State Semantics of Erasure in Sketch Applications

Jeffrey Browne
University of California,
Santa Barbara
jbrowne@cs.ucsb.edu

André Sayre
University of California,
Santa Barbara
asayre@cs.ucsb.edu

Timothy Sherwood
University of California,
Santa Barbara
sherwood@cs.ucsb.edu

ABSTRACT
Sketch researchers have produced impressive recognition accuracy in many different usage domains. Unfortunately, many applications that feature advanced recognition provide insufficient support for erasure. Researchers and developers can tell that allowing users to remove strokes from the board makes their job more difficult, but for the most part this is just a vague notion supported by trial and error. A better understanding of these issues would help developers reduce the intellectual overhead of design, either by avoiding the potential complexity of erasure when possible, or by crafting sensible design patterns for when it cannot be avoided.

In this paper, we formalize the semantics of sketch recognition applications that allow for both stroke creation and erasure. We divide the space of applications into seven classes defined by the type of information that can affect recognition (ordering of events and explicit tracking of erasure). Each class is modeled as a transition system over infinite states, where transitions between states occur when a stroke is either drawn or erased. With these semantics defined, we hope to simplify the task of developing sketch recognition applications that support erasure.

Author Keywords
sketch recognition, erase, semantics, transition system

ACM Classification Keywords
H.5.2 User Interfaces: Theory and methods

INTRODUCTION
Sketch-based applications up to this point have produced impressive results in terms of recognition accuracy over many different usage domains. At the same time, support for erasure has been largely relegated to only brief discussion or, more commonly, put off as future work. In many sketch-based systems, allowing users to remove their strokes often plays a secondary role to getting the recognition right in the first place, so it ends up being seen as an unnecessary feature.

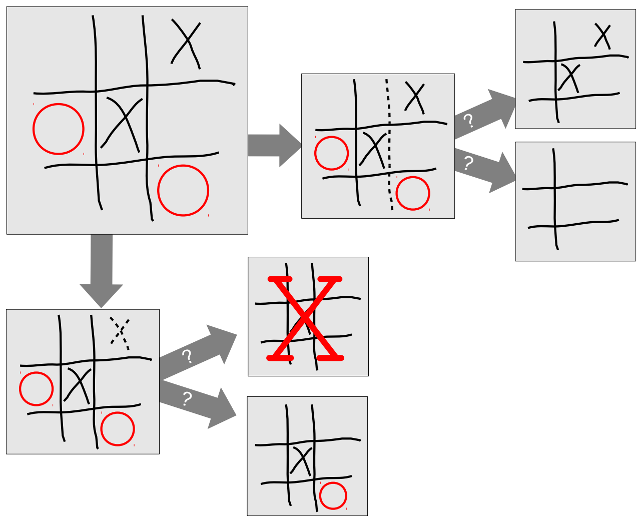


Figure 1. Erasure semantics can be tedious to enumerate even for simple sketch applications. When the user erases an X, the board could declare the game invalid, or roll back the moves. If she erases part of the board, the possibilities are even more abstract.

As a mode of interaction, however, erasure is a vital aspect of any realistic application driven by free-form pen input. Work by Dixon et al. [2] characterizing a typical whiteboard shows that 27% of user operations per drawing session are erases. Electronic whiteboards and tablets operate on free-form user pen strokes, so there are bound to be mistakes that go unchecked until the user notices them. Without a way to erase and correct mistakes, users of current sketch applications are largely restricted to clearing the board and starting from scratch in the event of an error.

In fact, erasure is used for more than just error correction, and applications should provide for rapid modification of drawings through stroke removal. One advantage of using a hand-drawn visualization is the ability to quickly try out ideas, often with the results of a drawing providing insight into ways a design could be improved. For example, a circuit diagram drawn out in full may hint at better ways to route wires, and a user must be able to make the required changes quickly to get much use out of such an application. Thus, even if recognition rates are perfect, the iterative nature of diagramming, alongside human imperfection, makes erasure a common action.

Though the value of supporting erasure in sketch applications is plain to see, this feature has been avoided in many projects. Removing strokes from an intelligent drawing board complicates applications, both in terms of engineering the system and designing the user interaction.

Some of the best recognition techniques employ machine learning algorithms whose operations are not easily reversed. For example, in SketchREAD, application context is exploited for stroke recognition by maintaining representative Bayesian networks. Partial networks are dynamically grafted onto the main network as strokes are added, but if strokes are erased from the board, shrinking the network accordingly "introduces subtleties into the recognition process that [the] system is not yet designed to deal with." [1]

Even if the mechanism of stroke removal is straightforward, the semantic subtleties of erasure can arise in simple applications. In an environment like a tablet PC, which allows erasure alongside free-form strokes, the number of possible use-cases explodes. For instance, in a tic-tac-toe game, the system might require the user to draw four strokes for a board at which point play begins. The user can then draw an X, followed by the computer playing an O iteratively until one side wins. Without erasure, this could be naively implemented by recognizing a board as four crossing lines, and then listening for a user's moves until the game is complete. As shown in figure 1, allowing users to remove strokes makes this rather simple application significantly more complex. After the first round of play, the application has drawn its O, but what happens if the user erases her X? The game could force a forfeit, but what if this was error correction and not cheating? Suppose play has progressed, and the user erases an X she drew two turns ago. We could clear all of her plays and start the game over, or treat the board as if the X is still there, or even play back the remaining moves as if the first move never happened. Beyond defining rules for the game, what if she plays several rounds and then erases a line making up the board? We could clear the board of all X marks, but what if she draws the line back in? The final quality of this program would wholly depend on how thoroughly the designer plans and tests the possible ordering combinations.

Many applications are clearly made much more complex by introducing erase, but sketch interface researchers are operating without a clear definition of which features are at fault. A better understanding of these issues could allow developers to avoid this complexity when possible, or to experiment with design patterns and best practices when it cannot be avoided. In either case, such a definition would help developers reduce the intellectual overhead of design.

In this work, we formalize the semantics that sketch recognition applications can follow regarding stroke creation and erasure. We classify the space of semantics that sketch applications can follow according to the type of events (strokes and deletions) that are tracked throughout the application's life and whether the ordering of these events matters (sequence vs. set semantics). We define each application class as an infinite transition system whose transitions (stroke addition and deletion) express its semantics.

RELATED WORK

Defining Semantics

While this is the first attempt to analyze of the semantics of stroke erasure, other work has looked at defining or using abstract semantics to benefit sketch. In their analysis [3], Freeman et al. examine the semantics of connectors (arrows, edges, and lines) in order to support general classes of diagrams. They divide the space of diagrams expressible through connector semantics into undirected graphs, directed graphs, and organization charts. While the context for their semantics is different than ours, the end goal of their work on semantics is the same: simplifying conceptual difficulties currently present in sketch application development. In their case, this involved extending the InkKit framework to better support diagrams.

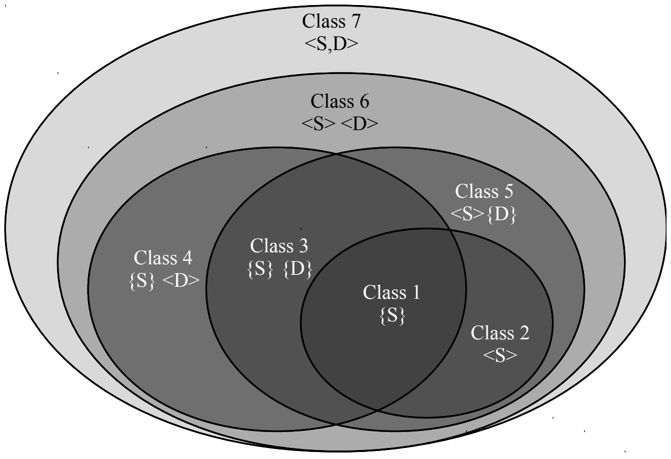
Implementations of Erasure

Many researchers have implemented erasure, despite a lack of formal semantics, though it is often limited in some way. Here we discuss how previous researchers have handled erasure in their projects.

Some applications simply disallow erasure of strokes. As discussed above, SketchREAD [1] users can only add strokes to the board, but this should only become a problem for complex diagrams. Flatland [8] divides drawing space into separate segments over which recognition "behaviors" can be executed. Though users can copy and delete segments as they wish, strokes within segments are permanent.

Several applications avoid much of the difficulty associated with erasure by re-evaluating modified strokes from scratch during recognition. InkKit [9] performs what the authors call "lazy" recognition of strokes, allowing users to edit the underlying board as they wish before explicitly invoking recognition. In this way, the iterative process of adding and deleting strokes can largely be ignored, as only the strokes present upon recognition affect application meaning. MathPath^2 [7] uses a circle-tap gesture to invoke recognition of strokes to similar effect. Erasure in this system involves removing strokes via a scribble gesture, and explicitly re-recognizing the modified strokes. SimuSketch [5] can also be categorized as explicit recognition through its "recognize-on-demand" feature. However, erasure in this system does not occur on strokes themselves; after recognition of basic circuit elements, a user may delete the higher-level objects in their entirety. SILK [6] handles erasure in a similar way to SimuSketch, in that erasure occurs by deleting higher-level objects (in this case interface widgets) rather than individual strokes.

The Electronic Cocktail Napkin [4] seems to take a different approach to erasure than the previous applications. Strokes in this system are recognized as basic glyphs in an "eager" manner, that is, immediately after they are drawn. These glyphs are then matched to higher-level spatial predicates, which describe the composition of more complex structures. As the system supports erasure through a gesture command, erased strokes must propagate and remove glyphs, which then affects which spatial predicates are matched.



Class	Strokes	Deletes	Example
Class 1	Set	Ø	ZombieBoard
Class 2	Sequence	Ø	SketchREAD
Class 3	Set	Set	
Class 4	Set	Sequence	Bomb defusing game
Class 5	Sequence	Set	
Class 6	Sequence	Sequence	Consumer-producer list
Class 7	Combined Sequence		Undo button / code bugs

Figure 2. Semantics classes in terms of sequence and sets of strokes and deletions. Each class is expresses some subset of the semantics of more specific classes.

THE SEMANTICS OF RECOGNITION

One can think of any sketch recognition application as assigning meaning to some drawing-board states, which change as the user draws or deletes strokes. One board state is only different from another if the recognition algorithm has different meanings for each; otherwise, to the application, the states are indistinguishable. For instance, systems that interpret strictly geometric relationships assign a different meaning to different sets of visible strokes, but the various drawing orders for those strokes all result in the same meaning. Other applications (such as certain games) differentiate between structurally equivalent sets of strokes depending on their drawing order, or even because of strokes that were previously erased.

In this section, we describe the classes of recognition algorithm semantics in terms of transition systems, which behave like finite automata with an infinite number of states. For an application to fit the semantics of the transition system, every state must map to a single possible "meaning." Any operation that can change the meaning of the board acts as a transition from one board state to another, so we consider stroke drawing ($AddStroke$) and stroke erasure ($DeleteStroke$) as possible transitions.

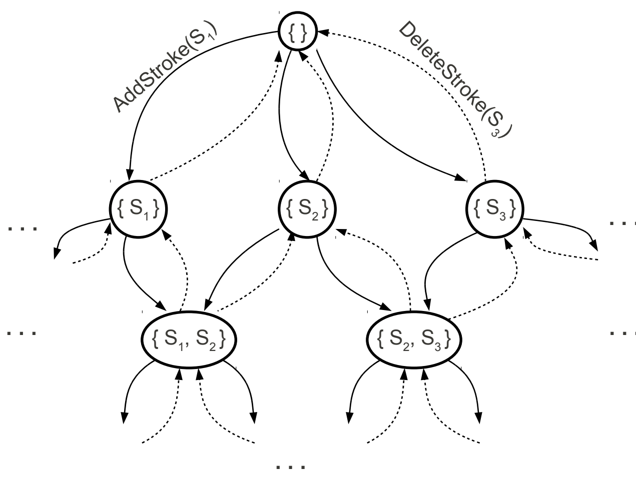


Figure 3. The transition system representing the semantics of class 1. Application state is wholly dependent on the unordered set of strokes.

A transition system is defined as

$$TS = \langle S, S_0, \delta \rangle$$

where S is an infinite set of recognition states for an application, with S_0 being the initial, empty board state and

$$\delta : S \times \{AddStroke \cup DeleteStroke\} \rightarrow S$$

is the transition function that maps a state and an $AddStroke$ or $DeleteStroke$ operation to another state. For our purposes, we consider strokes as the finest granularity of user input, and the operations of adding a stroke and deleting a stroke are presumed atomic. Since a user can draw any stroke at any time, each board state will have an infinite number of $AddStroke$ transitions. We further require that deleting a stroke can only occur after it has already been added to the board, so a state will only have as many $DeleteStroke$ transitions as it has strokes.

We define the classes of semantics according to the type of events (strokes and deletions) that affect recognition logic and whether ordering of these events can result in board states with different meanings. The breakdown of each class in terms of these variables is shown in figure 2.

We first discuss the extreme ends of our semantics classes as they are the most intuitive. We then explore intermediate classes of semantics by progressively relaxing information constraints, present in the most specific class.

Class 1: Stateless Erasure of Unordered Strokes

As discussed earlier, some applications assign meaning dependent only on the collection of strokes on the board independent of previously deleted strokes or their draw order. These applications act as if they become aware of the final set of strokes all at once, so deleting a stroke is equivalent to having never drawn it.

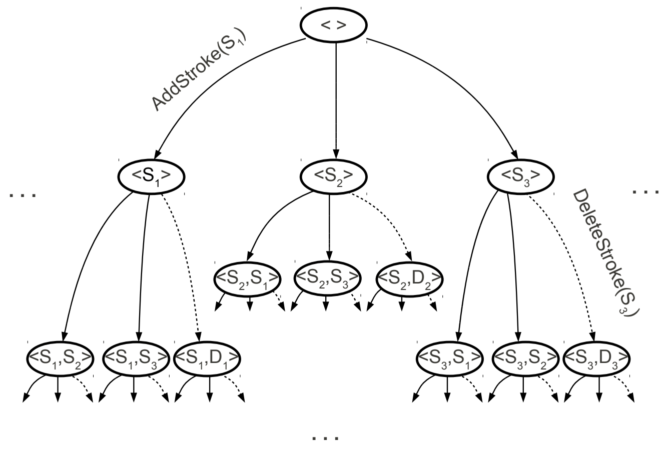


Figure 4. The transition system representing the semantics of class 7. A state's meaning is completely dependent on the ordered sequence of stroke and delete operations.

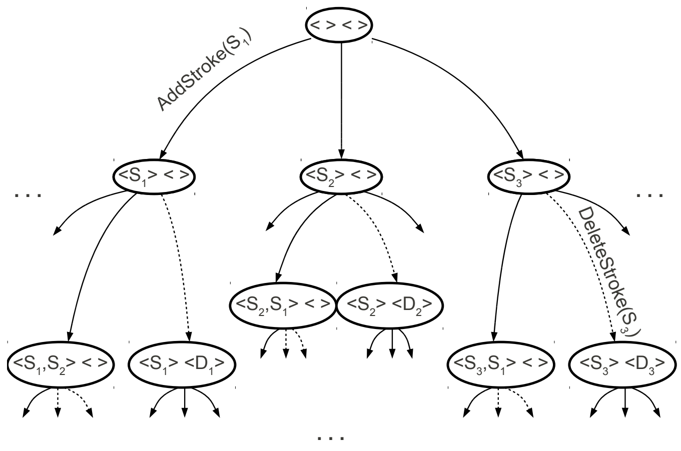


Figure 5. The transition system representing the semantics of class 6. Application state depends on both the ordering of the strokes and the ordering of deletes, but not on the ordering between strokes and deletes.

As shown in figure 3, the transition system equivalent to these semantics has states represented solely by strokes grouped into a set. If the application is in any state S_i, drawing a stroke means taking the $AddStroke$ transition to a state with the new stroke plus all of S_i's strokes. Erasing a stroke in S_i means following a $DeleteStroke$ transition to the state with all strokes except the erased one. $AddStroke$ operations are forward edges that always increase state complexity while $DeleteStroke$ operations are backward edges that reduce complexity. In these semantics, any state S_i of sufficient complexity can be reached by many paths– one for every ordering of $AddStroke$ operations, as well as multiple "backtracking" paths that reach S_i by removing strokes.

Having multiple, equivalent paths over stroke addition means that recognition is effectively offline, and algorithms can optimize the order in which strokes are considered; for example, a workflow diagramming tool may be able to avoid multiple passes over the strokes if they are evaluated from top to bottom, even though the user could have drawn them in any order. At the same time, however, since stroke ordering is not represented, applications cannot use this information to aid recognition (e.g. consecutive strokes can be related in some domains). Applications that operate on complete sets of user strokes (such as Saund's ZombieBoard [10]) fit this semantics

Class 7: Totally Ordered Strokes and Erases

In the most general case of our semantics space, applications assign a unique meaning to the board depending on the order of stroke drawing and erasure events; every stroke can change the board's meaning differently depending on when it is added, and deleting a stroke can have different meanings depending on what has been drawn or erased up to that point. Applications that fall into this semantics do not define their behavior according to a consistent structure, but rather can act as if each stroke or erasure is a special case. For instance, any application that provides a last-in-first-out "undo" feature fits into this class since the sequence of stroke and delete events uniquely determines the behavior of pressing an undo button. Applications can also fall into this category as a result of error since improperly "rolling back" the effect of previous events can lead to a state that is only reachable by a specific order of strokes and deletes.

As shown in figure 4, each state is represented by an ordered sequence of $AddStroke$ and $DeleteStroke$ events. By tracking stroke deletion as a first-class event, erasure no longer removes a stroke from the a board state as in class 1. This allows erased strokes to affect application meaning even after they are no longer visible. For example, the application could set some property once a stroke is drawn (e.g. marking a "G" changes some widget's color to green) that persists even after the stroke is erased (e.g. the widget stays green). However, this information comes at the cost of monotonically increasing state complexity with every operation; since deleted strokes are not forgotten, recognition of new strokes may have to take into account the entire history of the application. Since adding or deleting strokes transitions from one state to a different subtree of states every time a stroke is drawn or erased, every state in these semantics is uniquely describable by a unique sequence of $AddStroke$ and $DeleteStroke$ operations.

INTERMEDIATE SEMANTICS

Now that we have discussed the most specific (class 7) and most relaxed (class 1) semantics, it is valuable to explore the intermediate classes of semantics that sketch applications can express. We show that, beginning with totally ordered semantics, one can iteratively relax ordering requirements to express progressively less restrictive semantics.

Class 6: Stroke Sequence, Delete Sequence

This class of semantics is given by the transition system shown in figure 5. In this system, every state is defined by separate sequences of strokes and deletes. Thus, every state is reachable via multiple paths, each of which maintains a relative ordering between $AddStroke$ events and between $DeleteStroke$ transitions, but where $AddStroke$ and $DeleteStroke$ transitions are not mutually ordered.

An application that follows these slightly relaxed ordering semantics could be one that treats strokes and deletes as producer and consumer events. For example, one user could draw a tic next to a list of chores she wants her roommate to do in the order she needs them done. Her roommate erases tics in the order he completes the chores, and is alerted if he goes out of order. Note that the order of each user's actions only matters relative to his or her own actions, but ordering between strokes and deletes is irrelevant.

Class 6 expresses a subclass of class 7's semantics. This can be shown by ignoring the relative ordering between strokes and deletes in the class 7's single, totally-ordered sequence. That is, a class 6 state of $(\langle S_1, S_2\rangle\langle D_1, D_2\rangle)$ can be emulated by, for example, treating the class 7 state $\langle S_1, D_1, S_2, D_2\rangle$ as equivalent to $\langle S_1, S_2, D_1, D_2\rangle$.

Class 5: Stroke Sequence, Delete Set

The transition system describing these semantics uses states represented by a sequence of strokes and a set of deletes. States in these semantics can be reached through multiple operation paths, but each must enforce an ordering on $AddStroke$ operations. Since deletes are represented as a set, paths can follow any order of $DeleteStroke$ operations.

Class 5 states differ from those of class 6 by relaxing the ordering restriction on deletes, so class 6 applications can emulate the semantics of class 5 by simply ignoring ordering requirements between either strokes or deletes. For example, to emulate the class 5 state of $(\langle S_1, S_2\rangle\{D_1, D_2\})$, the class 6 state of $(\langle S_1, S_2\rangle\langle D_1, D_2\rangle)$ must have the same meaning as $(\langle S_1, S_2\rangle\langle D_2, D_1\rangle)$.

Class 4 Stroke set, Delete sequence

This class of semantics also relaxes class 6, but by ignoring stroke order (the order of erasure still matters). A state is defined by its set of strokes and a sequence of deletes, so a state S_i can be reached by multiple paths where the order of $AddStroke$ operations does not matter, but all paths have the same $DeleteStroke$ ordering.

These semantics cover any application that can operate on an unordered set of strokes, but where the order of deletes matters. Though this class is somewhat abstract, an example application could be a two-player bomb-defusing game. One player draws colored wires in any order to link the trigger circuits to the detonator, but the second player's victory depends on the order in which she erases the wire strokes.

Expressing class 4 semantics in terms of class 6 is similar to expressing class 5; ordering restrictions on strokes are ignored, but deletes remain ordered.

Class 3: Stroke Set, Delete Set

Recognition algorithms that keep track of added strokes as well as delete operations without retaining any ordering information fit into this class. Since states are defined over sets of strokes and sets of deletes, the ordering of strokes and deletes is irrelevant. However, like all higher classes, once a stroke is added with an $AddStroke$ operation, in these semantics it will never be removed, and state complexity increases monotonically with each operation. However, the relaxed ordering requirements means that there are still multiple execution paths that can result in the same state. For example, the transition sequences

$$\langle Add(S1), Add(S2), Del(S1)\rangle$$

and

$$\langle Add(S1), Del(S1), Add(S2)\rangle$$

both result in the state $(\{S1, S2\}, \{D1\})$.

These semantics can be seen as a relaxation of either class 4 or 5, where the remaining ordering information is ignored.

Class 2: Stroke Sequence

The transition system representing these semantics has states defined only by a sequence of strokes. These applications enforce an ordering between strokes, but since deletes are not tracked, erasure means removing a stroke from the state. Much like class 1, any state S_i in this system is reachable by multiple forward and backward paths where $DeleteStroke$ transitions remove strokes from the sequence, in essence forgetting they were ever drawn. However, like class 5, paths that lead to S_i must have the same ordering over $AddStroke$ transitions.

This kind of application builds up state as strokes are input, using prior information to affect recognition of later strokes, but following erasure behaves as if the stroke was never created. The resulting meaning after the user draws a new stroke is a function of the previous meaning and the new stroke, so to forget a stroke, erasure rolls back the board's state to one before the erased stroke was added, then replay stroke additions (minus the erased stroke).

In these semantics, a text recognition algorithm could leverage the left-to-right writing style of most Western users to anticipate upcoming letters. A user might draw a stroke that alone looks like an "F" following a sequence recognized as "PURPL." The application could then weight recognition results to preferentially complete the string "PURPLE" with an "E" character. Note that the same nudge would not necessarily occur if the strokes were drawn in another order. Since the "E" stroke depends on the interpretation of the "PURPL" strokes, then erasing the letter "L" would invalidate the logic that generated the "E" stroke's meaning. The application would then have to re-evaluate the "E" stroke in the updated context of "PURP".

Also, an application that provides limited "undo" support where strokes can be removed in a last-in-first-out order would need at least ordered strokes to be expressible in a transition system.

In order to express class 2 semantics in terms of class 5 semantics, deletion must act as if it removes the stroke from the board. Class 2 can thus be expressed in class 5 terms by ignoring any stroke that pairs with a delete. This is a potentially complicated transformation, since strokes in classes 2 and 5 are strictly ordered. Strokes that have a corresponding delete must not affect the meaning of strokes that follow them, so each delete must act as a reversal of its corresponding stroke. This transformation results in every class 2 state having infinite equivalent class 5 states. For example, the class 2 state $\langle S1 \rangle$ would be equivalent to class 5 states $(\langle S_1 \rangle \{\})$, $(\langle S_1, S_2 \rangle \{D_2\})$, $(\langle S_2, S_1, S_3 \rangle \{D_2, D_3\})$, etc.

Finally, relaxing class 2 semantics by ignoring ordering information will act to emulate class 1 semantics. In this way, class 2 states like $(\langle S_1, S_2, S_3 \rangle)$ would be treated as equivalent to $(\langle S_3, S_1, S_2 \rangle)$ (along with any other permutation) to emulate a class 1 state of $\{S_1, S_2, S_3\}$.

Combined Semantics

Real-world sketch applications often do not fall into any single semantics category all of the time. However, if an application expresses some higher-level semantics in even a single case, then it must accommodate the behavior of the most complex semantics when strokes are added or deleted because every state in the transition system must map to a single meaning. If even a single state is differentiable by information not in expressed in a simpler semantics, then an application must be represented in a more complex transition system with states that can represent the additional information.

The tic-tac-toe application discussed earlier, for example, may express different semantics depending on how much the user has drawn. When the first strokes of a tic-tac-toe board are added, the recognition algorithm does not need to know the order in which they are drawn, so it could be expressed in class 1 semantics.

However, once the board is drawn, the state of the game becomes intimately tied with the order in which the user draws her X marks. The application's state (the computer's move choices) is different depending on whether the user draws her first two X moves in the center square then in some corner or in the reverse order, so the application now expresses at least class 2 semantics.

When a user is allowed to erase strokes, all of the subtleties discussed earlier arise, and the programmer must decide on a higher level semantics to follow. If every case is evaluated individually, this may require the highest-level semantics, class 7.

EMULATING COMMON FEATURES

While our semantics classes have stroke addition and deletion a first class operations, many sketch applications have other features beyond these two primitives. In this section, we discuss ways of emulating some of these features within our semantics classes.

Gestures

One common feature in sketch applications that is not represented directly is support for gestures. When a user draws a gesture, an application will treat it like a command instead of a stroke to be analyzed, so the visible mark typically disappears immediately. Since the board state changes in response to a gesture, an application must differentiate meaning between board states depending on strokes that are no longer visible. In our semantics, this is expressed by maintaining delete information as a first-class event. In fact, gestures can be thought of as a special case of the interactions already expressible by classes 3 through 7; gestures are simply strokes that are immediately erased after they are drawn. Thus, the sequence $\langle S_1, S_2, D_1, G \rangle$ where G is some gesture can be seen as equivalent to $\langle S_1, S_2, D_1, S_G, D_G \rangle$. Classes 1 and 2 cannot represent gestures, since they assign meaning exclusively according to visible strokes.

Partial Erasure

While these classes of semantics describe applications that provide for erasure at the stroke-level granularity, applications may want to provide for sub-stroke granularity (individual points) when erasing. Partial erasure can be expressed in our semantics as a $DeleteStroke$ operation followed by one or more $AddStroke$ operations.

Partial erasure involves removing some parts of a stroke while leaving others untouched. Given any stroke s, a partial erase will split it into sub-strokes $s_0, s_1, ..., s_k$. Given the operations in our semantics, for class 1 applications this is equivalent to performing a $DeleteStroke(s)$ followed by $AddStroke(s_0), AddStroke(s_1)...AddStroke(s_k)$. Since deleted strokes are forgotten and ordering is not taken into account, this process sufficiently expresses partial erase.

Difficulty arises in class 2 where ordering is important, since the series of $AddStroke$ operations could add partial strokes to a later part of the stroke sequence than the original stroke. Partial erasure must then act as if it rolls back the application state, removes the original stroke (full erasure), adds the sub strokes to the sequence (partial erasure) and then replays subsequent strokes.

Since semantics classes 3 through 7 retain information about delete actions, partial erasure does not necessarily have the same meaning as deletion followed by adding sub-strokes. Thus, it may be necessary to express the semantics of these applications as keeping track of partial-erasure events as well as whole-stroke addition and deletion events, where each state has a transition for every possible partial erasure.

Here, subtleties of interpretation can affect further simplification. If an application regards a single multi-segment stroke the same as a matching sequence of two-point segments drawn separately, then partial erase can be expressed in our semantics. Every stoke would be broken down into its constituent line segments, and each would be added or deleted separately; $AddStroke$ and $DeleteStroke$ would effectively be replaced by $AddSegment$ and $DeleteSegment$ operations. In this modified semantics, a partial erase from a state would be a path of $DeleteSegment$ transitions.

If line segments drawn as part of a single stroke can be interpreted differently from identical segments drawn piecemeal, then our semantics classes would have to be fundamentally modified to capture this information.

FUTURE WORK

Though our model of semantics captures most critical aspects of sketch-based applications, there are interaction methods that we currently cannot formally express. Future work in this domain could involve extending or modifying the state semantics presented here to include these features. For example, our semantics cannot universally address the concept of stroke transformation. Modifying a stroke could range from translation, to rotation, scaling, smoothing, or splitting. By first approximation, the semantics of modifying strokes will be a significant generalization of stroke addition and erasure.

One interesting opportunity this work presents is how our classification system can directly benefit developers. The seven classifications we define have various trade-offs between implementation complexity and richness of interaction, and programmers might be willing to trade certain features for simplicity and robustness. It could also be valuable to investigate ways that a development framework could better support novice programmers through explicit mechanisms for maintaining the semantics level of an application. At some point, it may even be possible to automatically generate code to support erasure, given just a semantic description of an application.

ACKNOWLEDGMENTS

We would like to thank Dr. Tevfik Bultan for his input regarding this work.

REFERENCES

1. C. Alvarado and R. Davis. Sketchread: a multi-domain sketch recognition engine. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, UIST '04, pages 23–32, New York, NY, USA, 2004. ACM.

2. R. Dixon and T. Sherwood. Whiteboards that compute: A workload analysis. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 69 –78, 2008.

3. I. J. Freeman and B. Plimmer. Connector semantics for sketched diagram recognition. In *Proceedings of the eight Australasian conference on User interface - Volume 64*, AUIC '07, pages 71–78, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

4. M. D. Gross. Recognizing and interpreting diagrams in design. In *Proceedings of the workshop on Advanced visual interfaces*, AVI '94, pages 88–94, New York, NY, USA, 1994. ACM.

5. L. B. Kara and T. F. Stahovich. Hierarchical parsing and recognition of hand-sketched diagrams. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, UIST '04, pages 13–22, New York, NY, USA, 2004. ACM.

6. J. A. Landay. Silk: Sketching interfaces like krazy. In *In Proceedings of CHI'96 on Human factors in computer systems: common ground, ACM*, pages 398–399. Press, 1996.

7. J. J. LaViola, Jr. and R. C. Zeleznik. Mathpad2: a system for the creation and exploration of mathematical sketches. *ACM Trans. Graph.*, 23:432–440, August 2004.

8. E. D. Mynatt, T. Igarashi, W. K. Edwards, and A. LaMarca. Flatland: new dimensions in office whiteboards. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 346–353, New York, NY, USA, 1999. ACM.

9. B. Plimmer and I. Freeman. A toolkit approach to sketched diagram recognition. In *Proceedings of the 21st British HCI Group Annual Conference on People and Computers: HCI...but not as we know it - Volume 1*, BCS-HCI '07, pages 205–213, Swinton, UK, UK, 2007. British Computer Society.

10. E. Saund. Bringing the marks on a whiteboard to electronic life. In *Proceedings of the Second International Workshop on Cooperative Buildings, Integrating Information, Organization, and Architecture*, pages 69–78, London, UK, 1999. Springer-Verlag.