to the conclusion that the general approach to both debugging and tracing history is the tried and true `printf` (or `cout` or `writeLine`, etc.) or file dump approach.

2.3 Domain-specific languages

Evaluating domain specific languages is a well researched but open problem [2, 12], as described in "When and How to Develop Domain-Specific Languages" by Mernik et al. [17]. In this paper, Mernik et al. [17] present a cyclical process for creating a domain-specific language such as RSL and discuss several criteria for evaluating the language throughout the steps of the cycle. Unfortunately, the criteria are not hard metrics that can be measured and reported easily. Rather, the criteria are qualitative items like "traversals over complicated data can be expressed better" in a DSL.

The DSL development phases laid out and explained by Mernik et al. [17] are "*decision, analysis, design, implementation,* and *deployment.*" During the deci-

sion phase the authors of the potential DSL consider pros and cons of developing a new language versus using existing languages and tools. In the analysis phase, the authors gather domain knowledge. This includes documenting terminology and semantics. It is in the design phase that the language syntax and semantics are described and the relationship to other languages is considered. During the implementation phase, the choices of compilation or interpretation and what tools will be used are implemented and tested. The syntax and semantics are put to use and evaluated. Finally, in the deployment phase, the language is packaged and documented for distribution and use. Mernik et al. [17] note that these phases are not necessarily linear and the results of any phase will feed back to cause changes in the preceding phase or even earlier. Most DSLs pass through these phases, and this includes RSL. This research participated in the implementation phase as described in Section 4, and, as a result, in the feedback and changes to the design as described in Section 3.

To help evaluate RSL against the criteria listed by Mernik et al. [17], guidelines for qualitative evaluation are provided in several papers. ”Little Languages” by Bentley [2] provides a series of items to consider when evaluating a domain-specific language. These include design goals, abstractions, simplicity, and a set of ”Yardsticks of Language Design” that are associated with ”tasteful” design. The paper ”Domain-Specific Languages” by Hudak [12] discusses the need for ”more natural ways to express the solution to a problem than those afforded by general purpose languages”, a need for the language to capture only the semantics required by the domain and the importance of powerful abstractions. In ”An Embedded

Bentley [2]	Hudak [12]	Mernik et al. [17]
Parsimony: Are there unnecessary operations?	Are the programs easy to maintain?	Gains in productivity, reduced maintenance costs.
	Can it be written quickly?	Is application interaction simplified?
	Does it follow the KISS (keep it simple, stupid) principle?	
Generality: Do operations have multiple uses?	Is the language concise?	Does the language offer appropriate domain-specific notations?
Orthogonality: Are unrelated features unrelated?		
Completeness: Can the language describe all objects of interest?		Expressiveness in the domain
Similarity: Is the language as suggestive as possible?	Can the language be used by non-programmers?	Availability and traversal of domain-specific constructs and abstractions
Extensibility: Can the language grow?	Are the programs easy to reason about?	Offers analysis, verification, optimization, etc. of domain-specific constructs
Openness: Can the user escape to related tools?		Is task automation allowed?

Table 2.1: DSL Evaluation Criteria

Modeling Language Approach to Interactive 3D and Multimedia Animation", Elliot [7] dwells on the need for composable operations in a language. This particular language characteristic has been demonstrated for RSL by Zanibbi et al. [26], where decisions from multiple algorithms were composed together to make new observations about the algorithm for that dataset. These evaluation guidelines are qualitative rather than quantitative; however, they are the best we can do at this time [17].

Table 2.1 lays out some of the criteria drawn from the sources mentioned. Some of these criteria were originally expressed as language goals, so a language would be evaluated according to the extent the goals are met. We have tried to cluster related criteria from different authors together. No one uses the exact same words or phrases for their criteria, but, when considered together, it is possible to find some commonality. Table 2.1 is not an exhaustive list, and not every criterion applies to every DSL. This research focused on a specific feature of RSL, the history, and a set of criteria based on RSL goals established during the analysis and design phases. Section 1.3 discusses this in detail.

2.4 Recognition of continuous American sign language

In "Handling Movement Epenthesis and Hand Segmentation Ambiguities in Continuous Sign Language Recognition Using Nested Dynamic Programming", Ruiduo Yang [22] presents an algorithm for recognizing continuous American Sign Language (ASL). That algorithm was implemented and modified as part of this research in order to examine RSL's utility in vision algorithm development.

The ASL recognition algorithm is broken into two main stages for this implementation. The first stage is feature extraction, specifically hand detection, which finds likely hand candidates in each frame based on frame to frame movement and skin filters. Hand detection is followed by the sign detection stage that uses a nested dynamic programming algorithm to find the most likely sentence.

There were several reasons to use this algorithm to evaluate RSL. First, the two stages of the ASL recognition algorithm are very different, allowing the use of RSL in different ways on the same data set. Also, there were questions about the suitability of RSL for a dynamic programming solution. RSL is intended to enable the ”white-box evaluation” of algorithms [27], but dynamic programming involves the examination of every possible solution, so the utility of RSL for a dynamic programming algorithm is questionable. We have found in the past that applying language to both suitable and unsuitable tasks can be valuable in learning and evaluating that language, and the two stages provided that opportunity. Second, this ASL recognition algorithm uses available datasets. As a result, less time would be spent gathering data leaving more time for evaluating RSL. Finally, the ASL recognition algorithm is explained in detail. A common problem with development and evaluation of DSLs is that few people have knowledge of both the domain and languages [2, 17]. We hoped the detail provided by Ruiduo Yang [22] would help to compensate a lack of computer vision and pattern recognition domain knowledge.

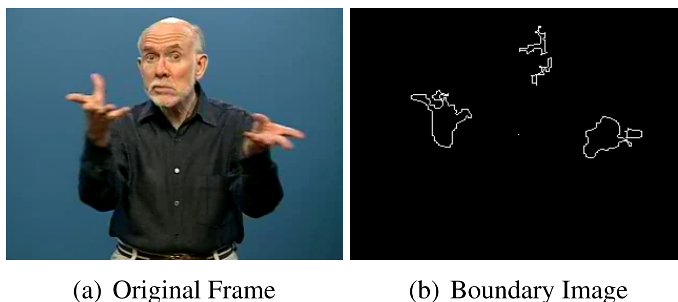
2.4.1 Hand detection

Keyframe selection The hand detection algorithm uses motion and skin detection to find likely hand candidates. This process begins by selection of keyframes, a step described in detail by Ruiduo Yang [22]. Keyframes represent series of frames that have little change between the images. In other words, there is little motion across these frames. The amount of change is determined by creating a difference image between a frame and the last keyframe and finding the largest connected component in that difference image. If the area of that largest connected component exceeds a threshold (T_1), then the image has changed a lot; i.e., there has been significant movement between this frame and the last keyframe, and a new keyframe is marked.

The area of the largest connected components is supposed to be measured in valid pixels, but "valid" is not defined by Ruiduo Yang [22]. We assumed it to mean skin pixels. The T_1 threshold depends on the frame size of the video and needs to be determined for a particular dataset. Image differences, connected component detection and measuring, and almost all image processing done in this research were done using the OpenCV library [3].

Hand candidate boundary generation Hand candidates are computed by finding the boundaries of the moving skin pixels in the image. First, a difference image for a frame is created by finding the average difference between that frame and all the keyframes. This is masked with a skin likelihood image to create the difference image SD. An edge filter E is created by running edge detection on SD and dilating those edges. Then E is removed from SD, and the small connected

components are removed to eliminate any remaining noise. The resulting image is called the ”motion-skin confidence map”. A boundary image is extracted from this result image, and the bounded regions are the hand candidates. Figures 2.1(a) and 2.1(b) show the original frame and the final boundary image respectively.



(a) Original Frame (b) Boundary Image

Figure 2.1: Hand candidate boundary detection

Histogram and space of probability functions Hand candidates need to be represented in a way that is conveniently comparable. Given the boundary image for a frame, a histogram is created from the horizontal and vertical distances of each boundary pixel to the center of the image. Distance values are grouped into thirty-two bins on each axis. The histogram is normalized to sum to one, and a space of probability functions [24] is generated by performing principal component analysis on the histogram, keeping only the first seven components. This is the final ’comparison-ready’ format for the hand candidates. The algorithm is trained with a similar space of probability function data for each known sign. The Mahalanobis distance is used to measure distance between the hand candidate and the possible sign match. Each known sign is compared to the hand candidate, and the sign with

the lowest Mahalanobis distance is the most likely match to the hand candidate.

2.4.2 Sign-matching dynamic programming

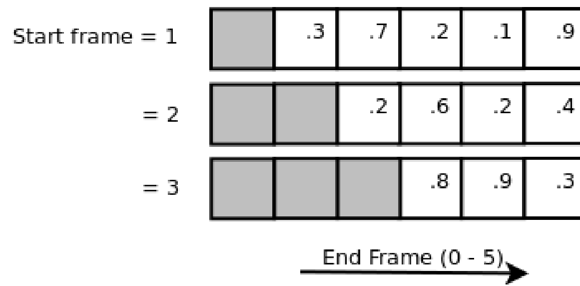
Matching the sign in a sequence of test frames to a sign in a sequence of model frames is more complicated than measuring the Mahalanobis distance between two histograms. A set of distances must be added to represent the distance of the entire sequence. There are a couple factors that complicate this aggregation of frame differences. First, the signing in test and model frame sequences may be done at different rates, or even at an inconsistent rate (i.e., the signer's hands may accelerate while signing). This means that the test and model frame sequences may be a different length, and the number of frames for any particular hand position could be different in each sequence. A second complicating factor is that the preceding hand detection step is not certain to detect only hands, nor will it label which hand is the left and which is the right.

Ruiduo Yang [22] addressed these complications with a dynamic programming solution. The dynamic programming solves for a three dimensional array. One dimension represents the frame sequence for the model sign being tested. Another dimension represents the frame sequence for the test video. The last dimension is for the hand-candidate pairs in the test frame. Any cell in the array can be addressed by `array[modelFrame][testFrame][handcandidate]` and contains the Mahalanobis distance of that hand candidate pair in that test frame from the hands in that model frame plus the best (lowest) distance of all possible preceding frames. Ruiduo Yang [22] give the recursive formula for dynamic

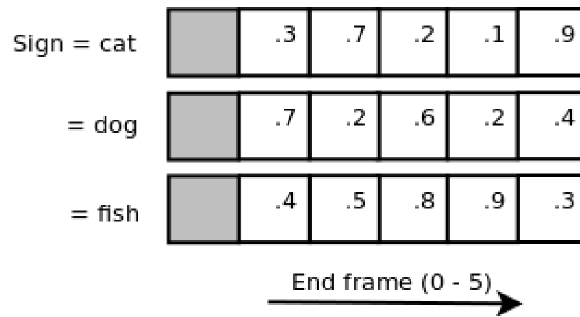
programming seen in Equation 2.1. This is the innermost of the nested dynamic programming algorithms.

$$Cost(i,j,k) = d(S^i_m, g_k(j)) + \min \begin{cases} \min\limits_{r, m(g_k(j), g_r(j-1)) \leq T_0} & Cost(i, j-1, r) \\ \min\limits_{r, m(g_k(j), g_r(j-1)) \leq T_0} & Cost(i-1, j-1, r) \\ Cost(i-1, j, k) & \end{cases} \tag{2.1}$$

2.4.3 Level building dynamic programming



(a) Distances for start and end frames



(b) Distances for each sign at each frame

Figure 2.2: Mahalanobis distances for frames

Sign selection A level building approach [21] is used to find the best fit sentence for a video by calculating the distance between every sign and every remaining

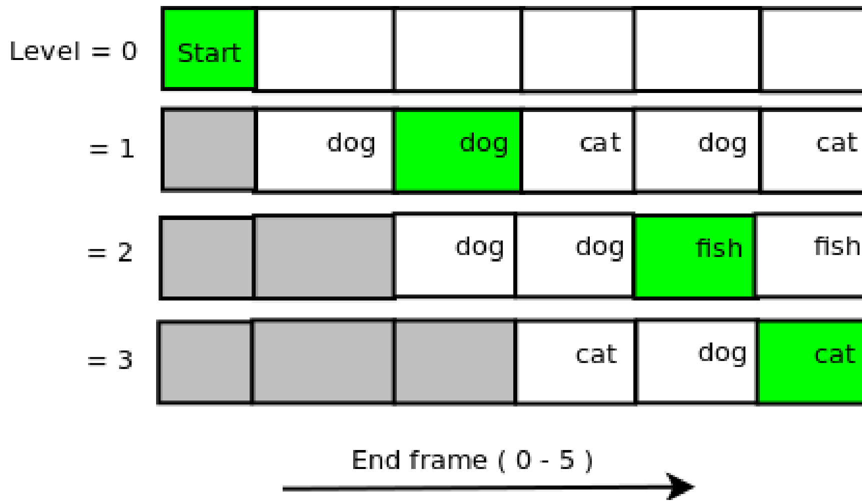


Figure 2.3: Best signs

frame subset at each level. For example, Figure 2.2(a) shows a possible set of distances calculated for some sign. For each starting frame, represented as rows, distances are calculated for all possible sequences up to an ending frame, represented as columns. The value in the cell is the distance of the frame sequence (start frame to end frame) from the sign being checked. This matrix is collapsed to represent only the best possible start and end pair for each end frame, so in Figure 2.2(a), the remembered distances would be `(1:1)`, `(2:2)`, `(1:3)`, `(1:4)`, and `(3:5)`.

This distance computation and selection is done for every sign on a given level, producing a matrix of signs as seen in Figure 2.2(b). Each row of this matrix represents a set of sign scores with each column holding the score for a particular end frame. For the level in Figure 2.2(b), the best sign for ending frame 1 would be `cat`, frame 2 would be `dog`, and so on.

Finally, the costs are accumulated. At each level, the best possible ending

frame score for each sign is added to the score for the predecessor - the best sign ending in the previous frame on the previous level. The result is a best sign at every level. Figure 2.3 shows a possible set of signs and levels. Each level finds the next word in a sequence, or sentence, that ends at each frame. The best sign ending at frame one in level one is `dog`. This is true for level one, frame two as well. However, the best possible sign for level one, ending with frame three is the sign for `cat`. Level one is complete when the best sign for every ending frame is found, taking into account the grammar rules. For example, the sign for `fish` may have had the smallest Mahalanobis distance, but if `fish` is not allowed to be the first word according to our grammar, it is not selected as an option in level one. Level two proceeds similarly, but now the score of the previous level is taken into consideration when selecting the best sign for an ending frame. For example, the distance for any interval (2:3) is that distance plus the distance of the best sign ending in frame one at the previous level. Since the total cost to reach a particular sign in a particular frame is accumulated through the algorithm, finding the best matching sequence is a matter of finding the best sign for the final frame and tracing its predecessors. The predecessor frame is stored for each selected end-frame and sign pair throughout the algorithm in order to allow backtracing.

Grammar In order to improve the accuracy of the search, a trigram grammar check is made for each sign. This check is based on a precomputed tree of legal signs. When the distance of a frame sequence from a sign is calculated, that value is used only if the sign can be found in the tree of valid sequences following the two

previous signs. When the sign is not part of a valid sequence, the maximum distance is used, effectively disqualifying the sign unless it is part of an extraordinarily good sentence match.

Modeling Epenthesis One of the contributions of this ASL recognition algorithm is the modeling of motion epenthesis. This is the movement between signs, a physical type of coarticulation. Along with measuring the distance between a sign and a frame sequence, the algorithm assigns a distance to ”matching” motion epenthesis. This distance, α, must be less than a non-matching sign, but greater than a matching sign in order to prevent mislabeling signs as epenthesis or falsely matching a sign. The actual distance depends on the training data. The cost of matching an epenthesis frame increases each frame so that even a poor match will eventually be preferred to a long epenthesis. For an interval (b:e), this is computed as $cost = \alpha * ((b - e) + 1)$.

2.5 Summary

We considered the history and goals of RSL as introduced by Zanibbi et al. [27] and compared RSL history to alternatives in general programming languages such as omniscient debugging and aspect oriented programming. Criteria for evaluating DSLs such as RSL were described and compared in Table 2.1. Finally, we described an ASL recognition algorithm that will be used to test RSL history.

Chapter 3

Evolving RSL

This chapter discusses the changes made to RSL from the initial version to the version used in this research. Some of those changes were made in response to findings during this research, incorporated into the language, and evaluated again. Table 3.1 summarizes the high-level changes made to RSL from the original version [27]. These are covered in detail in the Programmer's Guide to The Recognition Strategy Language [8].

3.1 Syntax and semantics

The original implementation of RSL as described by Zanibbi et al. [26] was quite different in both syntax and semantics from the current implementation. Previously, decision functions were broken into three categories: classification, segmentation, and parsing [26], and interpretations had regions, relations, and parameters. The current RSL syntax would not be described using these terms; however, the original goals are still met, and the decision-based description of an algorithm is still at the heart of RSL. Now the interpretation type is declared in a single block and relationships are not explicitly declared inside the RSL program. The number of operations on interpretations is down to a simpler six from the fourteen available

Syntax and Semantics	Significant syntactic and semantic changes have been made to RSL.
Implementation	RSL originally maintained all the interpretation data as text. The current version uses Standard ML [18] and the MLton compiler [19] to allow more powerful and efficient complex data structures
Recursive Functions	During this research we found that recursive functions were necessary to group interpretations for reporting or updating together. As a result, a small change was made to the compiler to generate functions that could be called recursively. The grouping technique is described in Section 4.2.4.
Trace model	An RSL program trace is stored in memory. This trace represents the program execution and contains references to the interpretations and annotations from every step.
Annotations	One of the contributions of this research was the observations that led to the creation of annotations. Using annotations, at each decision point the developer may record information relevant to that decision that does not belong in the interpretation. Accessor functions were also added to RSL to make traversing the trace graph and accessing the data easier. This is covered in Section 3.2
Second entry point	The first RSL implementation recorded its history to text files that were later examined using other tools. RSL has been changed and now has a second ”reporting” entry point that allows full access to the program trace and all history information from the first entry point.
Query operation	A new `hadd` operation has been made available during the second entry of an RSL program. The `hadd` operation allows the developer to add all interpretations or just a selected set to the current set being examined by the reporting function.
Layered design	A layered design for RSL programs was developed as part of this research that provides a clear boundary in responsibilities for each piece of code, based on which layer that code is in. This is described in Section 3.3.

Table 3.1: Changes to RSL

in the first version of the language. An annotation type has been added that allows the storage as history of relevant data that is not necessary in the interpretation. Also, examination of RSL history happens during the second entry point of the program instead of during post processing with separate tools as was required in the original RSL. These changes allow for RSL programs that should be easier to read and maintain.

The new syntax and functionality is enabled by a new RSL compiler that compiles to Standard ML (SML) code, then to machine code via the MLton compiler. This move to SML from a fully text-file based implementation allows the use of complex types in the interpretations and annotations, as well as enabling the creation of a second entry point that can traverse the execution graph and examine those complex types in the history. By coupling RSL with the MLton compiler [19], the RSL programmer now has access to fast interaction with external functions like those supplied by OpenCV [3] through the MLton foreign function interface (FFI). Using ML Basis files allows programmers to develop in the large with reusable components. For example, the hand detection decision functions used for this research could be reused by another RSL program, possibly unrelated to ASL recognition. As a result of these changes, the simpler syntax of the new RSL has a more powerful type system, access to extensive libraries in other languages, and the ability to grow its own set of reusable libraries and extensions.

3.2 RSL Annotations

When this research began, an RSL script expected decision functions would return only deltas from the input interpretations. For example, suppose a decision function `Inc` would accept an interpretation as a set of numbers and increment those numbers, each by a different amount. `Inc` may receive the set {`1, 4, 7`} and create the result set {`3, 7, 8`}. In this case, the result from `Inc` would be the set of differences between the new interpretation and the original interpretation, or {`2, 3, 1`}. The delta that generates each interpretation would be stored in the history and associated with the interpretation and decision function that generated the delta.

An RSL script proceeds through its decision functions, feeding the elements of the interpretation set to each function, collecting the resulting interpretation deltas, recording them to history, and applying them to create new interpretations. The new interpretation set is then fed to the next decision function.

This delta-based mechanism for updating interpretations was examined as part of this research. While the deltas for an interpretation provide all the data needed to trace the interpretation's history, that data may not be informative in delta form. Additionally information about why the change was made is completely lost. For example, consider a decision function `BCScale` to scale a bounding circle for some feature. `BCScale` accepts an interpretation containing a radius of `3` that is scaled three times to a circle with radius `9`. The delta returned from `BCScale` is `6`, but the meaningful information that the circle was scaled by a factor of `3` is not obvious. No information at all is available about why the bounding box was scaled.

In order to populate the interpretation history with more contextually meaningful data than deltas allow, and to provide information about the decision function result that is not part of the interpretation, a language modification was suggested that would allow recording of decision information that is meaningful to the decision function result but that does not belong in the interpretation. This modification eventually saw light in the form of *annotations*. A decision function now has the option of returning an annotation in whatever format is desired with information about what happened or why. As a result of this change, the decision functions no longer return deltas from the input set; if simple delta information is desired, it can be put in the annotation. Instead, the decision functions return the new interpretations. So, in the case mentioned above, `BCScale` would return the new interpretation with the circle with radius `9` and an annotation indicating that the circle was scaled by a factor of `3`.

Using annotations instead of deltas does not change the RSL script procedure significantly. Iteration and set operations are still managed by RSL. History records the decisions and interpretations and adds the annotations to the record instead of the deltas. This change encourages the use of history to store and analyze data that does not belong in the interpretation, simplifying and clarifying the interpretation itself.

When the algorithm has completed, the entire history is available for examination and reporting. Each decision function can be studied to determine its accuracy [26, 27], the interpretations can be traced through the program to see their source, and the annotations can be referenced to see what influenced the interpreta-

tion's creation.

3.3 RSL Style and Organization

When using a language for the first time, the syntax is learned, and then some time is spent learning to organize and partition a program in that syntax. RSL is no different in this respect. Following principles to code to the problem domain and to isolate complexity, RSL programs are separated into three major layers (Figure 3.1).

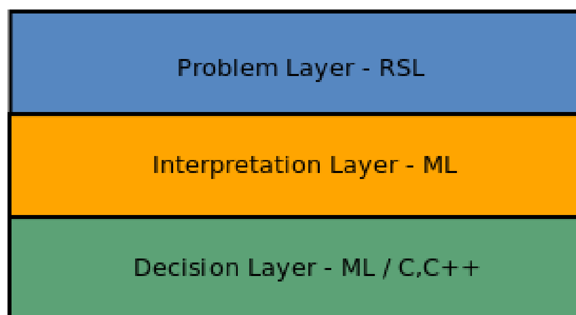


Figure 3.1: Layers of an RSL program

- The top layer is the RSL program itself. The code for an RSL program is in the language of the problem domain. In the case of ASL recognition in video, this meant coding with terms like `frame`, `hand candidate`, and `gloss`. The syntax of RSL dictates that this layer be limited to a high-level description of the problem steps: the interpretation structure and the possible annotations. While inlining Standard ML code is possible, it was found to be distracting in large sections, especially when dealing with the interpretations,

so this feature was used just to simplify certain initializations.

- Below the Problem Layer is the Interpretation Layer which is written completely in ML. This layer became necessary as the code to handle interpretations grew to be more complex. Listing 3.1 shows an invocation to a decision function to detect and mark keyframes. This is an `update all` call because a frame's state as a keyframe depends on the image difference from the previous frame. So all frames must be seen when checking each frame. In this case, the Interpretation Layer of the decision function in Listing 3.2 pre-computes all the keyframes with a call to `keyframeDiffs t1`, storing the differences, and then keeps a list of the keyframe ids in `keyIds`. Both functions used in keyframe calculation are defined in the Decision Layer, and the body of this decision function is a single expression devoted to the construction of an annotated function that accepts an interpretation and returns an annotated list of annotated interpretations.

  As you can see in Listing 3.2, the end of a decision function can look more like Lisp than ML. The `keyframes` decision function updates the state of a boolean in each interpretation. As the types in the interpretation grow in complexity to include lists and tuples, this construction expression can become quite complicated. To help mitigate this complexity, a style was developed that pushed the work of a decision function to a separate decision layer, leaving only the high-level steps and interpretation construction in the body of the decision function itself. In addition to isolating complexity, this layer has the effect of highlighting the content of the interpretations and annotations. The

Listing 3.1: RSL call to update keyframe

```
notes:
    FrameDiffs : real vector

interp:
    frameId : int
    keyframe : bool
    frame : char vector * int * int * int

fn findKeyFrames(t1) {
    [FindKeyFrames]
    update all keyframe
    observing frameId: keyframes(t1)
}
```

structure of the RSL layer, a high level description of the problem steps and the data they manipulate, has been mirrored on a smaller scale inside the decision function itself. This symmetry was satisfying even though the original motivation for the Interpretation Layer was to solve a readability issue.

- The third layer of the design is the Decision Layer. It is in this layer that the bulk of the decision function code resides. For the keyframe calculation example above, the Decision Layer handles the work of passing the frames to the OpenCV [3] library to calculate image differences, connected component areas, and any other image processing work that needs to be done. All the work of the algorithm happens in the Decision Layer, and when that is complete, the intermediate and final results are returned to the Interpretation Layer to be used in annotations and interpretations, respectively. The domain of the Problem Layer is the domain of the algorithm. The domain of the Inter-

Listing 3.2: Interpretation layer of keyframe decision function

```
fun keyframes t1 = fn  _ =>
  let
    val diffArray = keyframeDiffs t1
    val keyIds = keyframeIds()
  in
    (SOME (FrameDiffs diffArray),
     fn ({keyframe, frameId}) =>
      (NONE, [(NONE, { keyframe = (Vector.exists (fn i => i =
          frameId) keyIds) } )] ))
  end
```

pretation Layer is interpretations, annotations, decision function signatures, and accumulation of Decision Layer results. The domain of the Decision Layer is APIs and data transformations needed to calculate the results of the overall algorithm.

This three-tiered design, when represented as ML code using MLton's [19] ML Basis files and ML source files, forced a structure to the code and an include idiom on the RSL file itself. Decision functions are a large part of the code, with many external dependencies. These were packaged together into a ML Basis file as seen in Listing 3.3. ML Basis files must be self contained with no undefined references, so the decision layer ML Basis file may not reference any interpretation type, because that type is not known until the RSL compiler is run. This requirement forces the Interpretation Layer code into SML files because there are external dependencies, and an include file ordering is mandated in order to have all the decision functions defined before the interpretation functions that use the decision layer.

You can see the resulting programming idiom in Listing 3.4.

Listing 3.3: Decision Layer MLB file

```
local
    (* import Basis Library *)
    $(SML_LIB)/basis/basis.mlb
in
    $(DATAREAD_DIR)/aslio.mlb
    $(CVSL_DIR)/cvsl.mlb
end
```

Listing 3.4: RSL including layer files

```
inc "decision-layer.mlb"
inc "interpretation-layer.sml"
```

3.4 Extension

This research touched on the extensibility of RSL through the use of a small library that focuses specifically on computer vision.

3.5 CVSL

Computer Vision Strategy Language (CVSL) is an API based extension to RSL created for this research that provides computer vision specific tools to examine history. These tools provide functions to treat history information as images or histograms and to display or save these vision artifacts. The primary API for examining a computer vision history is in the CVSL ML signature, seen in Listing 3.5. This signature provides the ability to reference different frames or images with

a type id and a frame id. The type is meant to group frames or images with similar characteristics such as keyframes or skin confidence maps in the ASL implementation. Each frame has an id associated with it allowing a developer to access the twelfth frame of the skin confidence map, or the first keyframe, etc.

Every use of CVSL could potentially address a different computer vision problem. To allow this, CVSL is implemented in terms of a lower level API written in C since the image work is done with OpenCV [3]. Listing 3.6 shows the portions of this lower level CVSL API that is provided by the computer vision algorithm implementer. This API uses a type and id pair to identify each image. Both fields are determined by the implementer and can change with the problem. OpenCV image size and type information is made available about an image based on the type and id information. It can be useful to think of the type and id pairs as the tables and keys of an image database. The CVSL C API provides a way to access that database and get the images. This approach was selected when we found that algorithm decision operations would frequently access several image types at once. Copying the same three or four versions of each frame around for several steps of an image processing algorithm required a lot of management, so the database was added inside the decision functions, and CVSL was modified to take advantage of such a database. In the case that no database is required by the decision functions, the CVSL API also provides `getImage` and `showImage` functions that work with OpenCV `cv::Mat` data.

Listing 3.5: CVSL.sig

```
signature CVSL =
sig
    (* args: frame type and wait boolean
     * return unit *)
    val displayAll : int -> bool -> unit

    (* args: filename, frame type and id
     * returns unit *)
    val saveImage : string -> int -> int -> unit

    (* args: name, extension, type
     * return unit *)
    val saveAllImages : string -> string -> int -> unit

    (* args: frame type and id
     * return unit *)
    val displayImage : int -> int -> unit

    (* args: frame type
     * return unit *)
    val displayAllImages : int -> unit

    (* args: frame type
     * return unit *)
    val displayVideo : int -> unit

    (* Takes a type, return a vector of ids *)
    val getIds : int -> int vector

    (* takes a type and id, returns image, height, width, cv
       type *)
  val getImage: int -> int -> (char vector * int * int * int)

    (* take an image, width, height, and cv type *)
  val showImage: char vector * int * int * int -> unit
end;
```

Listing 3.6: CVSL.h

```
/*******************************************************
 *
 * These functions are specific to the problem domain and
 * must be implemented by the CVSL user
 *
 *******************************************************/
extern "C" {
    /*
     * Get the number of frames of a particular type
     * InitAslAlg() must be called first
     */
    int numFramesC(int type);
    /*
     * Get the ids associated with frames of a
     * particular type
     * InitAslAlg() must be called first
     */
    void getFrameIdsC(int type, Pointer ids);
    /*
     * Return the width, height, cv image type, and size of
     * a type of images
     * InitAslAlg() must be called first
     */
    void getFrameInfoC(int type, Pointer width, Pointer height,
                             Pointer dtype, Pointer size);
    /*
     * Return the image with id and type
     * InitAslAlg() must be called first
     */
    void getFrameC(int id, int type, Pointer img);
}
```

Chapter 4

Evaluation of RSL History

This chapter examines the use of RSL in the algorithm development (Section 4.1) and in the algorithm evaluation stage (Section 4.2).

4.1 Algorithm Development Stage

In this section, we describe how RSL was used in the development of the hand detection and dynamic programming portions of the ASL Recognition algorithm. We discuss the representation of each section and RSL's utility in debugging. We then evaluate RSL against the development criteria laid out in Section 1.3.1.

4.1.1 Representing the Hand Detection algorithm as RSL

As can be seen from Listing 4.1, the hand detection algorithm was implemented clearly and naturally in RSL. The process was laid out by Ruiduo Yang [22] in a way that allowed each step to be mapped to a labeled decision function. Using CVSL library functions described in Section 3.5, images were processed at each decision function and updated in the interpretation to be available for later analysis.

Listing 4.1: RSL function to find hand contours

```
fn getHandContours() {

    (* Section 4.1, step 2a *)
    [DiffImage]
    update all handImage
    observing keyframe, frameId, gray: initialDiffImages

    (* Section 4.1, step 2b *)
    [SkinmaskDiff]
    update handImage
    observing frameId, skin: skinmaskDiffs

    (* Section 4.1, steps c and d *)
    [EdgeAndMask]
    update handImage
    observing frameId: edgeAndMaskDiffs

    (* Section 4.1, step e *)
    [RemoveSmallComponents]
    update handImage
    observing frameId: removeSmallComponents

    (* Section 4.1, step f *)
    [BoundaryImage]
    update handImage
    observing frameId, handImage: extractBoundary
}
```

4.1.2 Representing the Dynamic Program as RSL

Representing the dynamic program in RSL presented a problem. The ASL recognition dynamic programming algorithm was defined recursively [22] and represented in the original source code as a single large array. Neither model mapped to RSL interpretations in a satisfying way, so a new approach was developed that took

advantage of the set tracking and iteration that RSL provides. Listing 4.2 shows the state information stored in the original implementation of this algorithm to track the algorithm's progress. The arrays `dist` and `prev` hold the distances and sentence to that point, and the rest of the variables track the current point in the algorithm, what frame is under consideration, etc.

Listing 4.2: State kept by original dynamic program

```
alSigns* subSentence=0;

int sp1=0;
int sp2=0;
int ep1=minSignLen/2;
int ep2=maxSignLen*2;

int Nt=testData->len; //length of test data
int Nv=signlist->GetSize()+1; //num of training template

if (ep1>=Nt)
    ep1=Nt-1;
if (ep2>=Nt)
    ep2=Nt-1;

float * dist=new float[maxLevel*Nt*Nv];
int * prev=new int[maxLevel*Nt*Nv];

memset(dist,0,sizeof(float)*maxLevel*Nt*Nv);
memset(prev,-1,sizeof(int)*maxLevel*Nt*Nv);
int s,e;
float tdist;
float bestsofar=1000000000;
```

Rather than manage the iterations, array, and state explicitly, each element of the array was represented as an RSL interpretation, seen in Listing 4.3

In the C++ implementation, the `dist` array tracks the distance for a sign at

Listing 4.3: RSL interpretation for dynamic program

```
interp:
    testFrames: int vector
    level : int
    word : int
    score : real
    interval : int * int
    prevs : int list * real
```

a particular end point at a level. The `prev` array elements hold the predecessors for the corresponding `dist` location. The `prev` array provides information about the end frame of the previous sign match, so, between the two arrays, an interval is created. This interval is represented directly in the interpretation, along with the level that interpretation is being considered for, the word (sign) the interval is measured against, and the distance for that word with that interval. Instead of a linked array of `prev`, each interpretation holds the list of predecessors for that particular word and interval.

Initially interpretations for each interval at every level were generated at once, then gradually updated. This generated well over one-hundred thousand interpretations in the first decision function, most of which did not contain any useful information. That early version of the compiler did not perform as well as later versions, so waiting a long time for the creation of essentially worthless (at that point) data did not make sense. The algorithm was revisited and adjusted so now only a single level is created at a time. That level is scored (the distance from the training data is calculated), the best previous interpretation is found and added to the `prevs` list, and all old interpretations are dropped. This keeps the interpretation

count in the tens of thousands, about an order of magnitude less than the original implementation. There were still performance problems associated with calculating the distance of all the frame intervals against all the training frames. This was addressed by generating the interval and distance data separately and storing that in a file. This is described in detail in Section 4.2.4.

Iteration over the intervals at each level is provided automatically by RSL's iteration over interpretations. Iteration over the words and the levels, the other two dimensions of the outer dynamic program, are implemented explicitly in the RSL code. This can be seen in Listing 4.4. The function `levelbuildLoop` uses a while loop to repeat until some maximum level is reached. The body of the loop makes a new level in the function `makeLevel` and then drops the old levels that did not completely match the frames. This ensures that only the current level remains in the active interpretation set. All interpretations, even those that are rejected, will remain in RSL history so they can be examined during reporting. This will allow us to see why each interpretation was rejected. Inside the `makeLevel` function, the RSL `munge` operation is used to generate a new set of interpretations for the next level. Then the previous level is filtered out and all the interpretations at the new level are scored. After scoring, the best legal previous sign is selected and added to the interpretation for each word. Each decision is labeled for later examination.

Listing 4.4: Iteration over levels in RSL

```
fn makeLevel(alpha, itemMap, grammar) {
    [NextLevel]
    munge: levelUpMunge(itemMap)
    (* LevelUp just created a set of interps at new highest
       level, so only look at those *)
    [InternalLevelCheck]
```

```
    if all observing level atMax {
        [ScoreLevel]
        update score observing interval,
        level, word: scoreLevel(alpha, itemMap)
        [KillHighScores]
        if observing score scoredOut {
            reject
        }
    }
    [GetPrevs]
    update all prevs
    observing score, interval, level, word: updatePrevs(
       itemMap, grammar, 3 )
    [InternalLevelCheck]
    if all observing level atMax {
        [TrimToBest]
        if all observing word,
        interval, prevs notBest {
            reject
        }
    }
}

fn levelbuildingLoop( alpha, numLevels, itemMap, grammar ) {
    [LevelCheck]
    while all observing level belowMaxLevel( numLevels ) {
        print all observing level: prlevel
        [OnlyHighestLevel]
        if all observing level atMax {
            makeLevel(alpha, itemMap, grammar)
            (* Drop the old level *)
            [DropOldLevels]
            if all observing level, interval,
            testFrames oldIncompleteLevel {
                reject
            }
        }
    }
}
```

As with the hand detection algorithm, there was no need to explicitly manage the construction or storage of history. At various points in the algorithm, inter-

pretations can be rejected for being too distant from a sign representing an illegal grammar, or for not being the best choice at this point. The history has the entire record of creation and rejection of each interpretation, and, since these reasons are all at separate decision points, the history shows the reason each interpretation was rejected.

4.1.3 Debugging RSL Programs

RSL programs work as scripts stepping over a series of decision functions. In this research the decision functions rely on a large body of algorithm functions that depend on the OpenCV [3] library. The amount of code that needs to succeed in the decision and interpretation layers before RSL starts to receive and store meaningful results in the history means that RSL has limited use during problem debugging.

A common location of code errors during this research was in the use of OpenCV. Unfortunately, the OpenCV library relies on runtime checks of type codes and other image information to detect errors. These checks result in calls to abort, preventing any information from being returned to the decision functions and the RSL scripts. In order to get any result from the image processing functions, they must all be provided with valid and appropriate data. Once all the input data and parameters are correct so that the image processing functions can work correctly, there are not many bugs left in the decision portion of the algorithm.

RSL was found to be of some help for the bugs that remained as a result of the trace graph RSL makes available in verbose mode [8]. The dynamic program-

ming loops were much more complex than the hand-detection portion of the ASL recognition algorithm, and the program trace provided valuable information during the later stages of debugging when all the lower level functions were behaving correctly.

4.1.4 Results of development stage

Our criteria under consideration for the development stage are:

- ”substantial gains in expressiveness and ease of use” - RSL development should substantially improve how the creation and examination of history is expressed over general purpose logging tools.
- ”task automation” - RSL should eliminate the tedious, repetitive work of creating and examining history, leaving only the problem specific work.
- complex ”data structure representation” - History information may be complex, and dealing with this complexity should be handled by RSL. The algorithm developer should have to do little work to generate or navigate history other than the work specific to the problem.

While structure and organization of code is nice, it is worth considering if the layered structure for RSL gives RSL history an advantage over logging. We believe that the first two criteria are met by our three-tiered design.

When implementing logging code for the C++ implementation of keyframe selection, a `cout` was added to the function calculating the image differences (Listing 4.5), and this line would be commented or uncommented to debug this portion

Listing 4.5: Logging of diffs in C++

```
Frame AccumKeyframes::operator() ( Frame lastKey, Frame next )
    {
     double diff = calculateDiff( next, lastKey );
     cout << "Diff: " << diff << endl;
     if( diff > T1 ) {
         keyframes.push_back( next );
         return next;
     }
     return lastKey;
}
```

of the algorithm. Since this particular function is buried many functions and several files deep into the hand detection algorithm, it is not easy to remember where to find the function when changes are necessary; grep was used more than once. In contrast, the RSL decision function in Listing 3.2, isolated to the Interpretation Layer, makes it clear where the annotation is being stored, offers options such as storing each difference with the associated frame, and provides reporting as desired without modifying the algorithm. Expressing what is being stored and the task of storing it for examination are, for this research, simplified compared to C++ logging. Isolating the annotation from the interpretation while automatically maintaining the association between them satisfies all three of the criteria under consideration for development. RSL provides a gain in expressiveness in creating history while at the same time automating the creation and representation of the complex history datastructure. The developer's only task for history creation is to provide the problem specific data.

When implementing the dynamic program, a new way to represent dynamic programming data had to be designed. RSL interpretations do not map intuitively to the array based representation usually associated with dynamic programming. This took some consideration; however, when an interpretation structure was selected, the actual implementation of the level-building dynamic programming solution was similar to dynamic programming solutions in other languages. Specifically, the implementation was done by using nested loops. Representing a data structure used in the domain in a non-intuitive way makes the implementation of dynamic programming in RSL harder than in another language, but, since this impacts only the creation of the history, not communication or storage that are the other elements to consider for development, we do not consider the effort to be much harder.

4.2 Algorithm evaluation stage

4.2.1 Algorithm evaluation criteria

As mentioned in Section 1.3.2, algorithm evaluation is the intended domain of RSL, and the area in which we must see improvement over traditional methods. As with the development stage, there is no quantitative measure of easier or harder, so we consider these qualitative measures from Mernik et al. [17]:

- complex ”data structure representation” - Evaluation will require access to the history generated during development. The initialization of this history should be easier than with general logging tools.
- ”data structure traversal” - Accessing the items of history should be done

more naturally than it would be with general purpose tools.

- ”interaction” - According to Mernik et al. [17], interaction with the application should be made easier. In this case, that would mean interacting with history and extracting information.
  - Is new information readily apparent that would not be noticed with general purpose tools?
  - Is it possible to produce new types of analysis that would be too difficult or time consuming with general purpose tools?
  - Are methods for visualizing the data accessible and informative?

4.2.2 Evaluating hand detection

Listing 4.1 shows the RSL code used to implement the hand detection algorithm. Each step of the `getHandContours()` function calls a decision function to implement a specific step of the algorithm. The `handImage` field of the interpretation stores the handCandidate image calculated to this point. The structure of this function is very similar to the C++ version of the same function, seen in Listing 4.6, with just a few important differences. In RSL, each call to a decision function is labeled for later reference (`DiffImage`, `SkinmaskDiff`, `edgeAndMaskDiffs`, `RemoveSmallComponents`, `BoundaryImage`). These labels are used later in the RSL program when examining history.

A similar storage and retrieval mechanism was written for the C++ code and used at the line `setItem( setSD, cleaned )`. The `setItem` and

Listing 4.6: C++ function to find hand contours

```
void FrameDB::makeSDs() {
    /* Section 4.1, Step 2a */
    FrameSet init = generateInitialSDs();
    /* Section 4.1, Step 2b */
    FrameSet SDs = maskedSDs( init );
    /* Section 4.1, Step 2c */
    FrameSet edges = getDilatedEdges( SDs );
    /* Section 4.1, Step 2d */
    FrameSet negated = negateAndMask( SDs, edges );
    /* Section 4.1, Step 2e */
    FrameSet cleaned = removeSmallConnectedComponents( negated
        );

    /* Store for later reference */
    setItem( setSD, cleaned );

    /* Section 4.1, Step f */
    FrameHandSet boundaries = getBoundaryImages( cleaned );
    for( FrameHandSet::iterator i = boundaries.begin(); i !=
        boundaries.end(); ++i ) {
        db[i->first.id].boundary = i->first;
        db[i->first.id].hands = i->second;
        db[i->first.id].handCenters = centers( i->second );
        db[i->first.id].histograms = generateHandHistograms( (i
            ->first).size(), i->second );
    }
}
```

`getItem` calls allow storage and retrieval of images in a simple in-memory datastore. The first argument is an accessor function for a field (in this case the `SD` field which holds the images referred to as SD in the paper) and the second argument is the collection of images to store. To store and examine a different step in the algorithm, a different collection would be passed to `setItem`, and the data already stored in the field would be lost.

Listing 4.7: RSL report function on number of hands

```
hfn report() {
    reject

    hadd ["DiffImage"]
    print "\n\nDiffImage\n"
    print all: sNumHands
    reject

    hadd ["SkinmaskDiff"]
    print "\n\nSkinmaskDiff\n"
    print all: sNumHands
    reject

    hadd ["EdgeAndMask"]
    print "\n\nEdgeAndMask\n"
    print all: sNumHands
    reject

    hadd ["RemoveSmallComponents"]
    print "\n\nRemoveSmallComponents\n"
    print all: sNumHands
    reject

    hadd ["BoundaryImage"]
    print "\n\nBoundaryImage\n"
    print all: sNumHands
    reject
}
```

To examine more than one algorithm step at a time, multiple fields would be added to the in-memory data store and multiple accessor functions would be written. Use of function pointers and C++ templates make this data access code fairly extensible for new fields, but the development of that code was time consuming and adding new, non-image, types was more difficult. As a result, each new step in the algorithm presented the developer with a choice between the overhead of adding a

data store field or reusing a field and losing the ability to compare results.

Listing 4.8: RSL report function on Hand Candidate Accuracy

```
hfn report() {
    reject

    hadd ["DiffImage"]
    print "\n\nDiffImage\n"
    print: sPrintDiffAccuracy
    reject

    hadd ["SkinmaskDiff"]
    print "\n\nSkinmaskDiff\n"
    print: sPrintDiffAccuracy
    reject

    hadd ["EdgeAndMask"]
    print "\n\nEdgeAndMask\n"
    print: sPrintDiffAccuracy
    reject

    hadd ["RemoveSmallComponents"]
    print "\n\nRemoveSmallComponents\n"
    print: sPrintDiffAccuracy
    reject

    hadd ["BoundaryImage"]
    print "\n\nBoundaryImage\n"
    print: sPrintDiffAccuracy
    reject
}
```

RSL does the work of managing that data store for the programmer and stores every intermediate state of each interpretation. This can be seen in Listing 4.7 and Listing 4.8. Both these reporting functions illustrate a programming idiom used throughout the algorithm evaluation phase in which each interesting step of the algorithm is presented to the same history function in order to examine the

interpretations at that step. Step-by-step analysis is one of the design goals of RSL [27], and this idiom achieved that goal in a way natural to the developer. In the `report` function, interpretations for a decision point are added, the report is run for each interpretation, those interpretations are cleared, and the process is repeated for the next decision point.

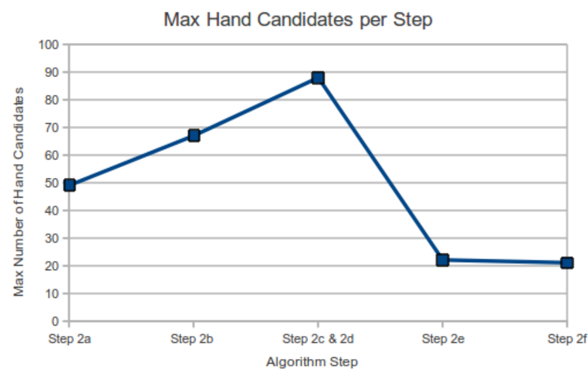
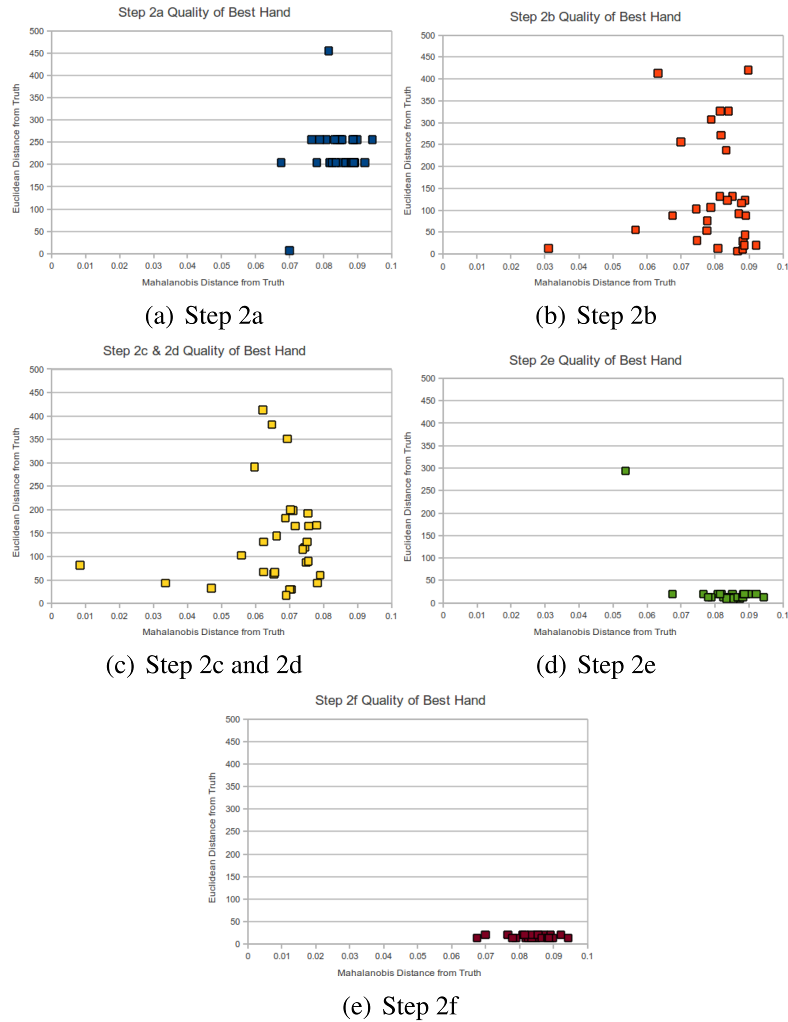


Figure 4.1: Maximum number of hand candidates across all frames after each algorithm step for the sign *why*

The dynamic-programming algorithm used for hand-candidate selection grows in both time and space with the number of hand candidates in a frame sequence. As a result, a benefit of the hand detection algorithm is to limit the number of hand candidates. Listing 4.7 calls a history function that displays the maximum number of hand candidates for a set of interpretations in a simple comma-separated-value format. This report was imported into graphing software to show the effect of each step of the algorithm on the number of hand candidates, and, as a result, the performance of the later dynamic programming algorithm. Figure 4.1 shows the results of the hand detection algorithm on the number of hand candidates.



(a) Step 2a (b) Step 2b

(c) Step 2c and 2d (d) Step 2e

(e) Step 2f

Figure 4.2: Euclidean distances for the hand candidate with the lowest Mahalanobis distance in every frame of the sign *why*

This research also examined the quality of hand candidates after each step of the hand detection algorithm. The `sPrintDiffAccuracy` function calculates the distance from ground truth of the image in the provided interpretation. This is done by finding the hand candidate in the image with the lowest Maha-

lanobis distance from ground truth and measuring the Euclidean distance of that hand candidate from the hand in ground truth.

The results were again printed to standard output as list of comma-separated values, but could have as easily been written to file using `write` instead of `print`. At every decision point there is an interpretation for every frame. Output from `report` describes the accuracy of each algorithm step for each frame. This is easily imported into graphing software (e.g., OpenOffice Calc, Matlab, or gnuplot) as seen in Figure 4.2.

We found that during the early steps of the algorithm, a number of hand candidates were detected that received a better (lower) Mahalanobis distance from ground truth but were too far away in the image to be the true hand. Mistaking an arbitrary background blob for a hand shape could throw off the accuracy of the sign language recognition algorithm. The elimination of closely matching, non-hand shapes was an interesting effect of the hand detection algorithm that may have been assumed but was not called out by Ruiduo Yang [22]. It seems possible that the removal of these hand-shape non-hand blobs could increase the accuracy of sign language recognition and not just improve execution time. This type of observation and potential route of investigation is what RSL is intended to enable.

Hand detection depends upon several thresholds [22] whose values are dependent upon the input data. Since the threshold units are in pixels, changing video size causes variations in algorithm behavior. This research examined how RSL history could assist in determining an appropriate threshold value for T_1. T_1 is the threshold used to determine if a video frame should be used as a keyframe. If the

area of the largest connected component in the difference image between image n and the last keyframe, then image n is the next keyframe.

Listing 4.9: C++ function to select keyframes

```
Frame AccumKeyframes::operator() ( Frame lastKey, Frame next )
    {
     double diff = calculateDiff( next, lastKey );
     diffs.push_back( diff ); // Put differences in history
     if( diff > T1 ) {
         keyframes.push_back( next );
         return next;
     }
     return lastKey;
}
```

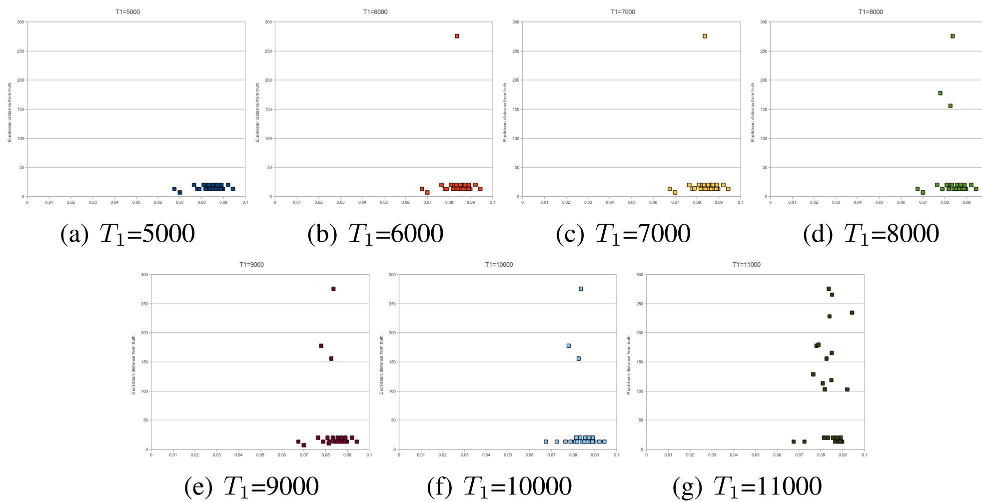
Listing 4.9 shows the keyframe selection function that has been instrumented to collect the frame to keyframe differences. This collection is stored in history as an annotation of the keyframe selection decision function as seen in Listing 3.2. The report function in Listing 4.10 prints the calculated differences at request. Once the differences have been reported, we have the information needed to select a range of values to examine.

Values from five thousand to eleven thousand were selected and tested to examine the effect of T_1 values on hand candidate quality for a short video of the sign *why*. The results can be seen in Figure 4.3. As you can see, at T_1 = 6000 hand candidate quality decreases for one frame of this dataset. That is, one of the frames has a hand-blob with a smaller Mahalanobis distance from truth than the true hand. Another drop in quality is seen at T_1 = 8000 with more frames with incorrect hands. Finally, at T_1=11000, many of the frames have a best-hand-

Listing 4.10: SML code to access annotations for image differences

```
(* Output attributes *)
fun noteToString (FrameDiffs ds) = (String.concatWith ","
                                     (List.map Real.toString (
                                        Vector.foldl op:: [] ds)
                                        )) ^ "\n"
  | noteToString (FrameId id) = ("FrameId: " ^ (Int.toString id
      ) ^ "\n")
  | noteToString _ = "Unexpected Note Value\n"

fun sHprintDiffs (is, ah, trace) =
  let
    val notesList = List.map (fn i => getar(Interp.rhcons i, ah
        , trace)) is
    val prFun = fn ns => List.app (fn (_,note,_) => print (
        noteToString note)) ns
    val _ = List.map prFun notesList
  in
    ""
  end
```



(a) T_1=5000 (b) T_1=6000 (c) T_1=7000 (d) T_1=8000

(e) T_1=9000 (f) T_1=10000 (g) T_1=11000

Figure 4.3: Hand candidate quality for values of T_1 for all frames of the sign *why*

candidate that is too far from the true hand.

4.2.3 Results of evaluating hand detection

Evaluation of the hand detection portion of the sign-language recognition algorithm allowed us to examine the qualitative measures of a domain specific language laid out by Mernik et al. [17].

- complex "data structure representation"

  Storage and retrieval of history information was compared to an extensible but problem specific data store written in C++. This data store is seen in Listing 4.11. The image for every intermediate step that needs to be examined later must be stored in a FrameData instance. This data store was useful for the ASL recognition algorithm to access the intermediate images, like the skinmask, that were used in multiple steps of the algorithm. However, when possible, we preferred to reuse storage rather than create a new slot in order to avoid the overhead of managing a new field.

  This data store is a C++ `std::map` of a frame's Id (frame number) to its various processed stages. Each decision function requires creating a new field in the `FrameData` structure or reusing an existing field, which would lose the information previously stored in that field. In comparison the RSL history is more complex, containing a tree tracing the execution of the algorithm and making the interpretations and annotations at any point available for examination. This eliminates the need to choose between creating extra storage

Listing 4.11: C++ Datastore definition and accessors

```
struct FrameData {
    int id;
    Frame original,
          skinMask,
          gray,
          SD,
          boundary;
    ContourSet hands;
    CenterSet handCenters;
    HistogramSet histograms;
    ProjectionSet projections;
    FrameData( int i, const cv::Mat &img );
    // Don't use. Provide for std::map
    FrameData() {}
};

typedef std::map<int, FrameDB::FrameData> DBType;
typedef DBType::value_type RowType;

std::vector<int> ids() const;
// Access frames of various types
FrameSet originals() const;
FrameSet grays() const;
FrameSet skins() const;
FrameSet sds() const;
FrameSet keys() const;
FrameSet boundaries() const;
```

fields or reusing a field and losing the previous value.

The creation of this data store took time and testing, and extending it for new types would require more work. Helper functions to access each data member (not shown) were required. Those helper functions would also have to be extended to support new items. RSL history is created and maintained automatically, and new types are added on demand with no more work for

the developer than declaring them. Accessing the fields of the interpretation during reporting is done by running the `hadd` query function for the desired decision function.

For the criterion of data structure representation, RSL provides a more powerful data structure with less developer work than the C++ implementation. This makes using RSL easier than using traditional logging methods and gives RSL history the advantage in this case.

- ”data structure traversal”

  Accessing history is done post execution as needed. C++ logging was able to provide easy access to the image difference calculations for keyframe selection. For this purpose, the logging was initially easier than accessing the annotations in the history. As a result of this research, annotation access helper functions were added so that the `sHprintDiffs` report function in Listing 4.10 is able to extract and print all the annotations from the provided interpretations. As you can see, traversing the execution graph to a particular decision point and accessing the interpretations at that point is done with a `hadd` query operation, making it easy to include or remove the image difference calculations from the report by updating the report function without searching through the C++ code for the logging method.

  The analysis of hand candidate quality across the algorithm steps was not completed in C++, because managing the data store at that resolution required too much code overhead. In comparison, storage and traversal of history at

different decision points made the analysis of the algorithm steps easier, requiring only the problem specific code (i.e., measuring hand candidate quality) be written by the developer. Simple access to annotations and support of the more complex analysis and traversal by RSL where it was prohibitively difficult in C++ means that RSL history is easier to use than traditional logging.

If other implementations repeat the idiom of applying the same reporting function to interpretations at various decision points, it may be worth investigating supporting this more succinctly, possibly with a mapping type operation.

- ”interaction”

  By applying RSL reporting mechanisms to history, we were able to examine the effectiveness of the hand detection algorithm at every step and notice an interesting elimination of false-positive hand candidates from the test image. It was this type of examination that was intended by the original RSL [27]. As mentioned above, the work required for this analysis using the C++ logging was prohibitive, so use of RSL history is easier in this respect as well.

4.2.4 Evaluating the dynamic program

The level building dynamic programming algorithm was run against five sentences listed in Table 4.1. These sentences were chosen from the data set provided by Ruiduo Yang [22] because they all share at least one word and span lengths

Sign sequence	*Sentence*
lipread can i	I can lipread
lipread cannot i	I cannot lipread
i understand not	I do not understand
don't-know i	I don't know
i need that i	I need that one

Table 4.1: Sentences tested

of two to four signs. The various sentence lengths mean that truth is found at different levels while the reuse of words in different sentences may allow more than one result to be considered by the algorithm.

The dataset we used included five instances of each sentence; however, the ground truth data was not available for one instance of those five, so we were unable to use it in training or testing. As a result, we trained with three sentences and tested (tried to recognize) the fourth.

Our work broke the dynamic programming part of the algorithm into three stages: Scoring, Level Building, and Reporting. This was done for two reasons. First, the scoring portion of the algorithm is time consuming, with longer sentences requiring more than a week of processing time with the current implementation and hardware. Optimizing this stage would be the next obvious step in further development of this solution now that it is shown to work. The second reason for separating the stages is to allow aggregation of all the separate tests so they could be considered together during reporting.

Scoring stage Stage one, calculating the Mahalanobis distance or scoring, is implemented in Listing 4.12. This script works by creating a single level of the level building algorithm and finding the Mahalanobis distance for every legal interval from every possible word. The call to `print all: dumpScores(destFile)` creates a data file called `allMahalanobisDists.dists` in the directory being tested. Later, during the level building stage, this file will be loaded into memory and work as a memoized cache of scores so that stage can be run quickly and separately.

Memoization and storage to disk was added late in the project as an optimization. Many of the scores are discarded deep inside the decision function before leaving the decision layer and becoming interpretations. As a result, using RSL history to cache and store these scores doesn't make sense. It would require a custom script that treated raw scores as interpretations just to write to file a data structure that is easily written already. Furthermore, the level building RSL script would need to read those scores in and pass them through all three layers to make the scores available to the lowest level of the decision layer. These steps would have nothing to do with the rest of the work of the level building RSL script or the interpretation layer. For these reasons the function `dumpScores` is used to save the calculated distances to file.

Level building stage As mentioned in Section 2.4.3, this ASL recognition algorithm relies heavily on the distance value associated with motion epenthesis frames. This value α is "the optimal Bayesian decision boundary between match and non

Listing 4.12: RSL script for scoring all intervals

```
interp:
    testFrames: int vector
    level : int
    word : int
    score : real
    interval : int * int

fn main ( testDir, trainDir ) {
    (*ML*)
        val _ = aslalgLoad trainDir testDir
        val dumpFile = (testDir ^ "/allMahalanobisDists.dists")
    (*ML*)

    [Init]
    munge: init
    [LoadVideo]
    update testFrames: getIds
    print "Trained and loaded\n"

    [LevelZero]
    update level, word, interval, score
    observing testFrames: levelZero()

    [NextLevel]
    munge: levelUpMunge()
    [InternalLevelCheck]
    if all observing level atMax {
        print "Scoring "
        print all observing level: len
        [ScoreLevel]
        update score observing interval, word: scoreLevel
    }
    print "Done scoring\n"

    print all: dumpScores(destFile)
}
```

match scores". The α value was found by running the scoring script against known sign intervals. Since these scores were already dumped to a file for the level build-

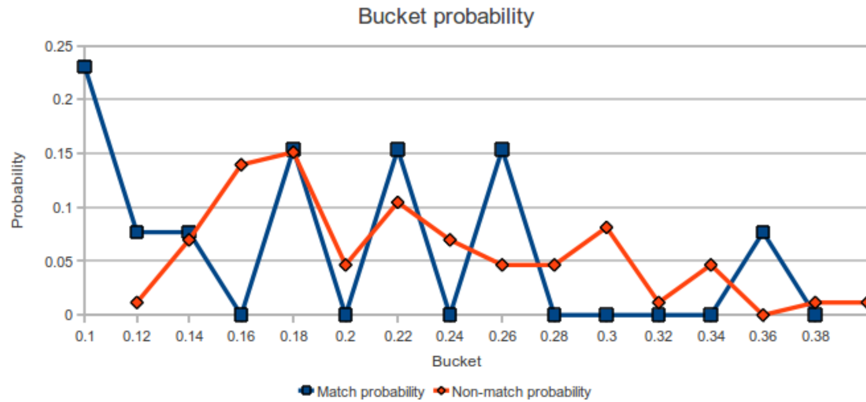


Figure 4.4: Probability mask function of match vs. Non match for all trained signs

ing stage, it was only necessary to load the correct intervals into a spreadsheet to determine α. The result can be seen in Figure 4.4. The match versus non match probability has not separated cleanly, probably as a result of too little data. If more signs were being tested, we expect the decision boundary would be clearer. Based on these results, we chose to test the level building algorithm with α values of 0.16 and 0.28.

The results of the level building algorithm can be seen in Table 4.2. This table shows for each α value and sentence whether the ground truth sentence was the best sentence detected (*best*) and if the ground truth was ever considered (*found*). In this case, considered means that is the best sentence for some complete set of intervals, but not necessarily the best sentence when compared to all possible sentences. Overall the performance of our implementation leaves something to be desired. The only consistent successes were with the two sentences ”lipread can i” and ”lipread cannot i”. ”i need that i” was found by the level building algorithm, but it was never

	$\alpha = 0.16$		$\alpha = 0.28$	
	Best	Found	Best	Found
lipread can i	no	yes	yes	yes
lipread cannot i	yes	yes	yes	yes
i understand not	no	no	no	no
don't-know i	no	no	no	no
i need that i	no	yes	no	yes

Table 4.2: Level building results

the best score. ”don't know i” was never considered when that sentence was tested; however, it appeared in the considered list for other sentences. ”i understand not” never appeared in the list of considered results for any test sentence. On considering these results, we think the choice of the ”lipread can i” and ”lipread cannot i” sentences, while an interesting test, may have been problematic. The sign for lipread consists of small movements by the bottom of the chin and upper chest, and the sign for i is also made up of small movements near the upper chest. This could lead to confusion between the signs, especially when considering we were training with only three-quarters of the training data used in the original implementation. Further complications may have arisen because several of the signs (lipread, understand, and don't-know) cross the signers face, making hand detection more complicated.

Despite the relatively low accuracy of the implementation, we felt that the results were consistent enough to move ahead with the evaluation of RSL for development and evaluation of this ASL recognition algorithm.

Because the test video files are all stored separately, and we wanted to consider all the results together during evaluation, the level building script's report

function pulls in the interpretations at the end of each level along with the final interpretations and stores those to a file to be combined with all the other tests. This can be seen in Listing 4.13 where `levelbuildingLoop` has an `accept` labeled `LevelEnd` at the end of the leveling loop. This allows the `report` function to use `hadd["LevelEnd"]` to add all the interpretations at the end of each level. We ran the level building with three α values for each of the five sentences. The interpretations were all written to files for aggregation during reporting.

Listing 4.13: RSL script for Level building

```
fn levelbuildingLoop( alpha, numLevels, itemMap, grammar ) {
    [WhileLevel]
    while all observing level belowMaxLevel( numLevels ) {
        print all observing level: prlevel
        [OnlyHighestLevel]
        if all observing level atMax {
            makeLevel(alpha, itemMap, grammar)
            (* makeLevel() made a new, higher level. Drop the
                old one *)
            [DropOldLevels]
            if all observing level, interval, testFrames
            oldIncompleteLevel {
                print all observing level: prlevelAt( "
                    DroppingOld" )
                reject
            }
        }
        [LevelEnd]
        accept
    }
}

hfn report ( alphaStr, levels, testDir ) {
    (*ML*)
        val ifilename = testDir ^ "/levelbuilding-" ^ alphaStr
            ^ "-ifile"
    (*ML*)
    reject
    hadd ["LevelEnd"]
    hadd ["Finished"]
```

```
    write ifile to ifilename
}
```

Reporting stage Reporting is implemented with two scripts. The first script (Listing 4.14) holds a hard coded list of interpretation files from the level building stage whose interpretations are added to the set using the RSL `add` function. Then the ground truth for that interpretation file is loaded into the interpretations using the `loadTruth` decision function so that the interpretation contains both the ground truth and the level building results. All the interpretations are then written to a user-provided output file. The script in Listing 4.14 shows only two files, but our experiment required fifteen interpretation files (five sentences at three runs each) which resulted in 92,015 interpretations.

Listing 4.14: RSL code to combine interpretation files

```
interp:
    file: string
    alpha : real
    truth : (string * int * int) list
    range : int * int
    level : int
    word : string
    score : real
    interval : int * int
    prevs : (string * int * int) list * real

fn main( outfile ) {
    (* ML *)
        val infile01 = "/path/to/interpfile01"
        val infile02 = "/path/to/interpfile02"
    (* ML *)
    add infile01
    if observing file nofile {
        update alpha, file, truth, range: loadTruth(infile01)
    }
```

```
    add infile02
    if observing file nofile {
        update alpha, file, truth, range: loadTruth(infile02)
    }
    write ifile to outfile
}
```

With all the interpretations combined into a single large set, it is possible to collect a series of reports using just RSL without falling back on external shell scripts to manage files as would have to be done if all fifteen interpretation files were to be processed separately. It is also possible to consider the performance of the algorithm across all sentences or all α as desired. In order to report on groups of interpretations, it was necessary to make a small change to the language to allow recursive functions. The use of recursion for grouping can be seen in Listings 4.15 and 4.16

Listing 4.15: SML code to group interpretations

```
fun isNextGroup is =
  let
    val {truth = firstTruth, alpha = firstAlpha, level =
        firstLevel} = hd is
  in
    (NONE, fn {truth, alpha, level} => (NONE, truth =
        firstTruth andalso level = firstLevel
                                               andalso Real.==(
                                                   alpha,
                                                   firstAlpha)))
  end
```

Listing 4.16: RSL code to group interpretations

```
fn eachGroupStats( outfile ) {
    if all observing truth, alpha, level isNextGroup {
        write ++ all to outfile: prGroupReport
    }
    else {
```

```
        eachGroupStats( outfile )
    }
}

fn main( interpfile ) {
    add interpfile
    eachGroupStats( "reportFile" )
}
```

SML function `isNextGroup` accepts a set of interpretations and returns a predicate that returns true for every passed interpretation whose ground truth, α value, and level building level all match those of the first interpretation in the set. When called from the RSL script's `eachGroupStats` function as the decision function of the `if` expression, the set of interpretations is effectively partitioned into those interpretations with the same ground truth, α value, and level building level as the first interpretation and those interpretations with differences. This first group is passed into the true branch of the `if`, where the reporting happens, while the rest are passed to the `else` branch. The else branch makes a recursive call to `eachGroupStats` with *only the remaining set of interpretations*. This continues until all the groups are passed to the true branch of the `if` and only the empty set remains for the `else` branch. In this example, there is no guarantee of what order the interpretations will be handled by the reporting function. However, this could be accomplished by sorting the interpretation set in `isNextGroup` before selecting the first interpretation for comparison.

One of the items we investigated was to see if the error would change across the levels with the number of true interpretations. The results for sentence ”i need that i” are shown in Figure 4.5. These figures show the percent of interpretations

that match ground truth with the purple bars. A match meaning the words appearing in the interpretation are a prefix of the truth. The minimum, maximum, and average edit distance error for all interpretations are shown as red, blue, and green lines, respectively. Edit distance is computed as a percentage to allow comparison between sentences of different lengths. An absolute edit distance of five for a sentence ten frames long is a very different result from the same edit distance for a one hundred frame sentence. Therefore we compare the results of the level building dynamic programming algorithm to the known truth for each sentence to calculate the edit distance, then divide that distance by the total number of frames in the test video to determine the error. The sentence is labeled at every frame when computing the edit distance. For example, if the signs in "lipread cannot i" are all two frames long with two frames of motion epenthesis in between the truth string provided to the edit distance function would be "lipread lipread *epenthesis epenthesis* cannot cannot *epenthesis epenthesis* i i". All error rates are much higher for the higher α value. Should we have more training data, we suspect this indicates that the optimal α value would be closer to 0.16 than 0.28. Note that the percentage of interpretations matching ground truth jumps about ten percent at levels 8 and 9 where the correct sentence (including motion epenthesis frames) is detected. This led us to investigate if similar jumps could be seen with the other sentences being tested. The other sentences in which the correct sentence was found did not display similar behavior. Another item of interest is that level 10 shows no interpretations that match ground truth (0%). Since "i need that i" is the longest sentence in our training base, and, with motion epenthesis frames, 9 levels is enough to find the

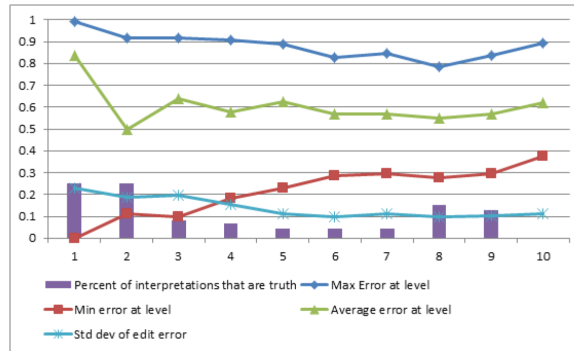
entire sentence, no valid sentences exist with 10 signs. Initially we graphed only the minimum, maximum, and average edit error for each level. The minimum error increase with each level is as we expected; however, the average remains fairly flat. This was an unexpected result and caused us to add variance to the graphs to have more insight into what was happening in the algorithm.

We next considered the error rates of correct (true) interpretations versus incorrect interpretations. These results are in Figure 4.6. These lines compare the average edit error for all the interpretation that match ground truth to the average edit error for the interpretations that do not match ground truth. The error rates for the correct interpretations are high, indicating that the frame boundaries for the signs are incorrect even if the words selected are correct. What we are probably seeing is the effect of the grammar. A single good match on a unique word will cause that sentence to be selected even if all the other intervals are far off. For example, if the Mahalanobis distance for the word ”can” is very low for a frame interval, the entire sentence ”lipread can i” will be selected even if the intervals for the other sentences are from correct. This will result in a sentence match with a large edit distance.

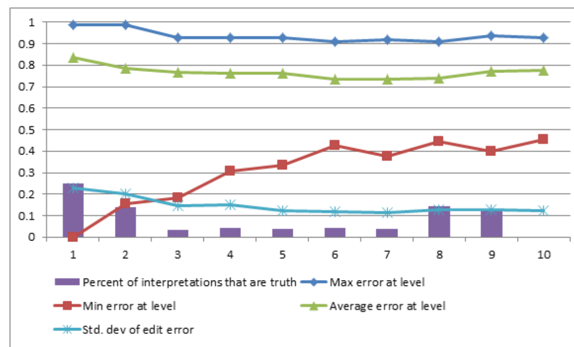
4.2.5 Results of evaluating the dynamic program

In this section we consider the use of RSL in the evaluation of the level building dynamic programming algorithm in terms of our criteria from Section 1.3.2.

- complex ”data structure representation”

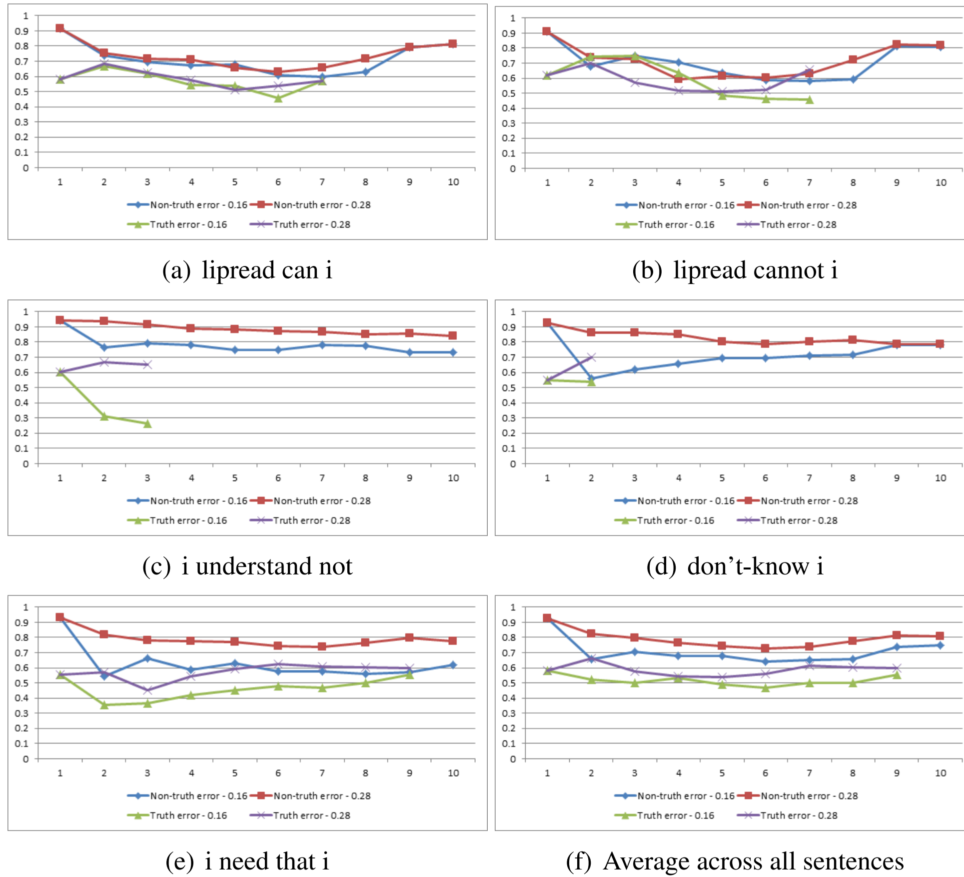


(a) $\alpha = 0.16$



(b) $\alpha = 0.28$

Figure 4.5: Average edit errors for all interpretations and the percent of interpretations that represent ground truth across all levels for sentence "i need that i". Edit error is the percent of the frame labeling that needs to change to make the interpretation match ground truth. Levels with no bar for interpretations that represent ground truth have no correct interpretations.



(a) lipread can i

(b) lipread cannot i

(c) i understand not

(d) don't-know i

(e) i need that i

(f) Average across all sentences

Figure 4.6: Average edit errors for correct interpretations versus incorrect. Edit error is the percent of the frame labeling that needs to change to make the interpretation match ground truth.

While both implementing and evaluating the level building algorithm, the RSL trace graph was found to be useful. This is a graphical representation of the program trace. Any time we have needed this type of information for other projects, debuggers and trace statements were used to follow algorithm progress. The visual graph along with information about what sets of interpretations were created or re-created was both complicated and useful information.

The level-building program tagged at a number of decision points throughout development to determine what each level looked like at the end, to verify that the appropriate interpretations were generated at level-up and that the grammar was rejecting the correct interpretations. This was all achieved by small changes to the RSL `report` function, with no framework or logging code required other than the code needed for the program-specific interpretation structure.

- ”data structure traversal”

  As with hand-detection, accessing the level building algorithms history at various points was fairly easy and intuitive. The only struggle encountered in this area was a desire to group interpretations according to level or α value. This was resolved by adding recursive functions to the language; if this needs arises in other implementations, it may be useful to add a grouping function to RSL. This would eliminate the need to write predicates and special recursive functions for every group type.

Unlike the hand-detection portion, the level building algorithm made use of RSL's ability to export and import interpretations from file to load multiple files into a single set of interpretations. This required writing the `toString` and `fromStream` functions, but, again, only the problem specific portions of the code were provided by the author. RSL silently manages the meta data that exists in the interpretation files.

- ”interaction”

  The RSL report exposed an interesting artifact; the number of correct interpretations increases suddenly for one of the sentences. Using the same reporting for other sentences, we were able to determine that this was not generally the case and not an interesting behavior of the algorithm. While this result is not particularly exciting, it is the type of observation and investigation that RSL intends to enable.

  As with the hand-detection, we found printing interpretations to simple comma separated value files to be convenient ways to provide to graphing software the algorithm's data at any decision point.

4.3 Algorithm implementation difficulties

A series of minor problems were encountered while trying to replicate the experiment described by Ruiduo Yang [22]. These problems do not cause us to question those results, but made repeating the results quite difficult.

- Incomplete data - The dataset for this work was generously provided by the

paper authors, but we found that ground truth was included for only 80% of the data (four out of five sentence instances). This resulted in a significant reduction in the amount of data to train the algorithm, likely reducing the accuracy of our implementation.

- Inaccurate data - The ground truth data was found to have errors that cause the hand contours to generally be twice as high as they should be. We tried to compensate for these errors by carefully designing the truth loading algorithm to detect and correct the "jumps" in pixel positions, but the result is a ground truth dataset that is not completely correct nor is it wrong by a known amount.
- Inaccurate implementation - The source code for the original paper was made available for comparison during our work, and we found that the histogram process was different from that described in the paper. The implementation scaled every x value by two. The reasons for this were not clear to us. Possibly the authors were trying to compensate for the improperly scaled ground truth data? We duplicated this scaling in our implementation, but we are uncertain of its importance.
- Missing information - We chose this algorithm because of the detailed information in the paper about each step, and it was very helpful to us throughout the work. However, we found that some more information was required. When selecting hand pairs from the hand candidates, it wasn't clear what rules should be applied in every case. For example, the description says that hands were limited to one hundred pixels of movement from frame to frame,

but it wasn't clear that this applied to each hand, or how to handle dropped (undetected) hands. Another question arose regarding only face pixels being selected in a frame. It may be better to try to match an empty frame rather than try to match the face blob. It wasn't clear what the correct approach should be in this case.

- Undefined terms - The term ”valid pixels” was used by Ruiduo Yang [22] to describe the hand detection algorithm, but no definition of valid was provided. We assumed this meant skin pixels. If we were incorrect, this may have impacted the accuracy of the algorithm.

- Non-standard implementation - In the original implementation, many of the standard algorithms for detecting contours or computing Mahalanobis distance were fully implemented instead of calling out to a common library such as OpenCV [3]. As is often the case, these implementations were tightly coupled to the data structures and context of that program and not reusable by others. This became a problem when we found that the algorithm thresholds were very different from those described in the paper. We did not know if there was difference in data (maybe the images were scaled), or if there was a difference in the algorithm for the image processing steps, or if there were a bug in one or both implementations.

The paper authors responded quickly and helpfully to our requests for assistance, and in the end we believe we have approximated the algorithm laid out by

Ruiduo Yang [22], but there are a few unanswered questions in the items above that may harm the performance of this implementation.

Chapter 5

Conclusion

This chapter presents the conclusions of our evaluation of RSL, lists contributions made as part of this work, and presents possible future directions for this ASL algorithm implementation as well as possible changes to the RSL language.

5.1 Summary of RSL evaluation

Hypothesis Recognition Strategy Language (RSL) history will provide information useful for evaluating decisions that would be more difficult to obtain using traditional logging methods.

In order to test our hypothesis we chose to consider RSL in two stages of algorithm design, development and evaluation, for each of the two stages of the ASL recognition algorithm.

Table 5.1 shows the criteria for the development stage. In the hand detection portion of the algorithm, we found that RSL made the creation, communication, and storage of history information easier compared to similar functionality in C++. During the level building portion of development, RSL was found to make writing the dynamic programming algorithm somewhat harder, but not *much* harder. As we discussed earlier, this neither supports nor refutes the utility of RSL history for

History is	*Creation*	*Communication*	*Storage*
Easier	Support	Support	Support
Same	Support	Support	Support
Harder			
Much Harder	Refute	Refute	Refute
than logging			

Table 5.1: Criteria to test the hypothesis in the development stage

History is	*Examination*	*Traversal*
Easier	Support	Support
Same		Refute
Harder	Refute	Refute
than logging		

Table 5.2: Criteria to test the hypothesis in the evaluation stage

evaluating decisions. For the development stage, we find that history is easier to obtain using RSL than it is using traditional logging methods, and the hypothesis is supported.

Table 5.2 restates the criteria for the evaluation stage. In the hand detection portion of the algorithm, interesting behavior was noted that was not mentioned in the original algorithm description [22]. This behavior was found using analysis that would be prohibitively difficult without the history tools provided by RSL. The level building portion of ASL recognition was less successful. A potential area of interest was noted, but it did not turn out to be a consistent or interesting outcome of the algorithm. Nevertheless, this examination was found to be easier than that allowed by traditional logging mechanisms. For the evaluation stage, we find that RSL

history provides information useful in evaluating the ASL recognition algorithm that would be harder to obtain using traditional logging. This is the assertion made in the hypothesis; therefore, the hypothesis is supported.

5.1.1 Contributions

We considered two subproblems: hand-detection and nested dynamic programming of an ASL recognition algorithm. For the hand-detection algorithm, a clear advantage in using RSL was demonstrated according to our evaluation criteria. For the dynamic programming algorithm, the advantage of using RSL for history storage and traversal is shown, but the utility is not.

Additionally, this research provides insight to the designers of RSL with regard to the usability of RSL history. As noted by Mernik et al. [17], ”DSL development is not a simple sequential process”, and each stage of language development may provide new insight or questions into previous stages. By implementing computer vision algorithms in RSL, this research becomes an integral step in the domain-specific language development process [17]. Specifically, the language is changed in the following ways at least in part through the feedback from this work: A language feature, annotations, and supporting functions are included in RSL as a result of observations made during this work. Minor compiler changes were made to allow recursive functions. RSL extensibility is demonstrated through the creation of a computer vision specific API. A design structure for RSL is proposed and implemented to create a clear distinction in responsibilities between RSL scripts and the called decision functions. Through replicating the ASL recognition algorithm,

several steps not called out in the original paper are made clearer, in addition to the identification and partial correction of errors in the dataset.

5.2 Future work

5.2.1 Future RSL experiments

An area that we would have liked to investigate had time permitted is the use of RSL history in a comparison between two algorithms used for ASL recognition. It may be valuable to correct any errors in our implementation and then find another ASL recognition that would allow such a comparison.

5.2.2 Possible changes to RSL

We found that, while not strictly necessary, a `not` keyword would be valuable for the `if` and `while` conditions. This would improve readability and simplify the interpretation layer, just by reducing slightly the amount of duplicate code.

It may be useful to add a map-like function to the reporting section that would apply a reporting function to a series of decision points. We found this pattern useful in our work, and it would be worth determining if that need comes up again.

Grouping by different interpretation fields was required for our evaluation of the dynamic programming algorithm. This grouping was accomplished by writing a series of predicates and a series of recursive RSL functions. It may be useful to provide a more natural way to express this in RSL directly. For example: `group observing alpha, level { ... }`.

While not a change to the language itself, it would be useful if the RSL compiler generated the IO code for interpretations and, ideally, history. We think the ability to run an algorithm and share the history of that run with other researchers would be a valuable feature.

5.2.3 Expansion of the ASL recognition algorithm implementation

Our implementation of the ASL recognition algorithm has several areas where improvement can be made:

- The accuracy of the algorithm could be improved. It is possible there are still bugs in the code that have not been found as well as the possibilities that the open questions in Section 4.3 hold the answers to improved correctness.
- This algorithm was implemented with no regard for speed. Data is copied regularly and unnecessarily. This was done in the spirit of making the code complete, then correct, then optimized, but the optimized stage was never reached. There is a lot of work here.
- Our inexperience with the computer vision domain may mean there are still valuable investigations to be made into the level building dynamic programming portion of the ASL recognition algorithm. It may be worth while to have a researcher with more computer vision knowledge take a look at the steps involved and the data available.

History	*Algorithm stage*		*Supports/Refutes*
	Hand Detection	*Dynamic Programming*	
Creation	Easier	Harder	*Undetermined*
Communication	Easier	Easier	yes
Storage	Easier	Easier	yes
Examination	Easier	Easier	yes
Traversal	Easier	Easier	yes

Table 5.3: Hypothesis evaluation criteria for each algorithm stage. Only *Creation* of history for the Dynamic Programming was not easier.

5.3 Summary

RSL history is extremely useful and informative for the hand detection portion of the ASL recognition algorithm. We were able to compare the success of each step of the algorithm using information that would have been prohibitively difficult to obtain using traditional logging methods. RSL history is less useful for the dynamic programming, and we think some examination of the types of algorithms for which RSL is applicable would be valuable. Table 5.3 summarizes the criteria we tested and the results. It is important to note again that, while we approached this work carefully, with clear criteria, the evaluation of a language is not a quantitative process, and other researchers may look at this and come to different conclusions.

While the results of the dynamic programming algorithm were not what we expected, the exercise provided insight into how RSL programs perform and offered opportunities to improve the language performance and understand how to create data structures using interpretations. For example, recursive grouping functions were developed while reporting on this algorithm. We find these results encour-

aging, and believe that applying RSL to more problems in the pattern recognition domain will allow further refinement of the language.

Appendices

Appendix A

RSL Implementation Languages

A.1 Use of Standard ML

The RSL compiler uses TXL [4] to create Standard ML (SML) [18] code which executes the strategy. This SML is compiled using the MLton compiler [19, 25]. There are several advantages to using SML as an implementation language. The syntax of ML is terse and expressive and well suited to the math-intensive nature of pattern recognition. Equation A.1 [22] describes the dynamic program used in the ASL recognition algorithm to select the best match for a series of frames, signs, and hand candidates.

$$Cost(i,j,k) = d(S^i_m, g_k(j)) + \min \begin{cases} \min\limits_{r, m(g_k(j), g_r(j-1)) \leq T_0} & Cost(i, j-1, r) \\ \min\limits_{r, m(g_k(j), g_r(j-1)) \leq T_0} & Cost(i-1, j-1, r) \\ Cost(i-1, j, k) & \end{cases} \tag{A.1}$$

The SML implementation of Figure A.1 in Listing A.1 closely follows, almost line-for-line, the mathematical expressions describing the dynamic program. Contrast this with the C++ implementation in Listing A.2. C++ provides a few functions in the standard library that allow application of certain kinds of functions to elements in a collection. However, it required non-standard libraries to adapt the

Listing A.1: ML implementation of Figure A.1

```
fun min vals = List.foldl Real.min Real.maxFinite vals

fun cost(i,j,k) =
  let
    val d = distance(i,j,k)
    val allHands = handCandidates(j, j - 1)
    val m1 = min (map (fn r => cost(i, j - 1, r)) allHands)
    val m2 = min (map (fn r => cost(i - 1, j - 1, r)) allHands)
    val m3 = cost(i - 1, j, k)
  in
    d + (min [m1, m2, m3])
  end
```

`cost(i,j,k)` function as needed, and target containers for the `std::transform(...)` call had to be manually allocated. It would be possible to implement the cost function in C++ using a more imperative style (e.g., using `for` loops), but this would take the implementation further from the style of the definition in Figure A.1. The base case of this recursive algorithm is not implemented in these listings in order to compare their similarity to the definition. However, ML stands out in the base case comparison as well because pattern matching in SML would allow the base case to be added without modifying the existing definition. The C++ implementation would require the use of conditional statements and deeper nesting of the algorithm.

SML's strong static typing and extensible type system allowed the implementation to use data structures that match those described in the algorithm definition. Pattern matching provided a powerful tool for taking those data structures apart in a type-safe way and ensured the various operations were performed on the

Listing A.2: C++ implementation of Figure A.1

```
#include <vector>
#include <algorithm>
#include <limits>
#include <boost/bind.hpp>

double min( const std::vector<double> &costs ) {
    if( costs.empty() )
        return std::numeric_limits<double>::max();
    return *( std::min_element( costs.begin(), costs.end() ) );
}

double cost( int i, int j, int k ) {
    double d = distance( i, j, k );
    std::vector<int> allHands = handCandidates( j, j - 1 );
    std::vector<double> m1costs;
    std::transform( allHands.begin(), allHands.end(),
                    std::back_inserter( m1costs ),
                    boost::bind( cost, i, j - 1, _1 ) );
    double m1 = min( m1costs );
    std::vector<double> m2costs;
    std::transform( allHands.begin(), allHands.end(),
                    std::back_inserter( m2costs ),
                    boost::bind( cost, i - 1, j - 1, _1 ) );
    double m2 = min( m2costs );
    double m3 = cost( i - 1, j, k );
    double allCosts[3] = { m1, m2, m3 };
    return d + *(std::min_element( allCosts, allCosts + 3 ));
}
```

correct data. The usefulness of the SML type system was readily apparent during this work when contrasted with the C/C++ language functions used from the OpenCV [3] library. OpenCV operations usually accept a matrix structure (Mat) that contains a type code indicating the type of data stored in the matrix. This type code is checked at runtime and causes program failure when the wrong matrix type is provided to a function. In Listing A.3, the function `getEdges(...)` accepts

an OpenCV matrix type for edge detection and possible dilation. The OpenCV function used for edge detection, `cv::Canny(...)`, demands an 8-bit, single channel (gray scale) image. However, the `cv::Mat` type is used to represent all image types in OpenCV, and the type of an instance is determined by reading an integer type code at runtime. OpenCV's runtime type-checking model required frequent testing of the C/C++ code to verify that the correct data type was being passed to the functions. On the other hand, SML functions never needed testing to verify that the correct data-type was provided because this was checked at compile time. If the code compiled, the function had been given the correct data type.

Listing A.3: C++ edge detection helper functions

```
cv::Mat getEdges( cv::Mat toEdge ) {
    cv::Mat edges = cv::Mat::zeros( toEdge.size(),
                                    toEdge.type() );
    cv::Canny( toEdge, edges, 200, 255 );
    return edges;
}
```

This did not eliminate the need for testing, but it did eliminate a common source of bugs. This research highlighted that type errors are especially common when using an unfamiliar library where misunderstandings are easy or documentation is not complete, and when passing data between languages with slightly different type representations. Early detection of these errors made use of SML a boon to this research and, likely, to RSL in general.

A.2 Calling C from SML

Several libraries used in this research were written in C or C++, and, as a result, so was much of the implementation. However, since RSL is compiled to SML [18], which is then compiled by the MLton compiler [19, 25], the implementation is driven by SML code. This required passing data and commands from ML to C/C++ via MLton's foreign function interface (FFI). MLton's web site describes the FFI in detail.

MLton provides a C header file defining the types that can be passed to C functions. These include the basic numeric types of various sizes such as sixteen bit integers or sixty-four bit floating point values. Additionally, MLton provides a `Pointer` type that allows arrays or vectors to be provided to the C functions. When passing any array from ML to C, the size must be included somehow. This can be done as a separate parameter to the function, by adding the length to the beginning of the array if the types are compatible, or by including some terminating value at the end of the array. This rule applies to strings as well as arrays since MLton's strings are not null terminated.

Listing A.4 shows C++ code to make two functions available to ML. Both `showImageC` and `saveImageC` take a vector of image data and a width and height that can be used by the C++ code to calculate the vector size. Since all types of array are passed from ML to C++ as the `Pointer` type, it is necessary to cast to the correct data type in the C++ code. In Listing A.4, this is made possible by a type code that the ML code in Listing A.5 can use to indicate if integers, characters, or reals are being passed in the vector. In addition to the image data, `saveImageC`

takes a filename parameter and a length that allows a string.

Listing A.4: C++ code to show or save an image

```
#include "export.h"
#include "ml-types.h"

extern "C" {
    void showImageC( Pointer img, int width, int height, int
       type );
    void saveImageC( Pointer img, int width, int height, int
       type, char *fname, int fnameLen);
}
```

Listing A.5: ML code to import and call C++

```
val showImage = _import "showImageC" : char vector * int * int
    * int -> unit;
val saveImage = _import "saveImageC" : char vector * int * int
    * int * char vector * int -> unit;

showImage( imageVector, width, height, dataTypeCode );
saveImage( imageVector, width, height, dataTypeCode, fileName,
    (size fileName))
```

It is through the `Pointer` type that information is passed back to ML from C code. ML must provide to the C function a reference to the allocated storage for the data. This storage can be a single value, such as an integer, or a collection, such as an array of color values for an image. In Listing A.6, two functions are provided. `getFrameInfoC` provides information about the data type of the image data and the amount of data to be returned. The C++ code puts the width, height, and type information into the ML allocated memory referenced by the `Pointers`. ML uses

this information to allocate storage for the image data, as seen in Listing A.7, and then `getFrameC` is used to pass the image data.

Listing A.6: C++ code to return an image to ML

```
#include "export.h"
#include "ml-types.h"

extern "C" {
    void getFrameInfoC( Pointer width, Pointer height, Pointer
        dtype );
    void getFrameC( Pointer img );
}

#endif
```

MLton's FFI allowed for transitions between C++ and ML. Representing the interface to C++ code as C is a fairly common requirement whenever writing C++ because so many available libraries are written in C.

Listing A.7: ML code to retrieve an image from C++

```
val getFrameInfoC = _import "getFrameInfoC" : int ref * int ref
    * int ref -> unit;
val getFrameC     = _import "getFrameC" : char array -> unit;

fun getFrameInfo (): int * int * int * int =
  let val width = ref 0
      val height = ref 0
      val dt = ref 0
      val _ = getFrameInfoC( width, height, dt )
  in (!width, !height, !dt) end;

fun getImage (): char vector * int * int * int =
  let val (width, height, dt) = getFrameInfo()
      val img : char array = Array.array(dt * width * height,
          Char.chr( 0 ) )
      val _ = getFrameC( i, t, img )
  in (Array.vector( img ), width, height, dt) end;
```

Appendix B

RSL source listings

This appendix contains the complete RSL source for all the RSL scripts referred to in this document. Other listings were modified to highlight the code relevant to the section.

Listing B.1: Hand detection script

```
inc "detect.mlb"
inc "hd-decision.sml"
inc "hd-interp.sml"

notes:
    FrameDiffs : real vector
    FrameId: int

interp:
    srcDir : string
    frameId : int
    keyframe : bool
    truehand : int vector
    frame : char vector * int * int * int
    gray : char vector * int * int * int
    skin : char vector * int * int * int
    handImage : char vector * int * int * int

fn fShowFrames() {
    print "Displaying frames (press any key to advance)"
    print all: sNumFramesMsg
    write all observing frame to "/dev/null": sDisplayFrames
}

fn fShowDifferences() {
    print "Displaying difference images (press any key to
       advance)"
```

```
    print all: sNumFramesMsg
    write all observing handImage to "/dev/null":
        sDisplayDifferences
}

fn fShowUnique() {
    (* Identify unique frames and discard, keeping current
        interp. set *)
    duplicate
        {
            [ Unique ] update: uniqueDiffImage
            print "Unique frames:"
            print all: sNumFramesMsg

            reject
        }
        {
            accept
        }
}

fn getHandContours() {

    (* Section 4.1, step 2.a *)
    [DiffImage] update all handImage
                observing keyframe, frameId, gray:
                    initialDiffImages

    (* Section 4.1, step 2.b part 1 *)
    [SkinmaskDiff] update handImage
                   observing frameId, skin: skinmaskDiffs

    (* Section 4.1, steps c and d *)
    [EdgeAndMask] update handImage
                  observing frameId: edgeAndMaskDiffs

    (* Section 4.1, step e *)
    [RemoveSmallComponents] update handImage
                            observing frameId:
                                removeSmallComponents

    (* Section 4.1, step f *)
    [BoundaryImage] update handImage
                    observing frameId: extractBoundary
}
```

```
fn main(test, train, minMaxPixelComponentSize) {
    t1 = valOf(Int.fromString(minMaxPixelComponentSize))
    (*
    print "Input file:"
    print test
    print "\nT1:"
    print minMaxPixelComponentSize
    print "\n"
    *)

    (* Create initial interpretation, attach directory for
       frame images *)
    munge: initInterp
    update srcDir: loadDir( test, train )

    [GetFrames] update frameId, frame, truehand:
       getFramesImages(test)
    (* print "Loading images\n" *)

    (* Get the skin, grayscale, and key frames used by other
       steps *)
    [GetSkinMask] update skin observing frameId: skinMasks
    [GetGrayScale] update gray observing frameId: grayScales
    [FindKeyFrames] update all keyframe observing frameId:
       keyframes(t1)
    if observing keyframe bIsKeyFrame {
        print "Keyframe count: "
        print all: sNumFramesMsg
    }

    getHandContours()

}

hfn report() {
    reject

    (*
    diffAccuracy("DiffImage")
    diffAccuracy("SkinmaskDiff")
    diffAccuracy("EdgeAndMask")
    diffAccuracy("RemoveSmallComponents")
    diffAccuracy("BoundaryImage")
    *)
```

```
    (*
    hadd ["DiffImage"]
    print "\n\nStep 2a\n"
    print: sPrintCenterAccuracy
    reject

    hadd ["SkinmaskDiff"]
    print "\n\nStep 2b\n"
    print: sPrintCenterAccuracy
    reject

    hadd ["EdgeAndMask"]
    print "\n\nStep 2c and 2d\n"
    print: sPrintCenterAccuracy
    reject

    hadd ["RemoveSmallComponents"]
    print "\n\nStep 2e\n"
    print: sPrintCenterAccuracy
    reject

    hadd ["BoundaryImage"]
    print "\n\nStep 2f\n"
    print: sPrintCenterAccuracy
    reject
    *)

    hadd ["FindKeyFrames"]
    hprint all: sHprintDiffs
    reject

}
```

Listing B.2: Scoring script

```
inc "aslalg.mlb"
inc "aslalg.sml"
inc "storeScores-interp.sml"
inc "levelbuilding-external.sml"

(*
  The interpretation is an interval (start frame to end frame)
      for a word
```

```
  with the distance for that word and the score of a
     predecessor. The
  total distance of the sentence is predecessor distance +
     interval distance
*)
interp:
    testFrames: int vector
    level : int
    word : int
    score : real
    interval : int * int
    prevs : int list * real

(* Reject any sequences that are illegal *)
fn makeLevel( itemMap ) {
    [NextLevel] munge: levelUpMunge(itemMap)
    [InternalLevelCheck] if all observing level atMax {
        print "Scoring "
        print all observing level: len
        [ScoreLevel] update score observing interval, word:
           scoreLevel(0.0, itemMap)
    }
}

fn finish( destFile ) {
    print all: dumpScores(destFile)
}

fn levelAndScore( itemMap ) {
    print all observing level: prlevel
    print all observing level: len
    [OnlyHighestLevel] if all observing level atMax {
        makeLevel( itemMap )
    }
}

fn main ( testDir ) {
    test = testDir
    train = "/home/secret/USF-ASL-Data-Set-v2"
    t1 = 300
    (*ML*)
        val _ = aslalgLoad t1 train test
        val itemMap = itemIndexMap()
        val dumpFile = (testDir ^ "/allMahalanobisScore." ^ ".
           scores")
```

```
    (*ML*)

    [Init] munge: init
    [LoadVideo] update testFrames: getIds
    print "Trained and loaded\n"

    [LevelZero] update level, word, interval, score, prevs
                observing testFrames: levelZero(itemMap)
    print "Level zero intervals created\n"

    levelAndScore( itemMap )
    print "Done scoring\n"

    finish( dumpFile )

    (*
    [SignIntervals] if all observing interval
                           truthInterval {
        accept
    }
    else {
        reject
    }

    [MadeTheCut] accept
    *)
}

hfn report( testDir ) {
    (*ML*)
        val itemMap = itemIndexMap()
    (*ML*)
    (*
    reject

    print "History\n"

    hadd["MadeTheCut"]
    write observing word, interval, score to outFile:
       writeInterp(itemMap, testDir)
    *)
}
```

Listing B.3: Level building dynamic programming script

```
inc "aslalg.mlb"
inc "aslalg.sml"
inc "levelbuilding-external.sml"
inc "lb-report.sml"

(*
  The interpretation is an interval (start frame to end frame)
     for a word
  with the distance for that word and the score of a
     predecessor. The
  total distance of the sentence is predecessor distance +
     interval distance
*)
interp:
    testFrames: int vector
    level : int
    word : int
    score : real
    interval : int * int
    prevs : (int * int * int) list * real

fn makeLevel(alpha, itemMap, grammar) {
    [NextLevel] munge: levelUpMunge(itemMap)
    print all observing level: prlevelAt( "NextLevel" )
    (* LevelUp just created a set of interps at new highest
        level, so
       only look at those *)
    [InternalLevelCheck] if all observing level atMax {
        [ScoreLevel] update score observing interval, word:
            scoreLevel(alpha, itemMap)
        print all observing level: prlevelAt( "ScoreLevel" )
    }

    [GetPrevs] update all prevs
               observing score, interval, level, word:
                   updatePrevs( itemMap, grammar)
    print all observing level: prlevelAt( "GetPrevs" )

    [LevelCheckForTrim] if all observing level atMax {
        [TrimToBest] if all observing word, score, interval,
            prevs notBest {
             print all observing level: prlevelAt( "ToTrim" )
             reject
        }
        print all observing level: prlevelAt( "UnTrimmed" )
```

```
    }
}

fn finish(itemMap, grammar) {
    [GetAtEnd] if observing interval, testFrames atEnd {
        [AddEnd] update word, prevs, score, interval: addEnd(
           itemMap)
    }
    else {
        reject
    }
    [KillBadGrammar] if observing word, prevs badGrammar(
       itemMap, grammar) {
        reject
    }
}

fn levelbuildingLoop( alpha, numLevels, itemMap, grammar ) {
    [WhileLevel] while all observing level belowMaxLevel(
       numLevels ) {
        print all observing level: prlevel
        [OnlyHighestLevel] if all observing level atMax {
            makeLevel(alpha, itemMap, grammar)
            (* makeLevel() made a new, higher level. Drop the
               old one *)
            [DropOldLevels] if all observing level, interval,
               testFrames
                            oldIncompleteLevel {
                print all observing level: prlevelAt( "
                   DroppingOld" )
                reject
            }
        }
        [LevelEnd] accept
    }
}

fn main ( alphaStr, levels, testDir ) {
    test = testDir
    train = "/home/secret/USF-ASL-Data-Set-v2"
    t1 = 300
    (*ML*)
        val alpha = valOf(Real.fromString alphaStr)
        val numLevels = valOf(Int.fromString(levels))
        val grammar = aslalgLoad t1 train test
```

```
        val itemMap = itemIndexMap()
        val scoreFile = (testDir ^ "/allMahalanobisScore." ^ ".
           scores")
        val str = loadScores( scoreFile )
        val _ = print str
    (*ML*)

    [Init] munge: init
    [LoadVideo] update testFrames: getIds
    print "Trained and loaded\n"

    [LevelZero] update level, word, interval, score, prevs
                observing testFrames: levelZero(itemMap)

    levelbuildingLoop( alpha, numLevels, itemMap, grammar )

    print "Done leveling\n"

    finish( itemMap, grammar )
    [Finished] accept

    print: i2s(itemMap)

    [GetBest] if all observing score notBestScore {
        reject
    }

    [OnlyBest] accept
    print "The best interpretation:\n"
    print: i2s(itemMap)

    print "DONE DONE DONE\n"
}

hfn report ( alphaStr, levels, testDir ) {
    test = testDir
    train = "/home/secret/USF-ASL-Data-Set-v2"
    t1 = 300
    (*ML*)
        val alpha = valOf(Real.fromString alphaStr)
        val numLevels = valOf(Int.fromString(levels))
        val grammar = aslalgLoad t1 train test
        val ifilename = testDir ^ "/levelbuilding-" ^ alphaStr
           ^ "-ifile"
    (*ML*)
```

```
    (* print: sPrintLevenshteinDistance(itemMap, testDir) *)
    reject
    hadd ["LevelEnd"]
    hadd ["Finished"]
    write ifile to ifilename
}
```

Listing B.4: Script to combine interpretation files with truth

```
inc "itc-report.sml"

interp:
    file: string
    alpha : real
    truth : (string * int * int) list
    range : int * int
    level : int
    word : string
    score : real
    interval : int * int
    prevs : (string * int * int) list * real

fn main( outfile ) {
    (* ML *)
        val infile01 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
           Sentence-1.4-lipread-can-i/Full-Sentence/
           s5c1lipreadcani/levelbuilding-0.16-ifile"
        val infile02 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
           Sentence-1.4-lipread-can-i/Full-Sentence/
           s5c1lipreadcani/levelbuilding-0.28-ifile"
        val infile03 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
           Sentence-1.4-lipread-can-i/Full-Sentence/
           s5c1lipreadcani/levelbuilding-0.30-ifile"
        val infile04 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
           Sentence-2.4-lipread-cannot-i/Full-Sentence/
           s5c2lipreadcannoti/levelbuilding-0.16-ifile"
        val infile05 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
           Sentence-2.4-lipread-cannot-i/Full-Sentence/
           s5c2lipreadcannoti/levelbuilding-0.28-ifile"
        val infile06 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
           Sentence-2.4-lipread-cannot-i/Full-Sentence/
           s5c2lipreadcannoti/levelbuilding-0.30-ifile"
```

```
    val infile07 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
        Sentence-3.4-i-understand-not/Full-Sentence/
        s5c3youunderstand/levelbuilding-0.16-ifile"
    val infile08 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
        Sentence-3.4-i-understand-not/Full-Sentence/
        s5c3youunderstand/levelbuilding-0.28-ifile"
    val infile09 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
        Sentence-3.4-i-understand-not/Full-Sentence/
        s5c3youunderstand/levelbuilding-0.30-ifile"
    val infile10 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
        Sentence-7.4-dontknow-i/Full-Sentence/s5c7idontknow/
        levelbuilding-0.16-ifile"
    val infile11 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
        Sentence-7.4-dontknow-i/Full-Sentence/s5c7idontknow/
        levelbuilding-0.28-ifile"
    val infile12 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
        Sentence-7.4-dontknow-i/Full-Sentence/s5c7idontknow/
        levelbuilding-0.30-ifile"
    val infile13 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
        Sentence-10.4-i-need-that-i/Full-Sentence/
        s5c10ineedthatone/levelbuilding-0.16-ifile"
    val infile14 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
        Sentence-10.4-i-need-that-i/Full-Sentence/
        s5c10ineedthatone/levelbuilding-0.28-ifile"
    val infile15 = "/mnt/raid/secret/USF-ASL-Data-Set-v2/
        Sentence-10.4-i-need-that-i/Full-Sentence/
        s5c10ineedthatone/levelbuilding-0.30-ifile"
(* ML *)
add infile01
if observing file nofile {
    update alpha, file, truth, range: loadTruth(infile01)
}
add infile02
if observing file nofile {
    update alpha, file, truth, range: loadTruth(infile02)
}
add infile03
if observing file nofile {
    update alpha, file, truth, range: loadTruth(infile03)
}
add infile04
if observing file nofile {
    update alpha, file, truth, range: loadTruth(infile04)
}
add infile05
```

```
    if observing file nofile {
        update alpha, file, truth, range: loadTruth(infile05)
    }
    add infile06
    if observing file nofile {
        update alpha, file, truth, range: loadTruth(infile06)
    }
    add infile07
    if observing file nofile {
        update alpha, file, truth, range: loadTruth(infile07)
    }
    add infile08
    if observing file nofile {
        update alpha, file, truth, range: loadTruth(infile08)
    }
    add infile09
    if observing file nofile {
        update alpha, file, truth, range: loadTruth(infile09)
    }
    add infile10
    if observing file nofile {
        update alpha, file, truth, range: loadTruth(infile10)
    }
    add infile11
    if observing file nofile {
        update alpha, file, truth, range: loadTruth(infile11)
    }
    add infile12
    if observing file nofile {
        update alpha, file, truth, range: loadTruth(infile12)
    }
    add infile13
    if observing file nofile {
        update alpha, file, truth, range: loadTruth(infile13)
    }
    add infile14
    if observing file nofile {
        update alpha, file, truth, range: loadTruth(infile14)
    }
    add infile15
    if observing file nofile {
        update alpha, file, truth, range: loadTruth(infile15)
    }
    write ifile to outfile
}
```

Listing B.5: Report generating script

```
inc "ir-interp.sml"
inc "ir-report.sml"

interp:
    wordMap : (string * int) list
    alpha : real
    truth : (string * int * int) list
    range : int * int
    level : int
    word : string
    score : real
    interval : int * int
    prevs : (string * int * int) list * real
    editDistance: int
    editError: real

(*———————————————————Display Helpers
   ————————————————————————————————————*)
fn countAtAlpha( a ) {
    if observing alpha atAlpha(a) {
        print all: prForAlpha(a)
        print all: prCount
    }
}

fn threeCount() {
    countAtAlpha( 0.16 )
    countAtAlpha( 0.28 )
    countAtAlpha( 0.30 )
}

fn statsAtAlpha( a ) {
    if observing alpha atAlpha(a) {
        print all: prForAlpha(a)
        print "\n"
        print all observing editError: prMinError
        print all observing editError: prMaxError
        print all observing editError: prAvgError
    }
}
```

```
fn threeStats() {
    statsAtAlpha( 0.16 )
    statsAtAlpha( 0.28 )
    statsAtAlpha( 0.30 )
}

fn bests() {
    if all observing score bestScore {
        print "\tBest score\n"
        print: interpToString
        print "\n"
    }
    if all observing editError bestError {
        print "\tBest edit distance\n"
        print: interpToString
        print "\n\n"
    }
}

fn truthSoFarCount() {
    if isTruthSoFar {
        print "Truth so far "
        print all: prCount
    }
}
(*
   ----------------------------------------------------------------------------
   *)

(*------------------Recursive grouping functions
   --------------------------*)
fn eachGroupStats( outfile ) {
    if all observing truth, alpha, level isNextGroup {
        (*
        print all observing truth, alpha, level: prOneGroup
        print all: prCount
        truthSoFarCount()
        bests()
        *)
        write ++ all to outfile: prGroupReport
    }
    else {
        eachGroupStats( outfile )
    }
}
```

```
(*
  ----------------------------------------------------------------------------
  *)

fn writeReportLine( outfile ) {
    write "Alpha\t Truth String\t Level\t Max Error\t Min Error
        \t Avg Error\t NumTruth\t NumInterps\t pctTruth\t
        Average Truth Error\t Average non-Truth Error\t
        fullMatch\n" to outfile
}

fn main( interpfile ) {
    (* ML *)
        oneSix = "cvsl_out/report-0.16-log"
        twoEight = "cvsl_out/report-0.28-log"
        threeOh = "cvsl_out/report-0.30-log"
    (* ML *)

    add interpfile
    update all wordMap observing truth, word, prevs:
        makewordmap
    update editDistance observing wordMap, truth, word, prevs,
        range, interval : levenshteinDistance
    [Error] update editError observing wordMap, truth, word,
        prevs, range, interval: levenshteinError

    (*
    if observing word atEnd {
        if isTruth {
            print "\n\nTruth:\n"
            threeCount()
            threeStats()
            eachGroupStats()
        }
        else {
            print "\n\n\nFailed:\n"
            threeCount()
            threeStats()
            eachGroupStats()
        }
    }
    *)

    if observing alpha atAlpha( 0.16 ) {
```

```
        writeReportLine( oneSix )
        eachGroupStats( oneSix )
    }
    if observing alpha atAlpha( 0.28 ) {
        writeReportLine( twoEight )
        eachGroupStats( twoEight )
    }
    if observing alpha atAlpha( 0.30 ) {
        writeReportLine( threeOh )
        eachGroupStats( threeOh )
    }

}
```

Bibliography

[1] Dana H Ballard and Christopher M Brown. Computer vision, 1982.

[2] Jon Bentley. *Handbook of Programming Languages Vol. III: Little Languages and Tools*, chapter Little Languages. Macmillan Technical Publishing, 1998.

[3] Gary Bradski and Adrian Kaehler. *Learning OpenCV*. O'Reilly Media Inc., 2008. URL `http://oreilly.com/catalog/9780596516130`.

[4] James R. Cordy. Txl - a language for programming language tools and applications. *Electron. Notes Theor. Comput. Sci.*, 110:3–31, 2004.

[5] Nathaniel Duca, Krzysztof Niski, Jonathan Bilodeau, Matthew Bolitho, Yuan Chen, and Jonathan Cohen. A relational debugging engine for the graphics pipeline. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 453–463, New York, NY, USA, 2005. ACM.

[6] Marc Eaddy, Alfred Aho, Weiping Hu, Paddy McDonald, and Julian Burger. Debugging aspect-enabled programs. In *SC'07: Proceedings of the 6th international conference on Software composition*, pages 200–215, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] Conal Elliot. An embedded modeling language approach to interactive 3d and multimedia animation. *IEEE Transactions on Software Engineering*, 1999.

[8] Matthew Fluet, Benjamin Holm, and Richard Zanibbi. Programmer's guide to the recognition strategy language (rsl) (version 2.0). 2011.

[9] Rafael C. Gonzalez, Richard E. Woods, and Steven L. Eddins. *Digital Image Processing Using MATLAB, 2nd ed.* Gatesmark Publishing, 2nd edition, 2009.

[10] John C. Handley. Table analysis for multiline cell identification. volume 4307, pages 34–43. SPIE, 2000.

[11] Jianying Hu, Ramanujan S. Kashi, Daniel P. Lopresti, and Gordon Wilfong. Table structure recognition and its evaluation. volume 4307, pages 44–55. SPIE, 2000. URL `http://link.aip.org/link/?PSI/4307/44/1`.

[12] Paul Hudak. *Handbook of Programming Languages Vol. III: Little Languages and Tools*, chapter Domain-Specific Languages. Macmillan Technical Publishing, 1998.

[13] D. Kelly, J. McDonald, and C. Markham. Continuous recognition of motion based gestures in sign language. In *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, pages 1073 –1080, sept. 2009.

[14] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming and modular reasoning. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, page 313, New York, NY, USA, 2001. ACM. ISBN 1-58113-390-1.

[15] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 304–317, New York, NY, USA, 1997. ACM. ISBN 0-89791-908-4.

[16] B. L. Loeding, S. Sarkar, A. Parashar, and A. I. Karshmer. Progress in automated computer recognition of sign language. In *Computers Helping People with Special Needs*, volume 3118/2004, 2004.

[17] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[18] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0-262-63181-4.

[19] mlton. MLton. `http://www.mlton.org`.

[20] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 535–552, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5.

[21] Lawrence Rabiner and Biing-Hwang Juang. *Fundamentals of speech recog-*

nition. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-015157-2.

[22] Barbara Loeding Ruiduo Yang, Sudeep Sarkar. Handling movement epenthesis and hand segmentation ambiguities in continuous sign language recognition using nested dynamic programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(3), 2010.

[23] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 481–497, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4.

[24] I.R. Vega and S. Sarkar. Statistical motion model based on the change of feature relationships: human gait-based recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(10):1323 – 1328, 2003. ISSN 0162-8828.

[25] Stephen Weeks. Whole-program compilation in MLton. In *ML'06: Proceedings of the 2006 ACM SIGPLAN Workshop on ML*, pages 1–1. ACM, 2006.

[26] Richard Zanibbi, Dorothea Blostein, and James R. Cordy. Decision-based specification and comparison of table recognition algorithms. In S. Marinai and H. Fujisawa, editors, *Machine Learning in Document Analysis and Recognition*, volume 90, pages 71–103. Springer, 2008.

[27] Richard Zanibbi, Dorothea Blostein, and James R. Cordy. White-box evaluation of computer vision algorithms through explicit decision-making. In *ICVS*, pages 295–304, 2009.