ScratchTalk and Social Computation: Towards a natural language scripting model

Ian Eslick
MIT Media Lab
20 Ames St. E15-320R
Cambridge, MA 02139 USA
eslick@media.mit.edu

ABSTRACT
Natural Language interfaces have been proposed as a solution to make application functionality more accessible to end users. Practical implementation of such systems has been hampered by both semantic and knowledge-engineering challenges. This paper introduces Social Computation, a theoretical model targeting both challenges. The model employs natural language in a planning framework to enable communities of end-users to collaborate, explicitly and implicitly, in the development of new functionality. The model intrinsically supports the acquisition of user goals and natural language semantics in the context of the application domain. ScratchTalk is a prototype of the natural language planning component of the model applied to the Scratch programming environment. This serves to ground the framework and provide insights into the space of applications likely to be addressable by the Social Computation model.

Author Keywords
Natural Language, Goal Oriented User Interfaces, Collective Intelligence, Social Networking

ACM Classification Keywords
H5.2. User Interfaces

INTRODUCTION

Today, customization of application behavior is accomplished through combinations of scripting languages, plugins, and macros. Unfortunately, the learning curve necessary to use any of these approaches is too steep for most users. This problem stems, in part, from the cognitive barriers that all of these models require of users: understanding precise semantics, modeling hidden state, and navigating a large design space [4].

User interface research on end user programming aims to reduce the cognitive burden on end users while improving their access to the capabilities of the underlying system. As the tasks carried out by the user grow in complexity, the user must discover and memorize complex sequences of actions and understand how to avoid negative effects. End-user scripting attempts to enable the user to abstract and automate those sequences to save both labor and reduce the cognitive burden . It is this process of abstraction, the means by which a user codifies combinations of primitives, that is the primary interest of this paper.

New functionality can be created by three classes of people: a developer, another member of the user community, or the end user. The next section describes the properties of these three classes and where features come from, arguing that today's scripting models are fundamentally too expensive and insufficient to cover a large space of functionality. Increased coverage of potential functionality can increase the desirability or utility of a given application.

Social Computation aims to change the cost-economics of scripting by dramatically altering the user interaction model and the representation of procedural behavior. This model borrows concepts from goal-oriented programming, planning, interpreted languages, collective intelligence, and social networks.

SOCIAL COMPUTATION

What users can't do

Program design requires that a programmer maintain a mental dictionary of possible steps and determine how to sequence those steps so as to bring about desired side effects while avoiding undesirable ones. Programmers have to accommodate environmental variability, model hidden state that is not easily inspected, and express each step with perfect precision; all of which increase cognitive burden. When completed, these programs are brittle and correspondingly difficult to modify.

These skills are not easily learned. Competence requires significant training and experience, limiting the populations that can contribute to the development of new functionality. Attempts to ameliorate this situation in commercially available languages have had little success [8].

Solutions to this conundrum typically trade expressive power for simplicity so that users are better able to model the implications of a statement. For example, visual programming using flowcharts make application dataflow explicit, yet exclude message passing and other forms of interdependence. These models usually fall short of addressing the space of user goals, leading to frustration and low adoption rates.

This outlook is not promising. The very notion of programming appears anathema to most end-users. However, humans can express their desires to experts and through a process of iteration and clarification, experts are generally able to satisfy user goals. The following section proposes that as-

pects of this interaction can be captured in a computational framework, facilitating a larger range of end-user goals.

What users can do

The research community should be asking what users are naturally good at and looking for opportunities to leverage those skills. Following is a list of some natural user skills:

- Asking. Word of mouth has always played a significant role in problem solving. The Internet today extends that natural process via blogs, mailing lists and other discussion forums. For the non-expert, this is often the only source of answers.
- Searching. Most users of the internet today are accustomed to searching for answers to their problem via keywords or browsing. This can lead to the resources above, but can also be used with a program's help system (which are rarely helpful) or as an index to program functions.
- Inspecting. In an environment where the behaviors or side effects are transparent, users can be quite good at recognizing errors. This is true in a domain such as animation domain, but is less true in the spreadsheet domain where hidden state increases dramatically with increasing complexity [9].
- Critiquing. A review of online discussion lists for plugins, extensions and scripts indicate that there are many more users than developers and that users are extremely prolific in expressing their discontent with specific failures or specific missing features.
- Tweaking. A smaller set of users (although still larger than those who program) can take example scripts and perform minor changes when the relationships of the script to the domain model is reasonably easy to comprehend. (i.e. Mail filter rules).

Users are good are pattern matching, but bad at pattern specification: users can't always describe exactly what they want, but they know it when they see it. Users are bad at design but very good at complaining. How can we exploit these observations?

The Long-Tail of Software Features

The cost of developing a software feature is extremely high; thus developers are forced to pick the features that will have the greatest impact on sales and satisfying existing users. If we chose a standard indexing method, such as menus, for making features available, then the index is likely to be extremely underpopulated with respect to all reasonable user goals.

Anderson [1] coined the term "long tail" to describe a related commercial phenomenon wherein a change in cost economics made selling small-volume products highly profitable. The motivating observation was that Amazon.com had more total revenue from low-volume books than from high-volume books. This is because the cost of selling low and high volume books is identical for Amazon, whereas for a brick and mortar store a linear foot of shelf-space has a significant variable cost that has to be amortized over sales, requiring physical bookstores to only carry books that have high volume.

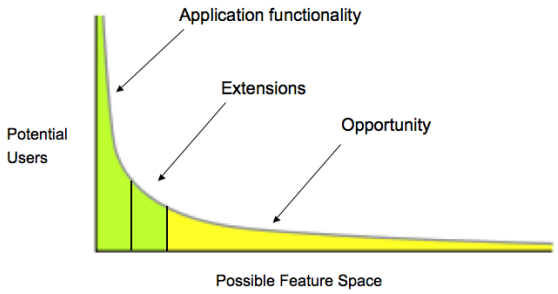

Figure 1. The long tail of software

The cost economics of application platforms today is akin to the brick and mortar bookstores of yesterday. As with Amazon, finding a way to serve the long tail of feature demand can dramatically improve the reach of software applications as well as the usefulness of those platforms to end users.

The obvious proposition is to find ways to reduce the costs of feature development. Open Source has proven to be a way to extend the reach of an application down the long tail of software features, but Open Source is limited by a different kind of economics. Even in Open Source, the cost of a feature remains high in terms of skilled labor, but using volunteer time instead of paid time. However volunteer developers are likely to release and maintain code when there is a sufficient incentive. Social incentives can be quite powerful, but tend to require a certain critical mass of peers or end users. Smaller open source projects are likely to have lower quality, limiting the population of users that can take advantage of them. So Open Source serves a modestly larger population than commercial software, mostly programmers and power-users, and is unable to effectively support or leverage the vast majority of users.

If the Open Source model doesn't significantly change the economics to capture the long tail of feature needs, what can?

Empowering End Users

The key idea behind the Social Computation framework is to find ways to enable the untapped end user population to contribute new features suited to their specific goals. This should be accomplished in a robust and sustainable way, while surmounting the problems of precision, hidden state, and design.

In this discussion features are considered to be pieces of code, or a description of a plan that can be interpreted, that is directly associated with a user's goal. Since user goals are often under specified, a plan or code fragment needs to cover a range of cases without users having to intervene directly.

We can characterize the user population with a similar curve as that applied to the software feature space. There are very

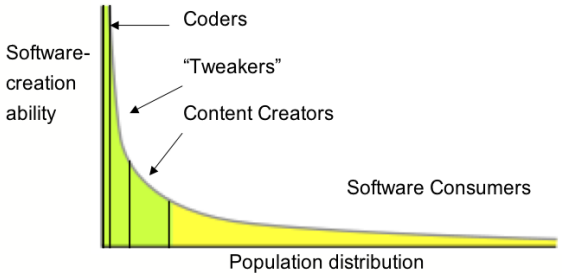

Figure 2. Distribution of user skills

few people who can write and maintain robust software. There are more people who can modify or add incrementally to existing code and an even larger group that can add various kinds of content like skins, templates, graphics, etc. However most of the population is incapable of directly providing value to other users. However, perhaps it is possible to do so indirectly.

A truly transformational impact on the cost-economics of new features becomes possible if can extract value from individual contributions in a way that enhances the next person's feature development. To make progress towards this end, I propose a key set of assumptions:

- Represent scripts as hierarchical plans consisting of goals and subgoals.
- Goals are specified by user-accessible natural language phrases that enable the system to map to subgoals.
- Plans execute in a "soft fail" environment; users are accustomed to failures but are able to repair failures in a multitude of ways.
- The user interface reifies side effects that the user is likely to care about.
- The user uses a restricted set of natural language or the user interface to critique plan failure; more sophisticated users can make suggestions and experts can directly manipulate the machinery used by less-sophisticated users.
- Plans are incrementally modified to fix the problem identified by the critique.

These assumptions eliminates two of the three barriers to end-user programming described earlier. Goals are not precisely described, but the related plan should provide increasing robustness to different environmental variation as well as handle the bookkeeping of hidden state that isn't expressed in the UI. To address the design problem we need to think creatively about the design problem itself.

Design requires an exploration of the space of possible data transformations, ordering and condition tests. This may happen in the mind of a developer, via a formal search algorithm, or via an exploratory implementation. The key assumption in Social Computing is that there is a community of users with similar goals.

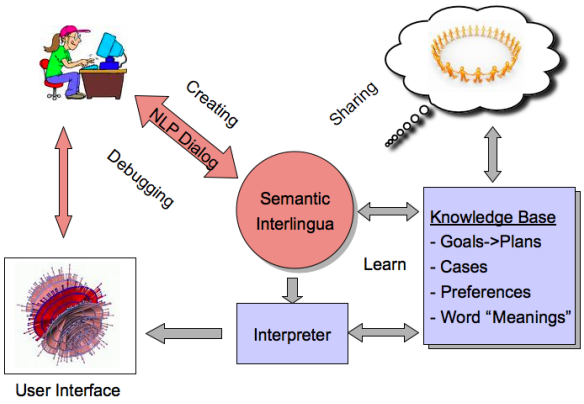

Figure 3. A Model for Social Computation

As shown in the graphic, there are two cycles taking place in any given user's application. The user is interacting with their system which uses a knowledge base to interpret a user speech act. The user observes the effect and makes any necessary corrections. Simultaneously, each statement made by the user that is correctly interpreted, is shared with other users who may be making the same statement.

With such a community there are a large set of techniques that can be used to sidestep many of the AI-hard problems that come to mind such as domain representation, planning, natural language understanding, and design space exploration. A sample of such techniques include:

1. **Collaborative filtering**. User critique provides immediate feedback on the reliability of a given goal interpretation. Even the most naive of users can say, "that didn't work" and then select an alternative version. Human to human interaction (such as an expert helping a neophyte) regarding a specific goal can provide direct, explicit feedback on plan success or failure.
2. **Sampling**. A new plan interpretation of a goal can be tried out on a small subset of users to investigate whether it improves reliability or acceptance rates in the same vein as the compiler techniques described in [3].
3. **Supervised structure learning**. The spotty reliability of a plan is an opportunity to learn about missing preconditions for a plan step. When the state variable is unrepresented by the system (i.e. no condition is found that is correlated to the plans reliability), users can be sampled to provide inputs as to why something works in one case and not another, hypothesizing a variable over which correlations can be observed.
4. **Collective mapping**. Users can specify a goal as a desired action or state and observe the resulting behavior. The user may change the goal, choose a different plan to fix a failed goal, or make an explicit correction to an existing plan. Each of these actions provides an implicit data point about whether a plan meets a goal, allowing users

to choose the preferred method even when there is not a clear model of state.

5. **Preference discovery**. Using the same data as in collective mapping above, the system can learn the order in which to present alternative interpretations to users.

6. **Preference clustering**. The behavior intended by a given natural language utterance may vary from user to user. When a user falls into a cluster of users with similar patterns of goal to plan selection or use of natural language terms, that user can share preference orderings with their affinity group rather than all users.

One of the largest advantages of this framework is the ability to view meaning as a mapping function and not a representational challenge. Most of the problems in traditional AI programs stem from the desire to explicitly represent the meaning of a goal or a state. Finding a plan becomes a search through a space defined by this formal representation, with all the convenient theoretical properties of correctness and complexity we are used to. Instead, interpretation is a process of finding a "near match" which *sufficiently* satisfies the intent of the user, allowing them to make forward progress towards a goal.

This model is likely to be grossly insufficient for large scale systems requiring complex software architecture, new user interface techniques, and new underling representations. However, these are the very elements that are common to many underlying features that developers should be putting their time and attention.

SCRATCHTALK

The ScratchTalk prototype emerged from an exploratory design and implementation intended to uncover the major design tradeoffs, representational issues, and user interface requirements involved in implementing the Social Computation model. At present, the system does not support a community model. The intent of this work was to investigate constraints on the natural language dialog and to identify specific mechanisms by which one user can create domain knowledge, plans, or language mappings that would be useful to other users.

ScratchTalk is an interface to the Scratch programming environment [6]. Scratch which was designed to be easy for children to learn. This simplicity emerges from its simple world state, small set of primitives, and a highly transparent graphical user interface. Scratch has many desirable properties:

- Fail-soft. Mistakes are not catastrophic
- Interactive. Mistakes are easy to fix.
- Visual. It easy to visualize state and behavior.
- Transparent. Behavior and state are not hidden.
- Simple. It is designed for non-programmer children.

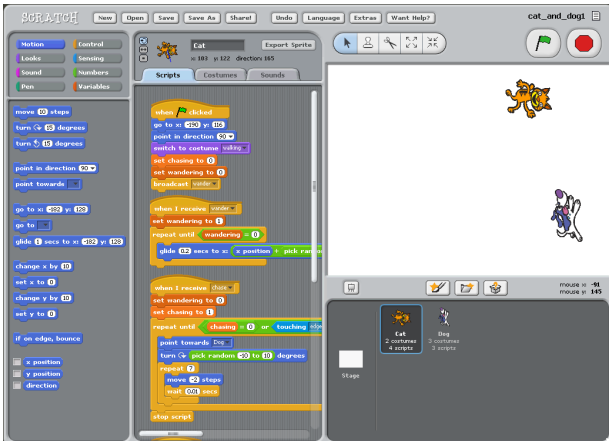

Figure 4. The Scratch User Interface

Scratch is a Logo-like world and user interface written in Squeak, a modern implementation of the Smalltalk language. The Scratch UI provides a visual, lego-block style view of Squeak language constructs which are used to control a set sprites that interact on a 2D canvas. The language consists of primitives for orienting, moving, and transforming sprites as well as detecting contact and other events. The language also provides a small set of conditional and iterative constructs to combine these primitives. There is no support for function calls or a call stack. The runtime model consists of a set of concurrently executing scripts, with asynchronous messaging and synchronized shared variables among them. Each script is initiated independently in response to a message, key command, mouse event, or a system-wide initiation event (greenflag).

EXAMPLE SCENARIO PART I

To simplify the potential space of dialog and the representation of both time and causality, I focus specifically on the using Scratch to create story-like animations.

The example animation used to drive the construction of the prototype is a simple "dog chases cat" scenario. The user expresses the steps of an animation in natural language, the natural language is converted into an intermediate representation, and the representation is in turn converted to a Scratch program that, when executed in the Scratch UI, the user can critique or extend.

A Simple Scenario

A ScratchTalk user might start an animation description with the first visible action:

```
"A cat wanders randomly
 around the screen."
```

In response, ScratchTalk instantiates a sprite called 'cat' and creates a 'wandering' event with an associated 'wander' action. A script that executes the wander action is synthesized and a program sent to the UI for execution. The user sees a cat making random movements in random directions on the

screen.

```
"When the cat comes near the dog,
 the dog chases the cat."
```

This next statement results in the system inferring that there is a new actor called 'dog' which waits for a 'near' event to occur (distance between dog and cat sprites is less than some value) and at that time starts a chase event. The animation now shows the cat wandering around the screen, if the cat comes near the dog, the dog will keep moving towards the cat while the cat wanders around.

The user finishes the initial scene with:

```
"If the dog catches the cat,
 the cat dies!"
```

This time, when the dog touches the cat during the animation, the cat stops wandering and performs a 'die' script. The default script in the knowledge base is to mimic the comic strip character Bill the Cat: stop moving and say "Ackthppht".

This simple example contains within it a surprising amount of complexity with implications for both program and language semantics.

Extending the Scenario

There are several problems with the initial program. When the dog chases the cat, the cat has no reciprocal response, as common sense would lead one to expect. The cat simply stops when he dies and should probably change its look to better capture the action.

The user can fix errors or add to the existing program as follows:

```
"When the dog chases the cat,
 the cat runs away."
```

This phrase triggers a template in the system for "when <X> , <Y>" that searches for an event matching <X>. In this case it first tries matching the phrase directly, assuming a surface variation of it was used to create the event. Failing this the system can look for near matches by identifying the verb class and looking for events that match the related classes, such as pursue in this case.

The "near" event that initiates the "chase" will now also initiate a new event, "runs away". This event is then linked to the other two events in a manner to be described in the following section.

Dialog and natural language processing in ScratchTalk benefit from the highly contextual nature of the interactions. The scope of the reference into the program representation is highly likely to be local, thus there are a finite number of possible matches <X>. Moreover, there is a reasonable belief from the conversational contract that <X> is sufficient to uniquely identify the event of interest.

A second kind of extension is modifying an existing action. In the case where the user has provided a sprite costume (or icon) that depicts the dead cat, the system will not automatically switch to this graphic.

```
"When the cat dies, switch to the
 'dead' costume."
```

Rather than creating a new event, this speech act will modify an existing event. The die action currently stops the runs away action and results in the cat saying "Ackthppht". The action for "switch" binds to the "cat" by default as no explicit subject is provided. While the exact mechanism is complicated[1], these two events (say and switch) can be combined by concatenation of the two events. ScratchTalk's representation of actions allows this to be done for any two events. The editing of actions is a central feature of the system and described in-depth in the section on planning below.

IMPLEMENTATION

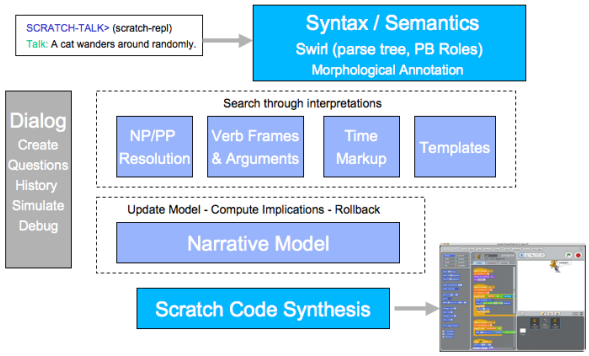

Figure 5. ScratchTalk Architecture

ScratchTalk is written in Common Lisp and communicates with a Scratch instance over a network socket. All of the logic described in this paper was implemented in the Lisp program. The socket simply allows the lisp program to upload programs to Scratch as well as the commands: start, stop and clear.

The core of ScratchTalk consists of natural language pipeline that operates roughly as follows:

1. Parse speech act => parse tree + VerbNet annotation
2. Compound sentence rules (i.e. when X then Y) => primitive subtrees
3. Dialog modeling => filters system commands, validates reference scope
4. Reference resolution (NPs, VPs, arguments) => SVOO style predicates

[1]Event merging is an optimization that occurs at program generation time so the full temporal structure of the graph is maintained during dialog with the user.

5. Goal resolution => choose a plan to implement goal in speech act from a library of plans
6. Modify program representation => insert, delete, reorder, or modify objects, properties, events, or actions according to chosen plan
7. Synthesize Scratch code and send to Scratch to be executed

In short, the system extracts goals from user speech acts, uses those to modify a high-level representation of the animation, then generates low level code to animate that representation.

Parsing and Reference Resolution

To quickly acquire a reasonably broad coverage of natural language input with which to perform these mappings, the SwiRL [11] semantic tagging framework was chosen. The system is simple to use and produces sufficiently robust results for the purposes of the prototype.

SwiRL uses Charniak's parser [2] to generate a parse tree, then annotates constituents with role labels such as actor, object, path, auxiliary arguments, etc. For simplicity, we do not adapt the system to develop new mappings for failed parses using user input. Instead we limit input to simple sentences on which SwiRL maps verbs in a meaningful way by asking the user to rephrase.

The system first applies a pattern matcher over the parse tree to handle a variety of compound statements indicating conditionality and conjunctions or disjunctions such as and, or, if, while, when, etc. A matching template drives the rest of the resolution process and performs any coordinate between the constituents, such as creating temporal, causal or conditional links between events in the animation narrative. For example:

```
(define-linear-template if-then
    (or (if $cond |,| then $consequent)
(if $cond |,| $consequent)
(if $cond then $consequent))
   ((link initiates
   (primary-event
    (resolve-event $cond))
   (primary-event
    (resolve-event $consequent)))))
```

Once the system has successfully identified a primitive sentence, it translates the argument phrases to objects in the system or to action modifiers. It then looks up action that matches the VerbNet type[2] as well as the arguments. Soft matching allows the system to degrade gracefully by dropping unknown or unneeded modifiers, looking for related VerbNet verbs that do have matching actions.

```
(defaction run ($agent $away)
  (do-until ()
    (progn
      (do-repeat 10
((forward 2)
 (wait 0.02)))))
  (:constraints (type-of $agent actor)
(member $away (away))))

(defaction run ($agent $away $np)
  (run-away $agent $np)
  (:constraints (type-of $agent actor)
(type-of $np actor)
(member $away (away))))
```

For example, the second definition of run above requires that the $np argument resolves to an actor object.

Representing Scratch Animations

ScratchTalk uses an intermediate representation to capture the essential phenomenology users are expected to reference. This representation is compiled into a concrete Scratch program which can then be run within the Scratch UI. For example the Scratch script below, taken from the cat and dog example, is initiated by an asynchronous message, "wander" and causes the associated sprite to perform a random-walk towards the bottom right corner of the screen.

```
((when-receive "wander")
 (set-var wandering 1)
 (do-until (= (read-var wandering) 0)
   ((glide-seconds-to-xy 0.2
      (+ (xpos) (random -25 35))
      (+ (ypos) (random -30 25))))))
```

ScratchTalk's intermediate representation is made up of four components: actors, events, actions and a narrative graph. While these are somewhat tuned to the Scratch world model, they are intended to be generic constructs that can be applied to other application domains.

Actors

An actor in ScratchTalk is a simple object representation of a Scratch sprite. It associates a string name with a set of properties such as sprite size, initial location, costume dictionary (alternative sprite graphics), sound dictionary, etc. Compiled actions result in a set of scripts being associated with the actor object, which is then serialized and sent to the Scratch UI.

Events

An event captures either a point or an interval in time and is associated with an actor that is the subject of the event. Events serve as anchors for temporal relations and typically have an associated action that tells the compiler how to generate code capturing the actor behavior associated with that event. Because Scratch is a dynamic environment, events can be linked causally ($A \rightarrow B$) or an event can be associated with a conditional action indicating the activation of an event.

Actions

[2]SwiRL generates Propbank annotations which ScratchTalk converts to VerbNet tags using semlink[5].

Actions are plan templates that act like generic functions with multiple dispatch. The semantic head names an action class, and the arguments determine which method of that action is selected and associated with the event identified by that phrase. An action description such as

```
(defaction chase ($agent $subject)
  (do-forever
     (progn
       (point-towards $subject)
       (do-repeat %segment-steps
 (progn (forward %segment-step-size)
(wait %segment-step-time)))))
  (defaults (%segment-steps 5)
            (%segment-step-size 2)
            (%segment-step-time 0.1)))
```

expands to the Scratch script

```
((do-forever
   ((point-towards "cat")
    (do-repeat 5
     ((forward 2)
      (wait 0.01))))))
```

which when linked with catch

```
(defaction catch ($agent $subject)
  (wait-until (touching $patient))
```

in the context of the narrative graph (see figure 6) becomes

```
((do-until (touching "cat")
  ((point-towards "cat")
   (do-repeat 5
    ((forward 2)
     (wait 0.01))))))
```

As we will see later, there is a more powerful method for parameterizing actions, action editing, which is integral to constructing the system's lexicon of word and phrase meanings.

The Narrative Graph

A ScratchTalk animation is tied together by a graph of narrative relations among events. This functions loosely as a control flow graph for the animation. An event can *initiate* another event (a touch of a wall initiates a bounce off that wall) or it can *terminate* another event (when the dog catches the cat, the dog stops chasing the cat). There is also a *concurrent* relation which allows a set of actions to be initiated as a group. In this case the compiler will generate a common message so that any initiating or terminating event that is applied to one event is applied to all events labeled as concurrent.

When the final program is synthesized, the events and actions contained in the graph are converted into a set scripts each associated with a specific sprite. A set of optimizations is first applied, such as combining 'chase' and 'catch' above. Temporal and conditional constraints are extracted from the

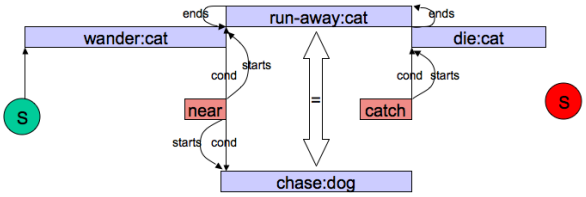

Figure 6. A narrative graph of the cat and dog example

narrative and expressed as messages or semaphore variables such that scripts start and stop in accordance with the constraints in that graph. The fully linked output of the chase script above becomes:

```
((when-receive "chase")
 (do-until (touching cat)
   ((point-towards cat)
    (do-repeat 5
      ((forward 2)
       (wait 0.01)))))
 (broadcast "die"))
```

The narrative graph structure is quite general and, with some extension, should work for any process that can be as an event-driven, control-flow graph.

EXAMPLE SCENARIO PART II

This simple example glosses over some significant issues of precision, such as where the cat and dog should start on the screen, how big should they be, and how fast should they execute these actions? Is "near" 100 pixels or 200 pixels? These are exactly the kinds of things a programmer must think about that the user shouldn't have to, unless the default assumptions violate their expectation.

The following sequences illustrate how the user and system can override defaults, create new defaults and augment an existing animation model[3].

By default, Scratch places all sprites in the center of the screen. In this case the dog is near the cat, the dog is touching the cat and thus the whole animation terminates immediately. It is easy to fix this by changing the default start location.

```
"The cat should start in the upper left
 corner and the dog should start in the
 lower right corner."
```

The conjunction template for "and" breaks this down into two phrases, each consisting of a "should start" action verb. These actions modify properties of the subject, rather than inserting or editing an event. A primitive script computes

[3]While the representational apparatus exists for the following examples, the system does not yet provide the end-to-end mapping from natural language to final code due to limitations in the semantic labeling system (the analysis section contains a short discussion of this problem).

the coordinates of a sprite of the size of cat and dog in the locations described so they start in the appropriate location. In the Social Computation model, placement primitives can be provided by more expert users. Ordinary users can then modify the application of those procedures to specific circumstances.

Another default behavior of the system is that the default wander script has no directional bias. The cat just keeps wandering randomly around the start location.

```
"The cat should wander towards
 the bottom left corner"
```

This speech act requires the system to perform a very sophisticated change to a pre-existing script. The step parameters of the loop that implements the random steps need to be modified to properly add the bias. This can be performed by having some general knowledge about scripts, parameters, coordinates, and motion. Most of the design work in ScratchTalk was ensuring that the representations above could support operations such as this.

Scripts often are missing default parameters; for example how much should the step calculation be biased? This example shows that the database of defaults is as important to the interactivity of the system as are the representations and behavioral descriptions. However, each person who creates or modifies a specific defaults value can empower other users in the network.

PLANNING IN SCRATCHTALK

Actions are essentially plans that can be tied to a goal. In the example above, goals were captured as imperative commands stated in natural language. The most interesting feature explored by ScratchTalk is the means by which actions are extended to accommodate new problems and new situations.

Plans as Programs

I employ a model of planning introduced by HACKER [12]. HACKER was an automated planning framework for a simple block stacking domain. Plans to satisfy goals were procedures. These procedures were incrementally modified to work around obstacles discovered in achieving a goal. In effect, programs are incrementally "hacked" together using a general representation of domain knowledge, programming techniques, bug types, and patch types. When a goal cannot be achieved in a single primitive step, it is treated as a bug. The bug is classified, which results in a patch attempt being made. The patch attempt may use domain knowledge or general programming knowledge, such as "if failed precondition, first satisfy precondition". Finally, a critic is generated that recognizes when a bug type has been created by the program and patches it automatically without actually running the program.

By incrementally training the system on goals and states of increasing complexity, the system constructs a rich lexicon of robust plans that solve many of the goals that one can formulate in the blocks world. In this way, HACKER not only becomes better at satisfying goals, but also better at constructing new programs.

There are some crucial differences between the blocks world and the Scratch world. ScratchTalk doesn't have a clean state model, and thus no means of formally classifying bugs in terms of that state. The system can have, however, a formal model of programming knowledge as well as different patch types.

The goal is to use the HACKER system model, but to rely on users to incrementally specify domain knowledge and bug types[4]. Domain knowledge and recognition of mistakes and problems are exactly what many humans can do that our machines can't.

- Recognizing errors. The transparency property of the domain is important because it enables users to see errors when they happen. Highly transparent systems tend to exhibit temporal locality of cause and observed effect, simplifying user critique.

- Critique and patching. Given a kind of critique, the system can apply a patch and attempt to repeat the prior behavior. The properties of interactivity and fail-soft are critical to being able to backup and retry in the presence of failures.

- Explanations and repair. By expanding an action plan, the system can generate explanations that can prime users to provide repair suggestions using language the system can understand.

Editing Events and Narrative Relations

As shown in the original example, users can update the program model by clarifying what happens to or around a specific event. The system currently supports the following editing operations for events and narratives:

- Inserting and removing events. The simplest editing operation is to do what the dialog does naturally, which is to have a prior event for the same sprite initiate the next even and the next even terminate the prior event. Concurrency of actions for the same object is not assumed by default. In fact such concurrency is typically implemented by editing actions.

- Temporal reordering. The user can change the order and casual relationships among events by adding or deleting temporal relations. This is currently not directly available to the user, but the same machinery is used by the dialog system when adding new events.

- Editing event action slots. An action can be overridden by an edit to create a new, derived action. This happens whenever the user augments an event as illustrated in the examples.

[4]With better access to state than in ScratchTalk, it should be possible to create critics which effectively model the domain without having to construct a complete model of the domain.

- Creating and revoking concurrency (see figure 6 for an example of the concurrency relation).

Action Editing

Action editing can be quite sophisticated. To make a certain kind of side effect on a program's behavior, we have to map the user's expression "chase faster" to a command that edits action scripts. The simplest editing operation might be "increment statement X in pattern Y" to match a specific statement in a procedure. We can make a pattern more complicated so that it can match more flexibly, similarly to HACKER. For example; "Increment the step-distance of the inner-most motion loop in this procedure." This latter form is what the system uses currently, although the number of such patterns remains small.

The editing operations available and used in the current system include:

- concatenating statement sequences
- insert a statement to the front of a loop body
- insert a statement at the end of a loop body
- remove a statement from a loop body
- add a loop exit condition
- override a default
- modify a parameter
- force an alternate action selection

Most of these operations are already in use by the event linker to merge event bodies and tie together all the actions using the variables and message passing. The problem of using natural language to evoke these editing operations is one of mapping properties or changes (faster, spin around, etc) to code sections that can serve as hooks for the editing primitives. An expert user can add these mappings automatically, or the machine can learn such mappings by generalizing from small numbers of examples and recording exceptions in the case of over generalization.

ANALYSIS AND FUTURE WORK

The central innovation in ScratchTalk is enabling users to use natural language to incrementally extend plans through critique and repair of existing plans. These individual contributions can be aggregated from a number of users to expand on the knowledge base available to a given user. These contributions come in the form of new actions, default values, action editing scripts, and new language mappings to these components. The work described here is insufficient to prove the full Social Computation model, but takes some important steps in that direction. During the development of the application several problems were identified that inform future work.

Domain Issues

The properties of the Scratch animation domain all contributed to the development of the program, however there are several properties lacking that I would want to see in the next experimental domain:

- Access to more internal state. Scratch provides an impoverished set of sensing primitives. A sprite cannot know the location of another sprite, only it's direction and distance[5].
- Synthesize and run. Generating code, then visualizing the program as a whole makes the interaction quite stilted. Far better to directly execute from the intermediate representation and use continuous planning techniques.
- Interpreter, not compiler. If always running interpreted, the system can learn when a certain state will cause a plan to fail and switch to another. ScratchTalk is forced to precompile all alternative plans into conditional code statements or parallel scripts, greatly increasing the complexity for learning.
- Direct manipulation. The Scratch UI was hard to use because of an inability to get inside the UI to allow users to perform direct manipulations that directly teach the system. A user-friendly system would provide multi-modal inputs, using direct manipulation where it makes sense and using language when the UI metaphors become awkward.

Plan Representations

One of the more challenging aspects of ScratchTalk is the action representation. A great deal of the time on this project was spent iterating through different action representations and the current implementation is far from satisfactory. The chief difficulty lies in handling plan variations under a specific goal, or different methods for performing the same plan step. This is further complicated by action editing. Some questions immediately arise. Do we edit a fully expanded version of a top level action, or do we edit specific sub-actions in the hierarchy? If we are editing the hierarchy, what information do we have about sub-actions (sub-goals) to determine the reference for an editing action?

This leads to an interesting design space. The following are some key parameters of this space.

- Dispatch. Is action dispatch a static process (easy for action expansion) or a dynamic dispatch (allowing for late binding based on world state). Static dispatch pushes complexity into conditional code, dynamic dispatch pushes complexity into the action editing rules.
- Hierarchy. How do we maintain information about a specific action hierarchy? If we want to edit a parameter or default for a sub-action, do we annotate the top level action or create an instance of the sub-plan and annotate that? How would that interact with dynamic dispatch?

[5]There are no function abstractions in Scratch, so the polar to Cartesian coordinate conversion makes the code very difficult to read.

- Expansion/Synthesis. Do we expand code as we construct it and edit the primitives, or do we maintain the hierarchy and synthesize as late as possible? I prefer the latter, but again it adds action editing rule complexity.
- Annotations. How do we keep track of what a parameters or default will be at final code synthesis time? Do we perform code expansion on every edit to validate all constraints, or do we wait until code synthesis time?

Learning

The chasing and running away example affords an interesting observation about the potential for acquiring general knowledge from users of this system. In general there are a large class of verbs with subject and object arguments for which a user would expect a reciprocal action. (For example: chase & run, attack & defend, threaten & fear, etc).

Given several examples of chase being associated with run away, the system can form the abductive hypothesis that anything chased runs away and add the reciprocal event by default in the future. The response of the user to this kind of assumption helps determine the reliability of a given default.

Complexity

Each layer of the action system requires default knowledge of various kinds and an ability to resolve natural language references to the appropriate subsystem, hierarchy layer, and editing routine. It is possible that the complexity of this architecture will severely limit the scaling of this system. The promise of Social Computation is that the collective contributions will compensate for the complexity by relying on human contributors to create, tune, and filter a rich library of heuristics that work often enough to keep users satisfied. This philosophical approach to developing computational functionality is heavily inspired by the views of Minsky [7] and Singh [10].

CONCLUSION

This paper introduces ScratchTalk, a prototype system enabling end-users to specify and modify programmatic behavior via natural language. This prototype system illustrates a representational apparatus amenable to the constraints of the Social Computation scripting model.

The current prototype allows users to describe simple animations in the Scratch programming environment using natural language. ScratchTalk introduces three key ideas: the concept of end user programming through plan elaboration, the central role of user critique and repair in that elaboration, and the use of natural language to reference plans, critiques, and repairs.

A version of the generated cat and dog example can be downloaded from

`http://web.media.mit.edu/~eslick/example.sb`

and run on the Scratch UI available from

`http://scratch.mit.edu/`

A Quicktime movie of this example can be viewed at

`http://web.media.mit.edu/~eslick/scratchtalk-demo1.mov`

REFERENCES

1. C. Anderson. The long tail. *natpe.org*, Jan 2006.
2. E. Charniak. A maximum-entropy-inspired parser. *Proceedings of NAACL*, Jan 2000.
3. A. DeHon, J. Brown, I. Eslick, J. Harris, L. Karbiner, and T. F. Knight. Global cooperative computing. *Workshop on Wide-Area Collaboration and Cooperative Computing, Second International WWW Conference: Mosaic and the Web*, 1994.
4. A. Ko, B. Myers, and H. Aung. Six learning barriers in end-user programming systems. *IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC'04)*, Jan 2004.
5. E. Loper, S. ting Yi, and M. Palmer. Combining lexical resources: Mapping between propbank and verbnet. *In Proceedings of the 7th International Workshop on Computational Linguistics*, 2007.
6. J. Maloney, L. Burd, Y. Kafai, N. andd Silverman Brian Rusk, and M. Resnick. Scratch: A sneak preview. *Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing*, Jan 2004.
7. M. Minsky. The emotion machine: Commonsense thinking, artificial intelligence, and the future of the human mind. *Simon and Schuster*, Nov 2006.
8. B. Myers, J. Pane, and A. Ko. Natural programming languages and environments. *Communications of the ACM*, 47(9):47–52, Jan 2004.
9. R. Panko. What we know about spreadsheet errors. *Journal of End User Computing*, 10(2):15–21, Jan 1998.
10. P. Singh. *Em-One: an architecture for reflective commonsense reasoning*. PhD thesis, Massachusetts Institute of Technology, 2005.
11. M. Surdeanu and J. Turmo. Semantic role labeling using complete syntactic analysis. *Proceedings Proceedings of CoNLL*, Jun 2005.
12. G. J. Sussman. *A Computational Model of Skill Acquisition*. PhD thesis, Masachusetts Institute of Technology, 1973.