Combining Sketching and Traditional Diagram Editing Tools

GEM STAPLETON, University of Brighton
BERYL PLIMMER, University of Auckland
AIDAN DELANEY, University of Brighton
PETER RODGERS, University of Kent

The least cognitively demanding way to create a diagram is to draw it with a pen. Yet there is also a need for more formal visualizations, that is, diagrams created using both traditional keyboard and mouse interaction. Our objective is to allow the creation of diagrams using traditional and stylus-based input. Having two diagram creation interfaces requires that changes to a diagram should be automatically rendered in the other visualization. Because sketches are imprecise, there is always the possibility that conversion between visualizations results in a lack of syntactic consistency between the two visualizations. We propose methods for converting diagrams between forms, checking them for equivalence, and rectifying inconsistencies. As a result of our theoretical contributions, we present an intelligent software system allowing users to create and edit diagrams in sketch or formal mode. Our proof-of-concept tool supports diagrams with connected and spatial syntactic elements. Two user studies show that this approach is viable and participants found the software easy to use. We conclude that supporting such diagram creation is now possible in practice.

Categories and Subject Descriptors: H.5.2 [**Database Management**]: Heterogeneous Databases

General Terms: Diagrams, Sketching, Intelligent Systems

Additional Key Words and Phrases: Sketching interfaces, diagram editing, tools for visual languages

ACM Reference Format:
Gem Stapleton, Beryl Plimmer, Aidan Delaney, and Peter Rodgers. 2015. Combining sketching and traditional diagram editing tools. ACM Trans. Intell. Syst. Technol. 6, 1, Article 10 (March 2015), 29 pages.
DOI: http://dx.doi.org/10.1145/2631925

1. INTRODUCTION

With the rapid advance of technology and the now-prevalent use of touchscreen devices, it is timely to consider the possibilities for software that helps users draw diagrams. The range of input mechanisms has increased and we should consider whether there is benefit to utilizing stylus-based diagram creation. Our experience suggests that when people create diagrams, they sometimes begin by sketching on paper or a whiteboard to fine-tune ideas and identify the diagram that they require. It can often be the case that people do not know exactly what they want their diagram to look like before they begin creating it. This, in itself, suggests that stylus-based diagram creation, which allows rapid diagram creation and editing, does have a place.

Authors' addresses: G. Stapleton and A. Delaney, School of Computing, Engineering and Mathematics, University of Brighton, Brighton, BN2 4GJ, UK; emails: {g.e.stapleton, a.j.delaney}@brighton.ac.uk; B. Plimmer, Department of Computer Science, University of Auckland, Auckland, New Zealand; email: b.plimmer@auckland.ac.nz; P. Rodgers, School of Computing, University of Kent, Canterbury, CT2 7NF, UK; email: p.j.rodgers@kent.ac.uk.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2015 ACM 2157-6904/2015/03-ART10 $15.00
DOI: http://dx.doi.org/10.1145/2631925

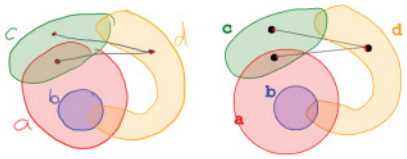

Fig. 1. A sketch and its formal version.

Sketching tools have the advantage of providing natural interaction with the diagram and aids problem solving and communication [Goldschmidt 1999]. The resulting hand-drawn diagrams are more effective for the external representation of a problem than using traditional computer editing tools [Goel 1995]. Thus, there is a role for sketched diagrams with benefits arising from allowing diagram creation in this way. The left of Figure 1 is a user-created sketch, whereas the right shows a beautified, so-called formal, diagram. Formal diagram creation and editing means specifying the location and appearance of shapes using a mouse or keyboard. In practice, this usually means that the user specifies the routing of curves by defining bend points with a mouse and often restricts shapes to simple ones such as ellipses or rectangles. Shapes can typically be rotated and scaled by mouse input. Formal diagrams also have a role to play, in part because of the perception that sketches are incomplete, unfinished, or inaccurate in some way [Yeung et al. 2008]. In addition, the formal diagram is generally required for distribution. Thus, intelligent diagram creation systems should support visualization via formal diagrams, which appear as though they have been drawn in an editing tool rather than by hand, as well as sketched diagrams. Hence, there is a need to advance our understanding of the methods and techniques that are necessary to produce software systems that support diagram creation via both sketching, using a stylus, and traditional diagram entry approaches, which typically use keyboard and mouse-based interactions.

For good interaction design, it is important to allow the sketch and formal diagram to be created and modified. This requires a close linkage between the sketch and the formal diagram so that editing operations are automatically reflected in both views. The tool used for this creation should automatically generate an equivalent representation in the other view. Supporting both views brings with it the challenge of ensuring that the sketch and formal diagram views are consistent and, consequently, that they convey the same information. It is certainly possible that, when diagrams are created or edited, a conversion between the sketch and formal views will produce visualizations that have different syntaxes and, therefore, represent different information. This is clearly important to avoid so we need techniques to automatically rectify differences.

This article provides insight into the issues that arise in providing intelligent diagram creation systems that support the previously described functionality. The novel contribution is a general approach and set of theoretical techniques for developing such diagram creation tools. To demonstrate the practical utility of our approach, we apply it to Euler diagrams augmented with graphs as a case study. We give an overview of related work in Section 2. Our general approach to supporting diagram creation and editing is presented in Section 3. The application of the techniques is exemplified in Section 4, where we present our case study and a user evaluation of our software.

2. RELATED WORK

This article builds on results on user interaction work that focuses on converting a sketch into a formal diagram. We also utilize approaches to formalizing visual languages via a concrete and abstract syntax. Finally, we provide an overview of visual languages that may benefit from this kind of work, in part to provide motivation for

devising intelligent sketching technology and to justify our choice of case study (Euler diagrams augmented with graphs).

2.1. User Interface Work on Sketching Technology

A sketch tool allows input to a computer using a stylus. Numerous sketch tools have been proposed for visual languages including concept maps [Jiang et al. 2011], graphs [Plimmer and Freeman 2007], UML class diagrams [Damm et al. 2000; Hammond and Davis 2002], and Euler diagrams [Wang et al. 2011]. For a general review of sketch tools research, we refer the reader to Johnson et al. [2009]. In such tools, when a user sketches an item, it must be understood by the tool in order to compute the so-called abstract syntax (see Section 2.2). Thus, to provide intelligent support, these tools include recognizers. The most common approach is to first recognize the individual strokes, and numerous gesture recognizers are proposed for this [Chang et al. 2012; Rubine 1991; Wobbrock et al. 2007]. Taking the output from the gesture recognizer, rules [Plimmer and Freeman 2007] or grammars [Costagliola et al. 2002; Hammond and Davis 2002] are used to infer the concrete syntax of the visual language. Some sketch tools convert the sketched input into a formal representation. For example, Damm et al. [2000] present UML class diagrams at various levels of formality and Plimmer and Freeman [2007] can generate files of appropriate format for formal diagramming tools. By contrast, conversion from formal to sketch is less explored. SketchNode provides formal to sketch conversion of graphs [Plimmer et al. 2010]. The tool presented in Section 4 directly extends the SketchNode code base. We are unaware of any tools, other than SketchNode, that provide the functionality that is necessary for combining sketched and formal diagrams into a system like that described in this article.

2.2. Formalization of Visual Languages

Over the last few decades, the approach to defining visual languages has evolved. In the early work on diagrammatic logics, for example, the formalization tended to be purely at the so-called concrete syntax level. This can be seen in Shin's seminal work, where she devised the first formalized sound and complete diagrammatic logic [Shin 1994]. The concrete syntax, sometimes called the token syntax [Howse et al. 2001], directly formalizes the drawn diagrams. Thus, the concrete syntax carries with it all of the geometric properties of the diagram. Since this early work, new approaches to defining the syntax of visual languages have been proposed, and it is now common to have both a concrete and an abstract syntax; see Baar [2006b] and Erwig [1998]. The abstract syntax, sometimes called the type syntax, can be thought of as a description of the diagram that captures the semantically important features. Thus, the abstract syntax need not capture geometric properties. The use of an abstract and concrete syntax can be seen in numerous visual languages, such as constraint diagrams [Fish et al. 2005; Kent 1997] and spider diagrams [Gil et al. 1999], which use spatial relationships to convey information, and deterministic finite automata [Maier et al. 2008] where connectivity is semantically important.

2.3. Visual Languages: Applications and Properties

There are a wide variety of visual languages in existence, such as statecharts and spreadsheets, which convey information using spatial relationships or connectivity properties, for example. From the perspective of this article, we are interested in visual languages that are diagrammatic in nature, drawn in two dimensions. Visual languages that fall into this class include the suite of UML notations (except for OCL) [Unified Modelling Language 2006], visual OCL [Bottoni et al. 2001], and logics based on Euler diagrams [Gil et al. 1999; Kent 1997; Mineshima et al. 2012; Shin 1994; Swoboda and Allwein 2005], graphs [Wilson 1996], conceptual graphs [Sowa 2013], existential graphs

[Dau 2007; Peirce 1933], and Petri nets [Petri Nets 2013]. These languages all consist of basic syntactic items that form the building blocks of concrete (or abstract) diagrams. The ways in which these building blocks are connected, their spatial relationships, and their specific geometric shapes can all be of semantic relevance and important for ease of interpretation, arise from aesthetic preference, or just simply be used out of convention. For example, class diagrams use arrows to connect two rectangles in order to provide information about an association between two classes. Thus, class diagrams use connectivity and geometric shapes to convey information.

Existential graphs use hyperedges[1] to represent the existence of elements, and the endpoints of those edges connect to other syntactic items (essentially textually displayed nodes) to assert set membership or to express binary or, generally, *n*-ary relationships between elements [Dau 2007; Peirce. 1933]. As well as using connectivity, existential graphs also exploit spatial relationships for conveying meaning: enclosure by a closed curve asserts negation and juxtaposition represents conjunction.

Euler diagrams augmented with graphs also use connectivity and spatial relationships to convey meaning [Stapleton and Masthoff 2007]. As an example, a user-sketched augmented Euler diagram can be seen on the left of Figure 1, with a formal version placed on the right. The labeled closed curves form an Euler diagram and they intuitively visualize exclusion, containment, and intersection of sets, generalizing Venn diagrams. That is, Euler diagrams use spatial relationships to convey semantics, and curves can take different geometric shapes. The spatial relationships utilize containment, disjointness, and overlapping, although other spatial relationships, such as alignment, are not of semantic importance. They often use curves with specific shapes, like circles [Wilkinson 2012] or regular polygons [Kestler et al. 2008], because of their aesthetically pleasing nature. Graphs are incorporated into Euler diagrams to represent connected individuals or items within the sets, allowing the visualization of more complex information. Thus, Euler diagrams augmented with graphs use all three features (connectivity, spatial relationships, and specific geometric shapes). Therefore, using Euler diagrams augmented with graphs, we can provide insight into how to devise intelligent diagram creation tools for other visual languages.

Of course, this is not the only visual language that we could have chosen for our case study. A further motivation for our choice is that these diagrams are of interest in their own right, having numerous application areas. In fact, they are a popular and widely used visualization technique. The varied areas in which augmented Euler diagrams are used for information visualization include crime control [Farrell and Sousa 2001], computer file organization [DeChiara et al. 2003], classification systems [Thièvre et al. 2005], education [Ip 2001], genetics [Kestler et al. 2005], and medicine [Soriano et al. 2003]. They form the basis of constraint diagrams [Kent 1997], spider diagrams [Gil et al. 1999], and concept diagrams [Howse et al. 2011] and can even be thought of as a basis of statecharts [Dunn-Davies and Cunningham 2005].

3. A GENERAL APPROACH TO SKETCHED AND FORMAL DIAGRAM CREATION

We have argued that diagram creation should be allowed in two ways: via a sketching interface using a stylus and via a formal diagram interface using more traditional point-and-click operations. In addition, users should be able to edit the diagram in either view, since this is a natural way of interacting with the software. Editing includes the ability to add and delete items and alter their layout. This section presents our general approach to supporting diagram creation utilizing sketching and formal interfaces, both of which allow diagram drawing and editing.

[1]Each hyperedge can have two or more endpoints.

A high-level summarization of the general framework is as follows. There are two important mechanisms in this framework: the conversion of sketch to formal and the conversion of formal to sketch. Both of these mechanisms rely on abstract syntax computations, which are essential for maintaining consistency between the two views.

Sketch to Formal. After a syntactic item has been sketched, it is sent to a recognizer for classification; the chosen recognizer may or may not use the existing syntax in the diagram for this classification. After classification, the sketched item is converted into a corresponding formal item in the other view in such a way that the abstract syntax is preserved. This ensures that the two parallel views are consistent.

Formal to Sketch. By contrast, the creation of a formal item does not require recognition to be performed: the act of creating such items involves the user specifying the type through the user interface. The procedure is then similar to that of sketched to formal, where abstract syntax is calculated and a sketched version is generated while maintaining consistency.

We discuss abstract and concrete syntax in Section 3.1. Approaches for converting automatically between the two views are described in Section 3.2. Lastly, Section 3.3 presents a method for rectifying differences that can arise between the sketched and formal diagrams as a result of a new item being added or another editing action being performed.

3.1. Concrete and Abstract Syntax

Typically, the concrete and abstract syntax capture what constitutes a well-formed statement or formula. The abstract syntax embodies the semantics of the diagram. Displaying the abstract syntax on the user interface allows the user to determine the semantics of the drawn diagram. Thus, users are able to edit the diagram if the semantics are not those required. A key feature of tools to support diagram creation should be their ability to compute the abstract syntax of both sketched and formal diagrams; we discuss abstract syntax computation in Section 3.3.

Inevitably, the act of drawing will include the construction of incomplete (not well-formed) diagrams. For example, consider class diagrams that comprise rectangles and arrows that connect rectangles. Their concrete syntax will include a set of rectangles and a set of arrows [Unified Modelling Language 2006]. In Figure 2, the diagram is not a well-formed class diagram (because of the arrow with no target) and, therefore, does not conform to the definition of the concrete syntax [Baar 2006a]. We should not delay computing the abstract syntax of the diagrams until the user has finished: any inconsistencies between the sketched and formal diagrams may not then be apparent for some time. We need to generalize previous definitions of concrete and abstract syntax, such as that in Howse et al. [2005], to capture partly formed diagrams. This is somewhat different from the usual approaches to defining abstract syntax, which tend to capture complete (or well-formed) diagrams. We demonstrate our more general approach via our case study, where we define an abstract and concrete syntax of augmented Euler diagrams that allows any of their basic syntactic items (i.e., their building blocks: closed curves, labels, nodes, and edges) to be drawn without requiring the diagram to be complete.

3.2. Conversion

Every time the user makes a change (addition or edit) to a diagram in either sketch or formal view, the corresponding change must be automatically reflected in the other view. For example, in Figure 4, a transition is added to the formal Petri net, which

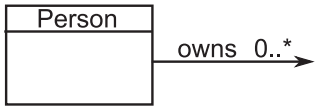

Fig. 2. An incomplete class diagram.

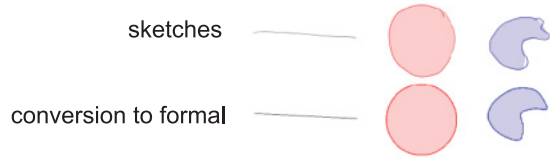

Fig. 3. Converting sketches to formal items.

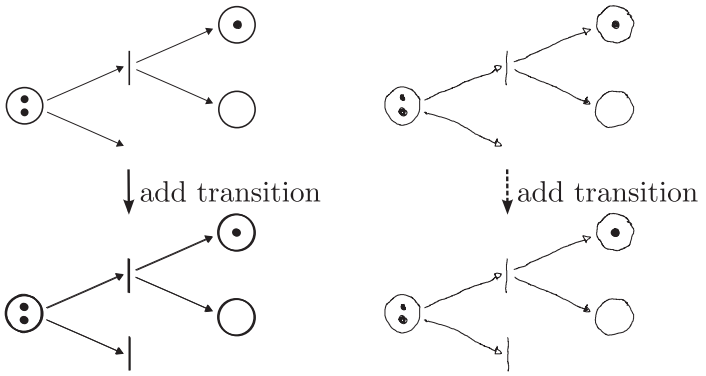

Fig. 4. Adding a formal item to a Petri net, converting to a sketch.

can then be automatically converted to a sketched transition, shown on the right. Likewise, when removing an item from one view, the corresponding change needs to automatically occur in the other view. An example is given in Figure 5, which shows an item being removed from a (partially formed) sketched statechart, with the formal version being updated.

Consider first the addition of a sketched item by the user, which needs to be replicated in the formal view. The beautification process makes the sketched item look as if it was created in a diagram editor. The way in which this is done is dependent on the nature of the syntactic item. For instance, a sketched "straight line" can be converted to an actual straight line. Sketched geometric shapes, such as circles, can likewise be replaced by actual geometric shapes. More general syntactic items, such as sketched closed curves, require a more complex approach to beautification, such as the application of a smoothing procedure. An example for each of these three cases can be seen in Figure 3. In order to maintain two parallel visualizations, sketch and formal, we need methods to render sketched items as formal items and vice versa.

This subsection describes techniques that can be used for the conversions. Given the numerous types of syntactic items, it is unrealistic for us to describe techniques that can be used to convert them all. Thus, we focus on what we see as primary syntactic elements that form a core part of many visual languages. We describe methods to convert text, straight lines, geometric shapes (e.g., circles, rectangles, etc.), and arbitrary curves (wiggly lines) that could be closed. Each part has its own challenge: when moving from sketch to formal, one has to ensure that the result is smooth or regular, for instance, and when moving from formal to sketch, the automatically produced result has to look hand-drawn. The conversion methods that we describe inevitably do not preserve the geometry and, thus, can introduce differences between the abstract syntaxes of the sketch and formal diagram. Section 3.3 presents techniques that can be used to ensure consistency.

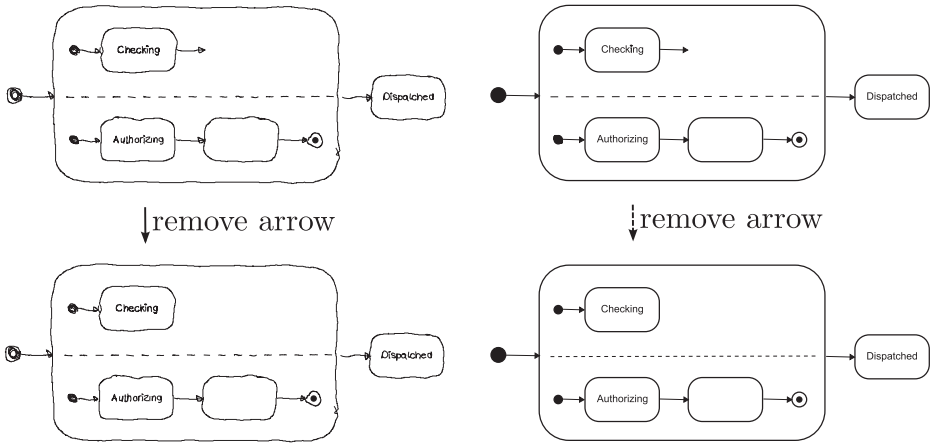

Fig. 5. Removing a sketched item to a statechart and automatically updating the formal view.

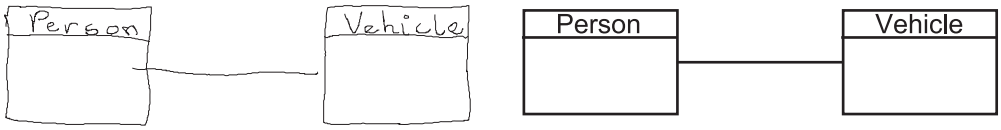

Fig. 6. Converting a sketched straight line.

3.2.1. Sketch to Formal. The conversion of sketched items to formal items assumes that the sketched items have been classified by a recognition engine, and we acknowledge that this in itself is a challenging research problem. Here we do not address the problem of segmentation or the identification of how many strokes form a single syntactic item. The framework we present assumes this recognition task has been completed. Our focus is on conversion and consistency.

The conversion of text is, at least at an implementation level, easy, since it can be sent to an off-the-shelf character recognizer. For straight lines, one can simply compute their start and end points and re-create an actual (formal) straight line in the formal view between these two points. There may be some fine-tuning required, depending on the visual language. For instance, if the line is intended to connect two items together, then some adjustment to the length of the line or position of the endpoints may be required. This is illustrated in Figure 6, where a line has been sketched between two rectangles, in an incomplete class diagram, that will form part of an arrow between them, representing an association. The endpoints of the line do not perfectly meet the rectangles. One end of the line needs to be trimmed and the other extended. The result is shown in the formal diagram.

Geometric shapes, such as circles, ellipses, triangles, rectangles, and regular polygons, can be realized in the formal view in a relatively straightforward manner. Our approach is to find a bounding box for the sketched item and use the center point of that bounding box as the center point of the formal shape. The bounding box of the sketched item then becomes the bounding box of the formal item. This is illustrated in Figure 7, where the sketched octagon is enclosed by a bounding box, with the formal octagon shown on the right. Of course, there is still some computation to perform in order to establish the lengths and endpoints of each of the polygon's edges. In practice, such algorithms would need to be developed specifically for the application and visual language in question. However, we have given an approach that can be tailored to suit

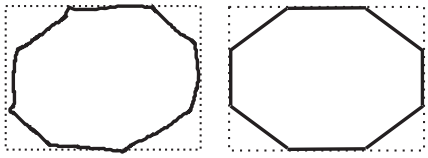

Fig. 7. Converting a sketched octagon.

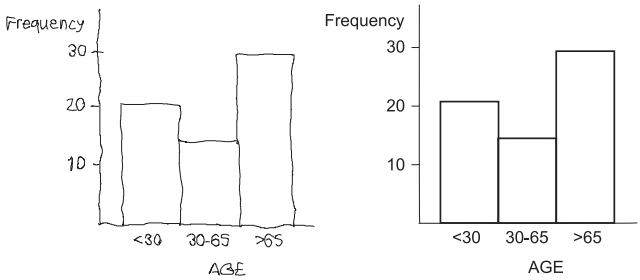

Fig. 8. Converting a sketched bar chart to a formal version.

specific circumstances. The basic idea just described generalizes the techniques we used for circles and ellipses in Wang et al. [2011].

Another example of conversion from sketch to formal can be seen in Figure 8. In this case, as the areas of the bars and the relative positioning matter, using the bounding box of each bar (rectangle), alongside the requirement that bars do not overlap, for the conversion seems sensible.

Arbitrary curves, including closed curves that are not specific geometric shapes and nonclosed curves that are not intended to be straight lines,[2] can be treated in much the same way as each other, although the specific details will again depend on the visual language in question. When these items are drawn in an editor, they are typically Bezier curves, which have a smooth appearance. A process was described in Wang et al. [2011] for smoothing sketched closed curves, which we now summarize here. First, many of the points on the curve (stored as a sequence of points via digital capture) are removed from the internal representation because, typically, digital capture is very detailed and thus includes many unintended wiggles. Only those points where the difference in the x or y value, as compared to x or y value respectively of the neighboring points, is greater than a given threshold are retained; setting this threshold to 1,000 himetric units, a value derived from experiments, is effective. A smooth curve can then be generated that travels through every point in the input array. The result is a smoother curve where sharp changes in direction are removed without changing the path of the curve too much. Figure 9 shows examples of curve smoothing using this technique. If the curve is not intended to be closed, then the smoothing process is complete and we can either preserve the position of the endpoints or adjust them depending on context, much as was the case for straight lines. The correct action to be taken depends on the syntax of the visual language in question and the arrangement of the other items already drawn.

Sketched closed curves are unlikely to have endpoints that coincide exactly. The ends of the stroke (or strokes, if more than one stroke is recognized as part of a curve) need to be joined to form a closed curve. Here, either the stroke crosses itself, there is a gap, or the stroke overlaps itself without crossing. Figure 10 shows an example: in the

[2]For example, when creating graphs, a user may sketch an edge (which is a nonclosed curve) that is intended to be formally rendered as a line composed of straight line segments.

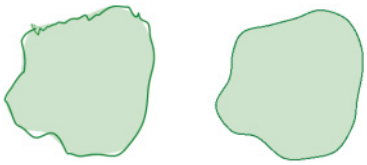

Fig. 9. Smoothing closed curves.

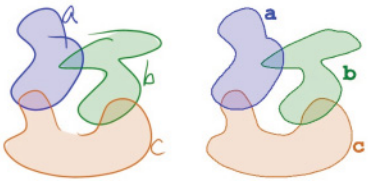

Fig. 10. Sketched closed curves and formal closed curves.

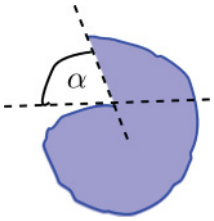

Fig. 11. Joining closed curves.

sketch, a crosses itself, b is overlapping, and c has a gap. In the first case, where the stroke crosses itself, the ends can be trimmed to the intersection point. When the curve does not cross itself, to determine which of the other two cases exists, we draw a line between the beginning and endpoints and another line between the beginning point and the second point on the curve (i.e., point 2), as shown in Figure 11. If α is greater than 90 degrees, we assume the ends do not overlap and can simply join the start and the end point using a line. In the overlapping case, we delete all the "overlapping points" and connect the new endpoints to close the curve. The formal diagram in Figure 10 shows the result of applying these operations to the sketch.

3.2.2. Formal to Sketch. For this conversion process, we take a user-created formal diagrammatic component and produce a visualization that looks as though it was hand-drawn. Any formal shape, including text, rectangles, ellipses, and lines, can be effectively converted to a sketch-like version by one of two methods: either using a library of hand-drawn predefined shapes or perturbing the existing shape.

Using a library of predefined shapes requires the prior collation of actual sketched examples. Libraries have been shown to work effectively for predefined shapes such as text, straight lines, and geometric shapes [Plimmer et al. 2009]. It is possible to ensure a similar placement and size to the formal shape, scaling as appropriate, in the sketch view. For example, in the case of geometric shapes such as rectangles and ellipses, an approach is to ensure the center points are the same and the bounding boxes are the same in the sketch and formal views once a sketched shape has been selected from the library.

An illustration of this process is shown in Figure 12. The geometric shapes (trapezoids, rounded rectangles, and parallelograms) can be chosen from a library of predrawn examples, as can the text, lines, and arrows. The relative positioning, while not essential to the semantics, can be preserved for readability—the top-down "flow" could be seen as important. In fact, this is a feature of the method described, since the center points and bounding boxes of shapes are the same across views.

Alternatively, a shape created in the formal view can be converted to a simulated sketch by perturbing the formal shape. This is achieved by subdividing the formal curve and introducing small perturbations at the start and end of the subdivided

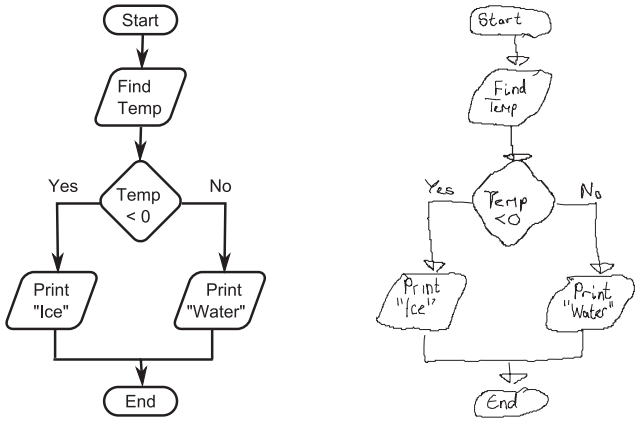

Fig. 12. Converting a formal flowchart to a sketched flowchart.

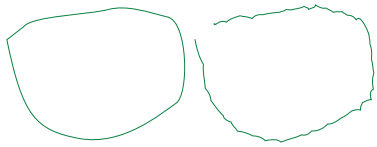

Fig. 13. Converting a formal curve to a sketched curve.

Fig. 14. Mismatched syntax between sketch and formal diagrams.

segments. This technique is effective for arbitrary shapes that do not lend themselves to classification in a library, such as general curves. For closed curves, the start and end points can also be offset. In this way, the start and end are disconnected and may result in a crossing between the start and end line segments. An example of a formally drawn closed curve, drawn using mouse-based interaction in a formal diagram creation tool, can be seen on the left in Figure 13 and its conversion into a sketch is provided on the right. The images deemed most *sketch-like* by the authors had small perturbations of approximately 0.1% of the area of the output viewport. Furthermore, as our implementation of a curve is a Beziér spline, we found that subdividing each Beziér curve two to three times was sufficient. Validating these observations with an empirical study remains a future line of enquiry.

3.3. Consistency

To fully support diagram creation, we need to check that the sketch and formal views are consistent and, where possible, automatically rectify any differences. For instance, the sketched graph on the left of Figure 14 has an edge incident with the vertices r and t, whereas this edge is incident with vertices r and s in the formal view. In Figure 15, the two Euler diagrams on the left are consistent (informally, this is because the overlaps between the curves are the same). However, after the formal diagram is edited, by stretching the circle labeled b into an ellipse, an inconsistency is introduced: there is an overlap between the curves b and c in the formal diagram that is not present

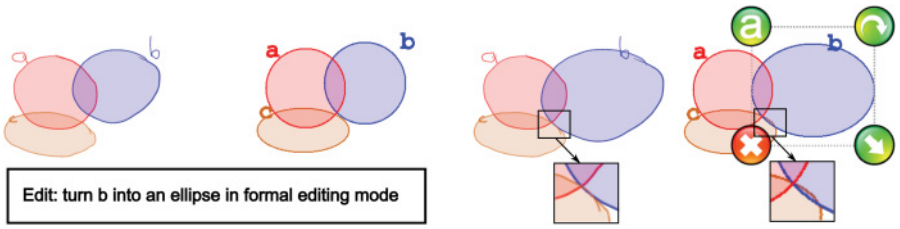

Fig. 15. Mismatched syntax between sketch and formal diagrams.

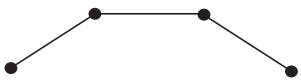

Fig. 16. A concrete graph.

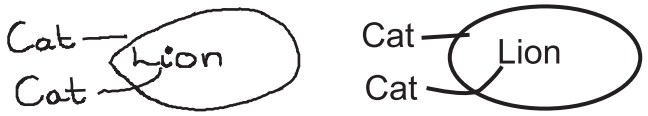

Fig. 17. Changing the meaning on conversion.

after the corresponding transformation is applied. Because of the imperfect nature of sketched elements, the conversion process may not preserve the diagram's structure.

Here we give a general overview of the processes required to maintain consistency between sketched and formal versions. These are not intended to be algorithms as they will vary depending on the visual language under consideration. In Section 4 and onward, we give specific algorithms in the case of Euler diagrams and graphs.

3.3.1. Checking for Consistency. For diagram creation tools to be able to check for consistency between the sketched and formal diagrams, we compute and compare the abstract syntax of each concrete diagram. Again, the abstract syntax has to cope with partly formed diagrams (for instance, when an edge of a graph does not yet have a vertex at one end), since users will draw them en route to creating their final diagram. The algorithms should update the abstract syntax of both the sketch and formal diagram every time an edit is made to allow on-the-fly consistency checking to be performed.

Given a formal diagram, where the type of each basic syntactic item is likely to be known, we only need to develop an algorithm that turns the concrete diagram into the abstract syntax. As a brief example, the (formal) graph, G, in Figure 16 can be converted to its abstract syntax by first noting that there are four vertices, giving $V(G) = \{v_1, v_2, v_3, v_4\}$ say, and giving three edges, $E(G) = \{e_1, e_2, e_3\}$ say. Next, we need to establish the vertices incident with each edge; one way of capturing this is via a function, σ, that maps edges to incident vertices. Here, assuming e_1 arose from the leftmost edge and v_1 and v_2 are the leftmost vertices, we have $\sigma(e_1) = (v_1, v_2)$. The other edges and vertices can be matched up in a similar way. The abstract syntax of G is thus $V(G)$, $E(G)$, and σ. For sketched items, they must first be sent to a recognition engine for classification, and then the same process as for formal diagrams can be followed.

3.3.2. Ensuring Consistency. To *ensure* consistency between the sketch and formal diagram, we need to be able to automatically rectify any differences that are detected at the abstract syntax level. The manner in which this is achieved is linked with the conversion process employed and the syntactic requirements of the diagrams (i.e., the definition of the concrete syntax). For instance, if lines are used to connect other syntactic items, then the conversion of a sketched line to a formal line should ensure that the endpoints are incident with the "same" items in each view. In addition, further management of the conversion process for the line may be required, such as to avoid crossing other items where this conflicts with the requirements of the diagram syntax.

We illustrate this with Peirce's existential graphs [Peirce 1933], which use closed curves, lines, and text to represent negation, the existence of individuals, and predicate symbols, respectively, with numbers giving the order of inputs to *n*-ary predicates. Figure 17 gives an example where the sketch expresses that there is an individual that is a cat (the top "Cat" has a line hitting it) and an individual that is a cat but not a lion (the other line hits "Cat" and "Lion," with the latter being enclosed by a curve to represent "not"). In the formal diagram, the meaning is different, since the ellipse encloses the end of the top line; the diagram means there is an individual that is not a cat and an individual that is a cat but not a lion. Clearly, these diagrams are not, therefore, semantically equivalent; see Dau [2007] for a more complete description of existential graphs.

Differences between the abstract syntaxes can arise in two cases: (1) when a new item is added (sketch or formal) or (2) when an edit is made. In case (1), the inconsistency arises because of the conversion process, when changes are necessarily made to the geometry of the syntactic item. When converting syntactic items, the problems that can arise are more complex when spatial relationships must be preserved than when connectivity is to be preserved. In case (2), every time an edit is made, such as the addition of a syntax item, erasure of an item, or moving or scaling of an item, the abstract syntaxes of the sketched and formal diagrams need to be recomputed and compared. We can view the edit as a deletion followed by an addition. Thus, we only need algorithms to compute the abstract syntax after these two types of edits. In either case, some geometric changes need to be made in order to rectify the differences.

The way in which these changes are effected is dependent on the semantics and conventions of the visual language being considered. For instance, suppose that a sketched item has been recognized as a square and that this shape is aesthetically desirable rather than of semantic importance. Further suppose that it could also have been recognized as a rectangle without changing the diagram's semantics. Then, changing the recognized type accordingly and reconverting it may lead to a rectification of the syntax differences. However, making alterations to just a single syntactic item is not necessarily sufficient. Additionally, in order to preserve the user intention, the syntactic item that the user added or edited should not be changed.

At a high level, our approach to rectifying differences is as follows. Identify the user-sketched syntactic items and user-created formal syntactic items involved in the discrepancy and iterate through the corresponding set of automatically created items. Make modifications to each automatically created item in turn until the differences are rectified or until each item has been considered; this can be made more general, where we continue to iterate through, making changes in different combinations, but here we keep the description simple to convey the spirit of the approach. The nature of the modifications is described in the next two subsections. Ideally, these modifications will resolve the discrepancy. There is no approach that is guaranteed to eliminate all differences, other than making both the sketch and formal diagram identical. This loses the benefits of having two modes of input and visualization. In our case study, if differences cannot be rectified, then the user is informed and has the opportunity to modify the diagrams manually. We now present some general approaches to automatic modification in each view for the previously considered types: straight lines, geometric shapes, and curves.

Sketched Items Converted to Formal Items Involved in the Discrepancy. Here we focus on items that the user created in sketch view (i.e., with stylus-based input), so we modify their formal versions. First we consider the case when a **sketched straight line** is involved in the discrepancy between the abstract syntaxes. If the line is intended to connect two already drawn items, then it is trivial to ensure the connectivity matches across both views: our conversion technique specified that connectivity must match,

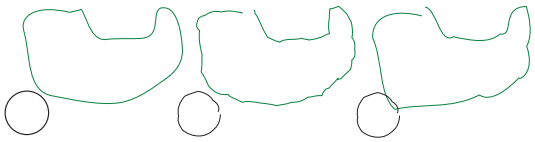

Fig. 18. Iterative refinement of sketch conversion.

so this cannot be the source of a difference. This applies to all other syntactic items involved in connectivity relationships too, so for the remaining cases we focus only on spatial relationships. The spatial case is much more challenging, and appropriate methods to resolve differences are likely to be specific to the visual language in question and could well tailor the approaches we present here.

Suppose that a **sketched geometric shape** is involved in the discrepancy. If the shape is not of semantic importance, then we can attempt to iteratively change the type of the sketched shape and reconvert it to a formal shape. Again, the details of how this is done will be dependent on the visual language in question, and we propose that defining a hierarchy of shapes (e.g., by user preference) for the retyping step is one part of this approach. If the shape cannot be altered, then one can attempt resizing and moving the center of the shape in small steps to see whether this resolves differences.

For the case of **sketched curves**, which are converted to smoothed curves, the smoothing process can be the source of the error. In the smoothing process, roughly speaking, the more points that are removed from the sketched curve, the more likely there is to be a difference in the abstract syntax: the formal curve's geometry will have deviated further from the sketched curve's geometry. To restore more of the original geometry, we can resmooth by removing fewer points, iteratively, until differences are rectified. This will, however, result in curves that are not as smooth.

Formal Items Converted to Sketched Items Involved in the Discrepancy. Since the user created the formal item, we should know the intended type (e.g., he or she added a circle) so it would not preserve the user's intention to change the type and attempt to reconvert to a different type of sketched item. In the cases where we have used a hand-drawn library to create the sketched item, we iteratively choose different hand-drawn examples of the correct type from the library to see whether that rectifies the differences. Effectively, this simply reconverts the formal item to a new sketched item. The likelihood of this rectifying differences will be partly dependent on the size of the library and the diversity of sketched items within it.

When a library is not used or cannot rectify inconsistencies, our approach that converts a formal curve to a sketched curve allows iterative refinement of the formal curve into an appropriate sketch, with the result possibly appearing less like a sketch and more like a formal curve. A series of refinements of a drawn closed curve can be seen depicted left to right in Figure 18. These three diagrams illustrate a formal curve and a formal circle (leftmost diagram), which have been converted to sketches in the middle and leftmost diagrams. In the rightmost diagram, the formal curve has been changed to such an extent that the topological structure of the diagram has been altered. Iteratively altering the conversion of this curve allows us to correct this difference, shown in the middle diagram. In fact, the sketched curves in the middle diagram represent an instance between the two examples that both appears to be more hand-drawn and maintains the user's intended semantics for the diagram.

4. CASE STUDY: EULER DIAGRAMS AUGMENTED WITH GRAPHS

We demonstrate how to apply the general framework to Euler diagrams augmented with graphs. First we describe the diagram creation and editing interface and functionality of our tool. Then we go on to define the concrete and abstract syntax of these

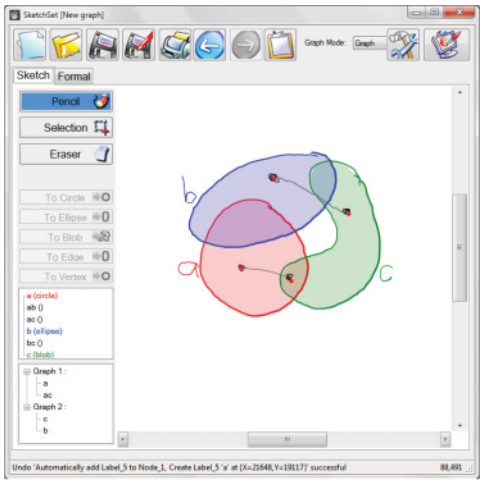

Fig. 19. The sketching interface of our tool.

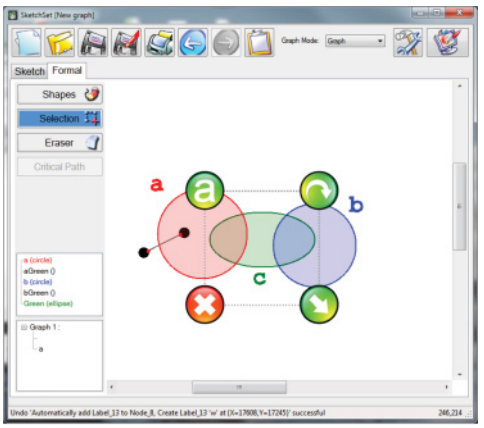

Fig. 20. Editing in the formal interface.

diagrams, including the case when they are only partly formed. The application of the conversion techniques is subsequently demonstrated and the approach to rectify differences is explained. Last, we present two user studies that evaluate our tool. The first study looks at the user interaction with the tool and the second study stress-tests the system, to establish the robustness of the consistency-preserving mechanisms.

4.1. Diagram Creation and Editing

This section describes the process of creating sketches and formal diagrams along with the editing functionality provided within our tool. Our code base uses SketchNode [Plimmer et al. 2010] as its starting point. There are two interfaces for diagram creation: a sketching interface and a formal interface. In the sketching interface, users create sketches by drawing as on a piece of paper; Figure 19 shows a screenshot. A stylus stroke is immediately rendered on the canvas as digital ink. On completion of the stroke (stylus up event), it is passed to a gesture recognizer that we have generated using RATA.Gesture [Chang et al. 2012]. RATA employs a training-based approach to generate customized recognizers for any 2D gesture set. The underlying algorithm for RATA.Gesture is the Vote ensemble from Weka [Hall et al. 2009], which is used to combine individually tuned algorithms from Weka: Bayesian Network, Logit Boost, Logistic Model Trees, Random Forest, and Multilayer Perceptron. Evaluations of RATA.Gesture suggest that users can expect recognition rates of approximately 95% depending on the number of different classes being recognized and the differences between classes; we detail our recognition rates in Section 4.6.

The recognizer for SketchSet 2.0 is generated using the RATA toolkit via the RATA API. The first step is to collect realistic sketches using the data collection component [Blagojevic et al. 2008]. Realistic sketches are important as they provide natural intergesture spatial and timing information that is beneficial for sketch recognition [Field et al. 2009]. For optimal recognition results, examples drawn by five or six people are needed, with each person drawing at least four examples of each syntax type. After the data is collected, manual classification of the gestures (the diagram's syntax) is required for the training algorithms; this is done in the *labeller* interface. We asked seven participants to each sketch four diagrams in order to provide training data.

The recognizer result is one of six classes: text; one of the three curve classes, namely, circle, ellipse, or closed curve; the vertex class; or the edge class. When text is entered, it is taken to be a label associated with the closest unlabeled curve. Curves are immediately assigned a color, selected in sequence from a list of 16 colors, and filled with a lighter shade of the same color. Curve labels have the same color as their associated curves. Vertices are immediately colored black and edges are colored grey. When edges are associated with vertices, this is visualized by the use of a red dot.

This immediate coloring of sketched items and association of labels with curves and vertices with edges allow the user to readily check the recognition result. Misclassifications can be corrected by selecting the appropriate stroke in edit mode and tapping on the button for correct classification in the left-hand panel of the user interface; for example, if an ellipse is misclassified as a closed curve, then the user selects it and reclassifies it appropriately. The process of label assignment is performed online, while the user is creating the diagram, and adjustments are made as the user performs edits. For instance, if a curve is erased but its associated label remains, then that label may be reassigned to the next closest curve or assigned to no curve if every remaining curve already has a label. The formal interface provides similar functionality. Based on experience with SketchNode [Plimmer et al. 2010], care has been taken to design simple interactions that are easy to perform with a stylus. A single tap on the canvas creates a circle and opens a textbox for typing the label, which is placed at the top left corner of the circle's bounding box. Vertices are added by tapping while the pen barrel is button down or holding the pen down for an operating-system-defined time. Edges are created by dragging the pen along the edge path. Circles can be stretched to form ellipses and selected items can be resized, moved, and rotated as in Figure 20. The usual editing functions are supported in both interfaces: add item (by sketching or tapping), erase, move, and resize. When strokes are moved or resized, curve labels are moved with their curves and edges resized so that they remain connected to incident vertices. After any editing event, coloring is updated as appropriate. Progressive undo and redo are available by tapping the buttons.

4.2. Concrete and Abstract Syntax

The abstract syntax is displayed on the user interface, allowing the semantics of the diagram to be determined. Thus, users are able to edit the diagram if the semantics are not those required. We argue that one should display the abstract syntax for both the sketched and formal diagrams. An example is in Figure 19, where the abstract syntax is displayed in the two bottom left panels for the Euler diagram and graphs, respectively.

The remainder of this section has two main goals: to define the concrete syntax of augmented Euler diagrams and to define their abstract syntax. The concrete syntax embodies the actual drawn diagram and, we note, a sketched diagram and its rendering in the formal panel will have different concrete syntaxes: their geometric properties will be different. The abstract syntax affords our tool with an internal representation of the diagrams. This allows the tool to determine whether the sketch and formal panels are displaying diagrams with the same abstract syntax.

The basic syntactic items of (concrete) Euler diagrams augmented with graphs are closed curves, labels, and graphs that are formed from vertices, and edges. Any finite collection of them can be entered by a user when creating a diagram. One kind of incompleteness that can arise when drawing a diagram is that an edge can be drawn for which an incident vertex is not present. An example can be seen in Figure 21, which shows a sequence of diagrams created by a user, where the fourth diagram has an edge with only one incident vertex. The diagram becomes well formed again at the next step,

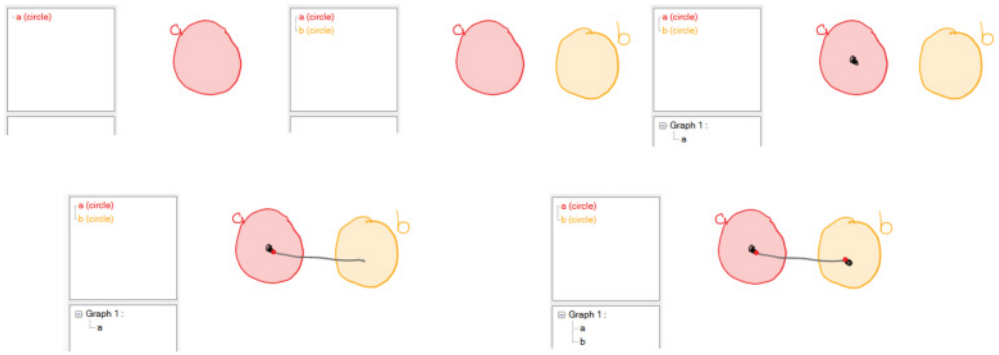

Fig. 21. A sequence of sketches including a diagram that is not well formed.

when the "missing" vertex is drawn. Another way in which non-well-formedness can arise is through labels that are not associated with any curve.

We first extend the standard (abstract) graph syntax to allow this kind of syntactic construction to occur. In particular, we define the notion of a partial graph. In our definition, each edge will be mapped to the pair of vertices, (v_1, v_2), with which it is incident, should they exist. When an edge has an endpoint that is not incident with any vertex, the pair will include the underscore symbol, _, as a place holder for the missing vertex. Moreover, if an edge is incident with one vertex, v_1, but the other end of the edge is not incident with any vertex, then the underscore symbol, _, will always be the second element in the pair: $(v_1, \_)$. More formally:

Definition 4.1. A **partial graph**, G, is a triple, $G = (V, E, \sigma)$, where

(1) $V = V(G)$ is a finite set of **vertices**,
(2) $E = E(G)$ is a finite set of **edges**, and
(3) at least one of $V(G)$ and $E(G)$ is not empty.
(4) $\sigma = \sigma_G$ is a function, $\sigma : E(G) \rightarrow (V(G) \cup \{\_\}) \times (V(G) \cup \{\_\})$, where _ is a special symbol that is not in $V(G)$ that maps edges to pairs, $\sigma(e) = (p_1, p_2)$, which ensures that if $p_1 = \_$, then $p_2 = \_$.

If $\sigma(e) = (p_1, p_2)$ and p_1 is a vertex, then e is **incident** with p_1. Similarly, if p_2 is a vertex, then e is incident with p_2. Given an edge, e, in $E(G)$, if $\sigma(e) = (p_1, \_)$, then e is **dangling**. If no edge in $E(G)$ is dangling, then G is a **graph**.

In Figure 21, the partial graph in the fourth diagram is $G = (\{v_1\}, \{e\}, \{(e, (v_1, \_))\})$, where e is dangling. In the last diagram, the partial graph is $G' = (\{v_1, v_2\}, \{e\}, \{(e, (v_1, v_2))\})$. Since e is no longer dangling, G' is a graph. We assume that standard graph theory terminology extends to partial graphs in the obvious manner. For instance, a partial graph is *connected* if *either* every pair of vertices has a sequence of edges connecting them and every edge is incident with a vertex *or* there is exactly one edge and no vertices. As defined, partial graphs are at the abstract syntax level, since they have no geometric information. An **embedding** of a partial graph in the plane is an actual drawing of the graph; that is, each vertex has a position (we treat such vertices as points) and each edge is realized as a line segment. An embedded partial graph is a **concrete partial graph**. To illustrate, consider the partial graph G, where

(1) $V(G) = \{v_1, v_2, v_3\}$,
(2) $E(G) = \{e_1, e_2, e_3, e_4\}$, and
(3) $\sigma(e_1) = (v_1, v_2)$, $\sigma(e_2) = (v_1, v_3)$, $\sigma(e_3) = (v_1, v_4)$, and $\sigma(e_4) = (v_2, \_)$.

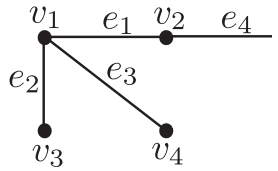

Fig. 22. A concrete partial graph.

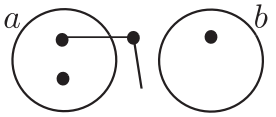

Fig. 23. A concrete Euler diagram augmented with a partial graph.

An embedding of G is the concrete partial graph in Figure 22; the vertex and edge labeling show the correspondence between the embedded vertices and edges with those in G.

We now consider the concrete syntax of an Euler diagram, as this is more intuitive than the abstract syntax. An Euler diagram comprises a set of closed curves (informally, lines, which may be wiggly, smooth, or bendy for instance, that have the same start and end point). The curves may each have a label. The act of augmenting an Euler diagram with a graph involves drawing the vertices inside the regions created by the curves. An example can be seen in Figure 23, where the Euler diagram has two curves, labeled a and b respectively, and there are three vertices and two edges. In this case, one edge is dangling, since it has an endpoint that is not incident with any vertex.

We can now define the concrete syntax of an augmented Euler diagram:

Definition 4.2. An **augmented Euler diagram** is a 4-tuple, $(\mathcal{C}, \mathcal{L}, l, \mathcal{G})$, where

(1) $\mathcal{C}$ is a finite set of simple closed curves drawn in the plane,
(2) $\mathcal{L}$ is a finite multiset of labels,
(3) l is a partial, injective function, $l : \mathcal{C} \to \mathcal{L}$, that assigns labels to curves, and
(4) $\mathcal{G}$ is a set of concrete partial graphs

such that no vertex lies on any curve[3] in $\mathcal{C}$, each partial graph in $\mathcal{G}$ is connected, and no two distinct graphs in $\mathcal{G}$ share a vertex or an edge. If l is both total and surjective and all partial graphs in $\mathcal{G}$ are graphs, then d is **well formed**.

Given Definition 4.2, the set of partial graphs, $\mathcal{G}$, in a concrete diagram is uniquely determined up to isomorphism. To identify the set $\mathcal{G}$, one identifies each maximal connected partial graph drawn, G, in the diagram. Each such G is an element of $\mathcal{G}$. So, in Figure 23, we have

(1) $\mathcal{C} = \{c_1, c_2\}$, where c_1 and c_2 are the curves in the diagram,
(2) $\mathcal{L} = \{a, b\}$; as there is no multiple label use, this multiset is also a set,
(3) $l(c_1) = a$ and $l(c_2) = b$; as each curve has a label, the function l is total (every curve has a label), and
(4) $\mathcal{G} = \{G_1, G_2, G_3\}$, where G_1 is the concrete partial graph in d comprising two connected vertices and the two edges, G_2 is the isolated vertex inside b, and G_3 is the isolated vertex inside a.

[3]While we cannot guarantee that a vertex will not be placed as such, because we treat vertices as single points, it is highly unlikely; if a vertex lies on a curve, then our tool takes it to be outside that curve.

Here, the diagram is not well formed as the partial graph G_1 has a dangling edge; the other two partial graphs do not have dangling edges.

Given an augmented Euler diagram, $d = (\mathcal{C}, \mathcal{L}, l, \mathcal{G})$, in the case where $\mathcal{G} = \emptyset$, d is an Euler diagram. In the previous definition, we could have merely stated that $\mathcal{G}$ is a concrete partial graph. In our interface, though, we display each connected component as a graph so that the user can see which parts the tool considered to be connected. As a consequence, we do not have to recompute the connected components after edits to the diagram, since this will be captured by our algorithms that produce the abstract syntax. The price to be paid for this saving is that the details of the algorithms for computing the abstract syntax are sometimes more complex.

The abstract syntax that we provide for augmented Euler diagrams records, for the curves, the spatial relationships between them, rather than any geometric information about location or size. These spatial relationships are encapsulated by the regions in the plane to which the curves give rise. These regions are called *zones* and can be described as being inside some curves but outside the rest of the curves. For example, both the sketch and formal diagram in Figure 24 have five zones described by $\{C_a\}$ (the zone inside only the curve labeled a, which we call C_a), $\{C_b\}$, $\{C_a, C_b\}$ (the zone inside both the curve labeled a and that labeled b, but outside that labeled c), $\{C_a, C_b, C_c\}$, and the zone outside all of the curves. As an abuse of notation, we will write $\{C_a, C_b, C_c\}$ as abc and so forth.

Definition 4.3. Let $d = (\mathcal{C}, \mathcal{L}, l, \mathcal{G})$ be an augmented Euler diagram. A **zone**, z, in d is a maximal set of points in the plane for which there exists a subset, C, of $\mathcal{C}$ such that every point in z is inside all curves in C but outside the curves in $\mathcal{C} - C$. The **description** of z, denoted $des(z)$, is C. The set of zones in d is denoted $\mathcal{Z} = \mathcal{Z}(d)$. Given a vertex, v, in a graph, G, in $\mathcal{G}$, embedded in z, the **location** of v is z and we define $loc(v) = z$.

For example, in Figure 23, there are two vertices located in the zone that is inside the curve labeled a and one vertex located in the zone inside the curve labeled b. In addition, there is also a single vertex located in the zone that is outside both of the curves.

We can now define the abstract syntax of an augmented Euler diagram:

Definition 4.4. Let $d = (\mathcal{C}, \mathcal{L}, l, \mathcal{G})$ be an augmented Euler diagram. The **abstract syntax** of d is a tuple $D = (\mathcal{AC}, \mathcal{AZ}, \mathcal{L}, al, \mathcal{AG}, aloc)$ where

(1) $\mathcal{AC}$ is a finite set of **abstract curves**,
(2) $\mathcal{AZ} \subseteq \mathbb{P}\mathcal{AC}$ is a finite set of **abstract zones**,
(3) $\mathcal{L}$ is the set of labels in d,
(4) $al : \mathcal{AC} \to \mathcal{L}$ is a partial, injective function that labels curves,
(5) $\mathcal{AG}$ is a the set of partial graphs in $\mathcal{G}$ but without the embedding information, and
(6) $aloc$ is a function, $aloc : \bigcup_{G \in \mathcal{G}} V(G) \to \mathcal{AZ}$, that identifies the abstract zones in which the vertices are located.

Moreover, there must exist a bijection, $f : \mathcal{AC} \to \mathcal{C}$, which pairs abstract curves in D with the curves in d such that for all $ac \in \mathcal{AC}$, $al(ac) = l(f(ac))$, in order for D to be the abstract syntax of d. In turn, f must induce a bijection $g : \mathcal{AZ} \to \mathcal{Z}$, which pairs abstract zones in D with the zones in d, where

(a) for all $az \in \mathcal{AZ}$, if $g(az) = z$, then $des(z) = \{f(ac) : ac \in az\}$, and
(b) for all $v \in \bigcup_{G \in \mathcal{G}} V(G)$, if $aloc(v) = az$, then $g(az) = loc(v)$.

Whenever we talk about a diagram and its abstract syntax, we assume that we have access to the functions f and g as just given. The abstract syntax of the diagram in Figure 23 is as follows:

(1) $\mathcal{AC} = \{\kappa_1, \kappa_2\}$ (there are two curves),
(2) $\mathcal{AZ} = \{\emptyset, \{\kappa_1\}, \{\kappa_2\}\}$ (there are three zones: one outside both curves, denoted $\emptyset$; one inside just a, denoted $\{\kappa_1\}$; and one inside just b, denoted $\{\kappa_2\}$),
(3) $\mathcal{L} = \{a, b\}$,
(4) $al(\kappa_1) = a$ and $al(\kappa_1) = b$,
(5) $\mathcal{AG} = \{G_1, G_2, G_3\}$, where
    (a) $V(G_1) = \{v_1, v_2\}$, $E(G) = \{e_1, e_2\}$, $\sigma(e_1) = (v_1, v_2)$, and $\sigma(e_2) = (v_2, \_)$
    (b) $V(G_2) = \{v_3\}$, $E(G) = \emptyset$,
    (c) $V(G_3) = \{v_4\}$, $E(G) = \emptyset$, and
(6) $aloc(v_1) = \{\kappa_1\}$, $aloc(v_2) = \emptyset$, $aloc(v_3) = \{\kappa_1\}$, and $aloc(v_4) = \{\kappa_2\}$.

4.3. Conversion

We now briefly describe how our conversion techniques are applied to augmented Euler diagrams.

4.3.1. Converting Sketch to Formal. Each stroke is recognized immediately as it is drawn, classified into one of six categories (label, circle, ellipse, closed curve, vertex, and edge) by the recognizer, and the conversion process depends on the recognition results:

(1) If an item is recognized as a label, then it is sent to an off-the-shelf character recognizer and converted to a formal (typewritten) label: this formal label is added to the formal panel with the center coordinate of the formal label the same as that of the original stroke.
(2) If an item is recognized as a circle or ellipse, it is converted to a formal circle or ellipse, respectively.
(3) If an item is recognized as a closed curve, then its endpoints are joined to ensure that it is a *closed* curve and it is smoothed.
(4) If an item is recognized as a vertex, then it is converted to a small filled circle, whose center point is the same as that of the sketched vertex.
(5) If an item is recognized as an edge, then it is converted to a straight line segment. The endpoints of the straight line segment are the same as those of the sketched edge, unless one or both of the ends are identified as incident with vertices. In this case, the respective end of the straight line segment is snapped to the vertex.

4.3.2. Converting Formal to Sketch. Currently we use a circle library, an edge library, a filled vertex library, and a text library. In the formal interface, ellipses can only be obtained by stretching circles so no ellipse library is needed. We have not implemented functionality to allow arbitrary closed curves to be drawn in the formal view. The process of converting from formal to sketch can be summarized as follows:

(1) If the user added a label, then choose a hand-drawn label from a library of hand-drawn examples.
(2) If the user drew a circle, then choose a hand-drawn circle from a library of hand-drawn examples.
(3) If the user drew an ellipse, then this was done by distorting an already drawn circle. Thus, a corresponding distortion is applied to the hand-drawn circle.
(4) If the user drew a vertex, then choose a hand-drawn vertex from a library of hand-drawn examples.
(5) If the user drew an edge, then choose an edge of about the right length from the library and offset the sketched edge's endpoints a small random distance from the

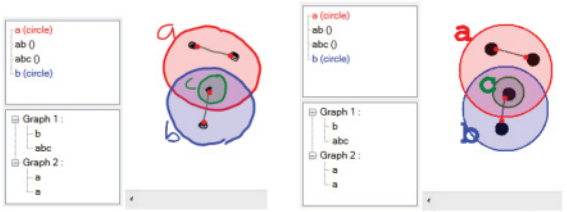

Fig. 24. A sketch with its formal version.

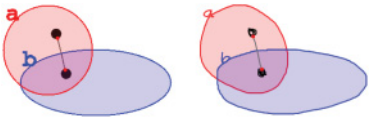

Fig. 25. Converting formal to sketch.

incident vertices (if already drawn) or the actual endpoint (if no incident vertex is drawn).

In all cases, the pseudo sketched stroke is placed on the sketching canvas with the same center coordinates as the corresponding formal syntax. Figure 25 shows a formal diagram on the left and the conversion to a sketch on the right. Here, the "sketched" curves were chosen from the library, as explained earlier, and the curve labeled *b* was formed by distorting a circle that was originally drawn in the formal interface.

4.4. Consistency

We now describe how we check for consistency, going on to show how our techniques for rectifying differences and, thus, ensuring consistency apply to our case study.

4.4.1. Checking for Consistency. Various algorithms have been given to compute the abstract syntax of Euler diagrams (without graphs) from (formal) concrete diagrams. The first approach to computing the abstract syntax was proposed by Clarke [2007], which analyzed the areas of the regions formed by the curves. An alternative approach, by Cordasco et al. [2011], computes the abstract syntax for Euler diagrams that satisfy a fairly limiting set of properties.[4] A slightly quicker version of Clarke's algorithm was detailed by Wang et al. [2011], which we generalize in this article to Euler diagrams augmented with graphs. We now proceed to demonstrate how to compute the abstract syntax of a diagram each time a new item is added or some other edit is made to the diagram. First, we note that when the canvas is blank, both sketched and formal diagrams have abstract syntax $(\emptyset, \{\emptyset\}, \emptyset, \emptyset, \emptyset, \emptyset)$. The abstract syntax of the diagram is computed by Algorithm 1, where we make reference to (sub)algorithms that are included in the OnlineAppendix.

The abstract syntax, once computed, is displayed on the interface in a stylized form, as seen in Figure 24. The zones are listed and the graphs are listed by way of displaying the zones in which their vertices are placed. Other information, such as which edge is incident with which vertex, is visually displayed in the editing panel (for edges and vertices, a red dot indicates an incidence relationship).

4.4.2. Ensuring Consistency. In augmented Euler diagrams, the spatial relationships between sketched items are fundamental to their semantics and are embodied in the abstract syntax. In particular, it is important to maintain the relative intersection, containment, and disjointness of the curves and the location of the vertices during conversions between sketched and formal diagrams. Rather than present in detail

[4]The properties are as follows: the curves must all be simple, there must be no triple points of intersection, the so-called *zones* must be connected, there must be no concurrency between curves, and no label can be used on more than one curve. The exact details of these properties is not of relevance to this article; the interested reader should see Stapleton et al. [2007]. The limitations of this approach imply that it is not appropriate for applying in sketching environments where users are free to draw diagrams that need not conform to strict conditions.

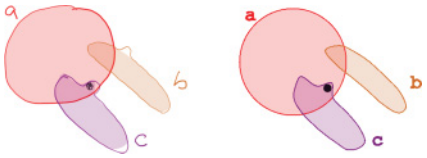

Fig. 26. Smoothing curves in the context of vertices.

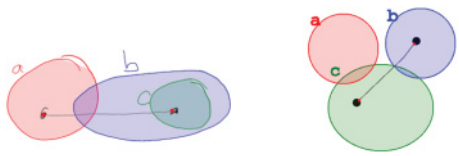

Fig. 27. Creating diagrams in the study.

ALGORITHM 1: Compute Abstract Syntax of the Sketch and Formal Diagrams

Input: A sketched diagram, $d_s = (\mathcal{C}_s, \mathcal{L}_s, l_s, \mathcal{G}_s)$, with abstract syntax $D_s = (\mathcal{AC}_s, \mathcal{AZ}_s, \mathcal{L}_s, al_s, \mathcal{AG}_s, aloc_s)$. A formal diagram, $d_f = (\mathcal{C}_f, \mathcal{L}_f, l_f, \mathcal{G}_f)$, with abstract syntax $D_f = (\mathcal{AC}_f, \mathcal{AZ}_f, \mathcal{L}_f, al_f, \mathcal{AG}_f, aloc_f)$. An edit action that is either the addition of an item of syntax, s_a, or the erasure of an item of syntax, s_e.

Output: Abstract syntax $D'_s = (\mathcal{AC}'_s, \mathcal{AZ}'_s, \mathcal{L}'_s \cup \{\lambda\}'_s, al'_s, \mathcal{AG}'_s, aloc'_s)$ and abstract syntax $D'_f = (\mathcal{AC}'_f, \mathcal{AZ}'_f, \mathcal{L}'_f \cup \{\lambda\}'_f, al'_f, \mathcal{AG}'_f, aloc'_s)$;

(1) If the action is an addition, then call one of algorithms 2 to 5 as appropriate, once with inputs d_s, D_s, and s_a and once with d_f, D_f, and s_a. Set D'_s and D'_f to be the outputs of these respective calls to the appropriate algorithm.

(2) Else the action is an erasure, so call one of algorithms 6 to 9 as appropriate, once with inputs d_s, D_s, and s_e and once with d_f, D_f, and s_e. Set D'_s and D'_f to be the outputs of these respective calls to the appropriate algorithm.

the application of the techniques to Euler diagrams (which would somewhat repeat the previous text), we focus on the case when a sketched item has been added and converted to a circle or an ellipse, since this is the most likely occurrence of a mismatch before any corrective action is taken. When there is a mismatch, we convert the newly sketched item to a closed curve and check whether the abstract syntaxes match; we could have first changed a circle to an ellipse. If they still do not match, then we iterate through the curves involved in the zone error, modifying them using the methods in Section 3, and rechecking the zones and vertex locations. As part of this process, when closed curves are smoothed, we ensure that we do not alter vertex containment. If the error is not corrected, then an error message is displayed on the interface, alerting the user to the differences.

An example is in Figure 26, where the sketched curves labeled b and c are very similar in appearance. In the formal diagram, though, b is smoothed more than c, with the bulge being removed in this case. The similar bulge in c cannot be smoothed out, since this would leave the vertex outside c in the formal diagram but inside c in the sketch.

4.5. Evaluation

To demonstrate whether the development of diagram creation tools using our general approach produces effective, usable systems, we conducted two studies: a task-based usability study to ensure that our tool was usable and a stress test to validate its effectiveness. For the first study, participants were asked to complete five tasks with Euler diagrams augmented with graphs. The first two tasks asked them to create diagrams, using circles, in the sketch and formal interface, respectively. The third task tested moving diagram elements. The fourth task asked for input of closed curves. The last task focused on deleting elements. Information was gathered from observation and a questionnaire. In the second study, participants were given the freedom to draw

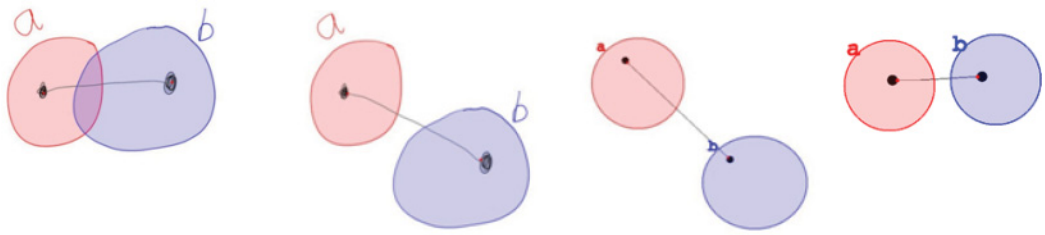

Fig. 28. A sequence of tasks: draw initial diagram, move the curve b in sketch view, switch to formal view, and then move b back in line with a in the formal view.

Fig. 29. Testing the entry of closed curves.

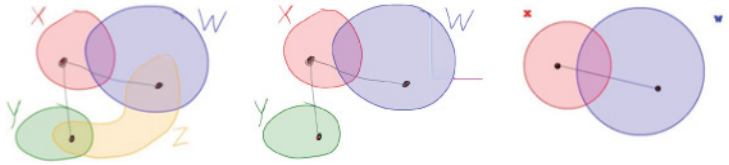

Fig. 30. A sequence of tasks: draw initial diagram, erase the curve z in sketch view, and erase the curve y in formal view.

diagrams of their choice, with a large number of curves, vertices, and edges. We now describe the methodologies used for the studies and their results.

4.5.1. Usability Study Methodology. The 10 participants (eight male, two female) aged between 18 and 21 years had varied backgrounds including software engineering students (seven), computer science students (one), and other engineering students (two). All had basic knowledge of set diagrams, but not necessarily of Euler diagrams augmented with graphs. Each participant undertook the study individually, and we captured the screen activity for later review. As our goal is an excellent user experience, we planned to observe enough participants to show any major flaws, fix these, and then continue with the study.

The study started with the experimental facilitator showing the participant how to create and edit diagrams in both views. Participants were then asked to familiarize themselves with the tool by creating a simple diagram in each view. For the initial part of the study, the participants were then asked to reproduce the diagrams in Figure 27, with task one being to create the sketched diagram shown on the left and another task being to create the formal diagram shown on the right.

The third task tested moving curves, with participants asked to create the diagram shown on the left of Figure 28 and move curve b away from a, switch to formal view, and move curve b back in line with a. This task tests the selection of components and the naturalness of the edge reflow when edges must be moved, lengthened, or shortened.

The fourth task was designed to test the input of closed curves. For this, users were asked to draw the sketch in Figure 29. The final task tested erasing. The participants had to first draw the diagram in Figure 30 and then erase curve z, then switch to formal view and erase curve y and the associated vertex and edge. After the tasks were completed, the participants filled in a qualitative questionnaire.

4.5.2. Usability Study Results. After six participants completed the tasks, we reviewed the results and made a number of minor refinements to the interfaces and then continued with the last four participants. The refinements after the first participants were as follows.

Fig. 31. The new formal view cursor and the new menu on the formal canvas.

When participants moved from the sketch view to the formal view, three participants tried to use the pencil tool to scribe a curve (where a tap creates a curve). We made two changes to the interface to address this. First, we changed the cursor to a significantly different custom-designed cursor and changed the label on the mode button to say "Shapes" (as opposed to "Pencil"); the new versions can be seen in Figure 31.

Our tool is specifically designed for pen-only interaction. However, four people tried to use the delete button to erase selected shapes in the formal view. We added this support to the formal view but not to the sketch view, reasoning that the eraser is more natural in sketch view. Both views (sketch and formal) have three interaction modes (draw, select, and erase) and it remembers the mode for each view independently: when switching between views for the moves and deletes, users expected the mode to follow. We have made that modification.

Drawing graphs caused problems in both views. In the sketch view, three participants drew at least one vertex and edge in a single stroke. In the formal view, both of the pen interactions, press and hold or tap with the barrel button down, are difficult. These problems are more difficult to solve. A possible solution to the first problem would be to segment the ink stroke into vertex and edge; however, segmenting strokes is a difficult problem [Sezgin et al. 2001; Wolin et al. 2011] and is likely to be error prone. As for the second problem, timeouts and stylus barrel buttons are both known to be difficult for users [Plimmer 2008]. We cannot assume that a tap within a curve is a vertex because curves can lie within curves. With the limited interaction of a stylus, an alternative is a new mode for adding vertices. However, mode changes can be annoying and adding another mode may also cause problems. We chose not to make any alterations for either of these problems, reasoning that our tool supports the creation of incomplete diagrams and that if users are aware of the expected interaction, they are likely to adapt to it.

There were two issues with deletion. The first was that users expected a curve to be deleted if they tapped inside it with the eraser button: they must erase the curve itself. We noted that users adapted to this interaction quickly and changing to erasing on tap would make erasing very sensitive and likely result in unwanted erasing of vertices. A similar situation occurred in the formal view: when users tried to tap on an edge to erase it, hitting exactly on the edge was extremely difficult. When this did not work, they then dragged the eraser over the edge, which was successful. We expanded the hit zone around the components to make tap delete easier.

We observed that some of the curve labels were recognized as numeric digits rather than letters, so we invoked the character recognizer's factoid function. With this, character recognition is restricted to lowercase letters. We also corrected a number of minor bugs in the software.

After these modifications, we recommenced the user study with another four participants. The only change to the instructions was a specific instruction to draw the edges and vertices in separate strokes. The two changes in regard to view change—the new cursor and retaining active mode across views—were very effective. Only one participant tried to draw in the formal view and there were no frustrations of being in a

different mode when the view changed; we observed that the transition was seamless. The erase problem in the formal view was solved with the larger hit zone with no participant problems. The modifications had the desired effect, with users having fewer problems with these functions, and no other major issues were identified.

The questionnaire asked participants to rate the software on a scale of 1 to 5 (5 being the best) on a variety of features. The responses for all 10 participants were very positive, with all participants rating the software 4 or 5 on overall ease of use (mean 4.6, SD 0.59). For each view, we asked participants to separately rate how easy it was to find buttons, create the diagrams, edit the diagram, and correct the diagram. For the sketch view, the responses were all 4 or 5 for both usability testing rounds. The formal view received seven 3 ratings (out of 24) in the first round, and this improved in the second round with only two 3 ratings (out of 16). We attribute this improvement to the new cursor and easier deletions. All participants were satisfied with the time and ease of completing tasks with scores of 4 or 5 (time (mean 4.8, SD 0.42), ease (mean 4.8, SD 0.42)). The first set of participants completed the study in an average time of 7:53 minutes, while the second group took just 4:20 minutes. We attribute this to smoother transitions between views and fewer problems with deletion in formal view.

4.5.3. Stress-Test Methodology. The goal of this test is to see how the software performs under less controlled conditions, thus evaluating robustness. This study was conducted on a Dell Latitude XT3 running 64-bit Windows 7. It has an Intel i7 2.90GHz process or 8GB of RAM. The eight participants (five male, three female) came from a range of backgrounds. Four were computer science graduate students who were familiar with Euler diagrams, and the others had varied backgrounds; one was familiar with Euler diagrams. To begin, the facilitator demonstrated the tool while describing the basic components of the diagrams. It was emphasized that diagram elements could be drawn in any order and that the shapes of the curves, intersections, and so forth were entirely up to the participants. They were also shown how they could erase and select and edit ink. Before commencing drawing diagrams for the stress test, they were given a few minutes to explore the application and practice drawing. For the stress test, the participants were asked to draw a diagram with whatever arrangement of curves, vertices, and edges they liked. Our testing showed that a reasonable response time, by the software, could be maintained when the diagrams had fewer than about 10 curves and 10 vertices—this is dependent on the size of the items and number of intersections, further discussed later. Therefore, the only condition given to the participants was that their chosen diagram should have about 10 curves and up to 10 vertices and edges.

4.5.4. Stress-Test Results. The participants drew a total of 88 curves that resulted in 171 minimal regions (these are regions formed by the intersections of the curves; zones are made up of minimal regions), 60 vertices, and 39 edges. The complexity of the diagrams varied. Figure 32 is typical; however Participant 6's (Figure 33) is much more complex with 48 minimal regions. In general terms, the software performed well with no crashes. However, when the diagrams get very complex (such as Figure 33) and there is a large amount of ink and filling to render, the software slows to a degree where one would not wish to use it. With more conventional diagrams such as Figure 32, the response time is acceptable at this level of complexity. The results of the recognition and conversion are discussed in the next section.

To further stress-test the software, in terms of software response times, we sketched 12 diagrams of different complexities measured by number of curves, regions, vertices, and edges. The most time-consuming part is the recognition and subsequent updating of the abstract syntax on the addition of a curve. Table I shows times taken to recognize the last sketched curve, which was drawn after the vertices and edges had all been

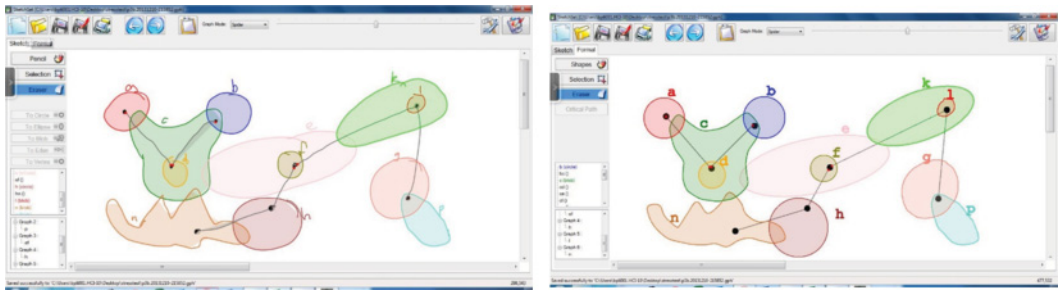

Fig. 32. Participant 3's sketched and formal diagram.

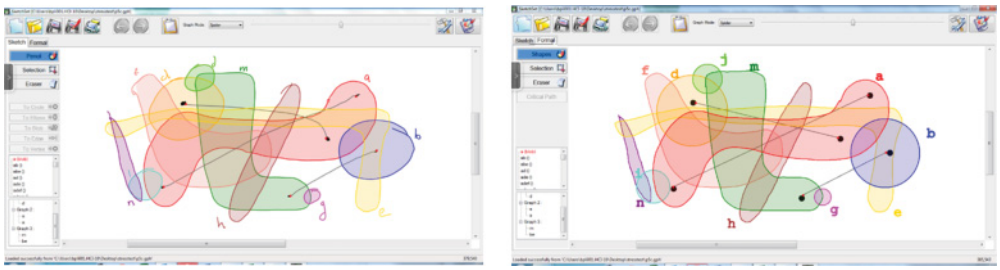

Fig. 33. Participant 5's sketched and formal diagram.

Table I. Computation Time for Curve Addition

Curves	Regions	Vertices	Edges	Time
1	2	1	0	0.240
2	4	2	1	0.252
3	7	3	2	0.395
4	10	4	3	0.459
5	13	5	4	0.485
6	16	6	5	0.634
7	19	7	6	0.834
8	16	8	7	0.945
9	19	9	8	1.006
10	11	10	10	0.345
10	22	10	9	1.231
10	40	0	0	3.403

drawn, and update the abstract syntax; times were computed on a Dell Latitude XT3 with Intel i7 2.80GHz CPU and 8GB Ram, running 64-bitWindows 7.

4.6. Conversion and Recognition

In this section, we first discuss the usability study results and then the stress-testing study results. During the usability study, the participants drew 329 strokes, of which 97 were labels, 59 were circles, 35 were ellipses, 17 were curves, 45 were edges, and 79 were vertices. We did not observe any conversion errors (i.e., differences between the abstract syntaxes of the sketch and formal diagram) during the study. We note that it is possible to create inconsistencies, but the only way in which we have been able to do so is via contrived examples like that in Figure 15.

We calculated the recognition rates at two levels. First, when determining whether a stroke was a label or a drawing element, the recognizer was 98.5% accurate. Labels

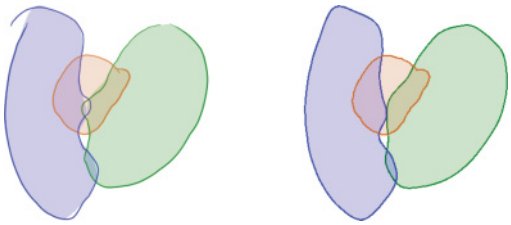

Fig. 34. An inconsistency created by the authors.

were then classified by the Microsoft character recognizer, which for the first round was 86.20% accurate (i.e., eight errors out of 58). For the last four participants, when we invoked the factoid constraint, the recognition rate was 94.87% accurate (i.e., two errors out of 39). The success rates for the drawing elements were as follows. Circles: 59 out of 59, 100% accurate. Ellipses: 35 out of 35, 100% accurate. Curves: 17 out of 17, 100% accurate. Edges: 40 out of 45, 88.88% accurate. Vertices: 66 out of 79, 83.54% accurate. In the first round, three participants drew the edges and one or both vertices in a single stroke; we count these as recognition errors for both edge and vertices. In the second round, we advised people that they must draw vertices and edges separately, and one person did this incorrectly a couple of times but soon modified the technique, drawing them separately. When drawn separately, there was a 100% recognition rate for the edges and a 92.75% recognition rate for the vertices. The five vertices misrecognized were recognized as text. Given the similarity of vertices and some letters, this is not surprising.

In the second (stress test) study, the participants were not required to replicate a preexisting diagram. Therefore, we have categorized the elements simply as curve, label, edge, and vertices using our subjective judgment (although in all cases we felt the intended type of each sketched item was clear). In many cases, the participants observed that a recognition error had occurred at the time of drawing and corrected it immediately—all of these correction incidents were counted as errors. The curves were recognized 100% correctly; labels 100%; edges 94%, with the others recognized as curves; and vertices 90%, with the others recognized as labels. The character recognition was not as good as in the user study at 67%. In part, this was due to the more informal nature of the task coupled with some participants taking less care with their writing; if we could not decipher the letter (for instance, “a” versus “d”), and if the recognizer gave a reasonable result, we did not count this as an error. In addition, the Microsoft recognizer seems to be more reliable on some letters than others: “f,” “i,” and “k” were particularly unreliable. Importantly, there were no inconsistencies between the sketches, formal diagrams, and abstract descriptions, even with these much larger diagrams produced freeform by the participants. This demonstrates that the conversion method is robust.

To further test the robustness of the conversion methods, we attempted to draw diagrams in a way that would cause the conversion method to introduce an inconsistency. Being experts in the underlying algorithms, we are best placed to create such examples. Despite considerable effort, we only found one feature of diagrams that caused inconsistencies to arise: two curves that have small intersecting peaks that get smoothed out by the conversion process. A typical example can be seen in Figure 34. In our experience, diagrams with small, peak-like intersections are not typically drawn by people.

5. DISCUSSION AND CONCLUSION

This article has described the theoretical considerations that are important when developing diagram creation tools that allow both sketched input via a stylus and formal

input using traditional mouse-based input. Both views are necessary: sketching is an important mode of entry, allowing users to focus on the task of diagram creation, but ultimately users want formal versions of their sketches. Moreover, the ability to edit the diagram in both views is important for achieving the final required result and a good user interaction experience. We have presented a conceptual framework required to support systems that switch between sketch and formal approaches to diagram editing and used a case study to demonstrate how the inconsistencies that may result can be rectified. Ensuring the consistency of the semantics between sketch and formal diagram is embodied by the production of their abstract syntaxes. We have demonstrated that this abstract syntax should be more general than is usually seen for visual languages, since it must capture partly formed diagrams. To establish that our approach is practically applicable, we have presented a proof-of-concept software implementation centered around Euler diagrams augmented with graphs.

The size of the diagrams that can be drawn on the current version of the software is limited by the processor power and efficiency of the software. While processors continue to get more powerful, there are two parts of the software that could be optimized. First, the algorithms that extract and match the formal descriptions could be investigated. Second, the graphics repaint could be customized; currently we are using the standard canvas components and the entire canvas is repainted after each event. This could be optimized so that only the region that has been changed is repainted. Thus, there is potential to further enhance the usability of our tool.

The choice of case study was informed by two factors. First, we considered the ways in which visual languages convey information (using spatial relationships as well as connectivity and specific geometric shapes). Euler diagrams augmented with graphs exhibit all these features. Second, our chosen notation has numerous application areas and form the basis of other, more expressive visual languages. Thus, the specific details of our case study provide a basis for future work developing diagram creation tools for these richer notations. For instance, Euler diagrams augmented with graphs can be extended with additional syntax, including arrows, shading, and mathematical symbols, to give a notation rich enough to model ontologies [Howse et al. 2011]. Such syntax can be seen in other applications including visual logic representations and visualization of semantic networks and computer networks. To fully support diagram creation in these areas, we will need to further develop our software for these more complex diagram types. Thus, the work in this article lays a foundation for providing diagram creation tools for visual languages having even wider-ranging practical application.

ELECTRONIC APPENDIX

The electronic appendix of this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We would like to thank Mengdi Wang and Dhruv Dhir for their substantial contributions to the software implementation required for this article.

REFERENCES

T. Baar. 2006a. Correctly defined concrete syntax for visual modeling languages. In *MODELS*. Springer, 111–125.

T. Baar. 2006b. Correctly defined concrete syntax for visual models. In *MoDELS/UML*. Springer, 111–125.

R. Blagojevic, B. Plimmer, J. Grundy, and Y. Wang. 2008. A data collection tool for sketched diagrams. In *SBIM*. 80–93.

P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. 2001. A visualization of OCL using collaborations. In *UML 2001*. Springer, 257–271.

S. H. H. Chang, R. Blagojevic, and B. Plimmer. 2012. RATA.Gesture: A gesture recognizer developed using data mining. *Artificial Intelligence for Engineering, Analysis and Manufacturing* 26, 3 (2012), 351–366.

R. Clarke. 2007. Fast zone discrimination. In *Visual Languages and Logic*. 41–54.

G. Cordasco, R. De Chiara, and A. Fish. 2011. Efficient on-line algorithms for Euler diagram region computation. *Computational Geometry: Theory and Applications* 44, 1 (2011), 52–68.

G. Costagliola, A. De Lucia, S. Orefice, and G. Polese. 2002. A classification framework to support the design of visual languages. *Journal of Visual Languages and Computing* 13, 6 (2002), 573–600.

C. Damm, K. Hansen, and M. Thomsen. 2000. Tool support for cooperative object-oriented design: Gesture based modelling on an electronic whiteboard. In *SIGCHI Conference on Human Factors in Computing Systems*. ACM, 518–525.

F. Dau. 2007. Constants and functions in Peirce's existential graphs. In *Conceptual Structures*. 429–442.

R. DeChiara, U. Erra, and V. Scarano. 2003. VennFS: A Venn diagram file manager. In *Proceedings of Information Visualisation*. IEEE Computer Society, 120–126.

H. Dunn-Davies and R. Cunningham. 2005. Propositional statecharts for agent interaction protocols. In *Proceedings of Euler Diagrams 2004, Brighton, UK (ENTCS)*, Vol. 134. 55–75.

M. Erwig. 1998. Abstract syntax and semantics of visual languages. *Journal of Visual Languages and Computing* 9 (1998), 461–483.

G. Farrell and W. Sousa. 2001. Repeat victimization and hot spots: The overlap and its implication for crime control and problem-oriented policing. *Crime Prevention Studies* 12 (2001), 221–240.

M. Field, S. Gordon, E. Peterson, R. Robinson, T. Stahovich, and C. Alvarado. 2009. The effect of task on classification accuracy: Using gesture recognition techniques in free-sketch recognition. *Computers and Graphics* 34, 5 (2009), 499–512.

A. Fish, J. Flower, and J. Howse. 2005. The semantics of augmented constraint diagrams. *Journal of Visual Languages and Computing* 16, 6 (2005), 541–573.

J. Gil, J. Howse, and S. Kent. 1999. Formalising spider diagrams. In *Proceedings of IEEE Symposium on Visual Languages (VL'99)*. IEEE Computer Society Press, 130–137.

V. Goel. 1995. *Sketches of Thought*. MIT Press.

G. Goldschmidt. 1999. The backtalk of self-generated sketches. In *Visual and Spatial Reasoning in Design*. University of Sydney, 163–184.

M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. 2009. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter* 11, 1 (2009), 10–18.

T. Hammond and R. Davis. 2002. Tahuti: A geometrical sketch recognition system for UML class diagrams. In *AAAI Spring Symposium on Sketch Understanding*.

J. Howse, F. Molina, S.-J. Shin, and J. Taylor. 2001. Type-syntax and token-syntax in diagrammatic systems. In *Proceedings Fof the 2nd International Conference on Formal Ontology in Information Systems (FOIS'01)*. Maine, USA. ACM Press, 174–185.

J. Howse, G. Stapleton, and J. Taylor. 2005. Spider diagrams. *LMS Journal of Computation and Mathematics* 8 (2005), 145–194.

J. Howse, G. Stapleton, K. Taylor, and P. Chapman. 2011. Visualizing ontologies: A case study. In *International Semantic Web Conference*. Springer, 257–272.

E. Ip. 2001. Visualizing multiple regression. *Journal of Statistics Education* 9, 1 (2001).

Y. Jiang, F. Tian, X. L. Zhang, G. Dai, and H. Wang. 2011. Understanding, manipulating and searching hand-drawn concept maps. *ACM Transactions on Intelligent Systems and Technology* 3, 11 (2011), 21.

G. Johnson, M. Gross, and J. Hong. 2009. *Computational Support for Sketching in Design*. Now Publishers.

S. Kent. 1997. Constraint diagrams: Visualizing invariants in object oriented models. In *Proceedings of OOPSLA97*. ACM Press, 327–341.

H. Kestler, A. Muller, J. Kraus, M. Buchholz, T. Gress, H. Liu, D. Kane, B. Zeeberg, and J. Weinstein. 2008. VennMaster: Area-proportional euler diagrams for functional GO analysis of microarrays. *BMC Bioinformatics* 9, 67 (2008).

H. Kestler, A. Muller, H. Liu, D. Kane, B. Zeeberg, and J. Weinstein. 2005. Euler diagrams for visualizing annotated gene expression data. In *Proceedings of Euler Diagrams 2005*, Paris.

S. Maier, S. Mazanek, and M. Minas. 2008. Layout specification on the concrete and abstract syntax level of a diagram language. In *Layout of (Software) Engineering Diagrams*, Vol. 13. ECEASST.

K. Mineshima, M. Okada, and R. Takemura. 2012. A diagrammatic inference system with euler circles. *Journal of Logic, Language and Information* 21, 3 (2012), 365–391.

Petri Nets. 2013. Standardisation. Retrieved from http://www.petrinets.info/.

C. Peirce. 1933. *Collected Papers*, Vol. 4. Harvard University Press.

B. Plimmer. 2008. Experiences with digital pen, keyboard and mouse usability. *Journal of Multi-modal User Interfaces* 2, 1 (2008), 13–23.

B. Plimmer and I. Freeman. 2007. A toolkit approach to sketched diagram recognition. In *HCI*. British Computer Society, 205–213.

B. Plimmer, H. Purchase, and H. Laycock. 2009. Preserving the hand-drawn appearance of graphs. In *Visual Languages and Computing*. 347–352.

B. Plimmer, H. Purchase, and H. Yang. 2010. SketchNode: Intelligent sketching support and formal diagramming. In *OZCHI 2010*. ACM, 136–143.

D. Rubine. 1991. Specifying gestures by example. *ACM SIGGRAPH Computer Graphics* 25, 4 (1991), 329–337.

T. Sezgin, T. Stahovich, and R. Davis. 2001. Sketch based interfaces: Early processing for sketch understanding. In *Workshop on Perceptive User Interfaces*.

S.-J. Shin. 1994. *The Logical Status of Diagrams*. Cambridge University Press.

J. Soriano, K. Davis B. Coleman, G. Visick, D. Mannino, and N. Pride. 2003. The proportional Venn diagram of obstructive lung disease. *Chest* 124 (2003), 474–481.

J. Sowa. 2013. Conceptual Graphs. Retrieved from http://www.jfsowa.com/cg/.

G. Stapleton and J. Masthoff. 2007. Incorporating negation into visual logics: A case study using euler diagrams. In *Visual Languages and Computing 2007*. Knowledge Systems Institute, 187–194.

G. Stapleton, P. Rodgers, J. Howse, and J. Taylor. 2007. Properties of euler diagrams. In *Proceedings of Layout of Software Engineering Diagrams*. EASST, 2–16.

N. Swoboda and G. Allwein. 2005. Heterogeneous reasoning with euler/venn diagrams containing named constants and FOL. In *Proceedings of Euler Diagrams 2004 (ENTCS)*, Vol. 134. Elsevier Science.

J. Thièvre, M. Viaud, and A. Verroust-Blondet. 2005. Using euler diagrams in traditional library environments. In *Euler Diagrams 2004 (ENTCS)*, Vol. 134. ENTCS, 189–202.

Unified Modeling Language. 2006. Unified Modeling Language (UML) Resource Page. Retrieved from http://www.uml.org/.

M. Wang, B. Plimmer, P. Schmieder, G. Stapleton, P. Rodgers, and A. Delaney. 2011. SketchSet: Creating euler diagrams using pen or mouse. In *IEEE Symposium on Visual Languages and Computing*. IEEE, 75–82.

L. Wilkinson. 2012. Exact and approximate area-proportional circular Venn and Euler diagrams. *IEEE Transactions on Visualization and Computer Graphics* 18, 2 (2012), 321–331.

R. Wilson. 1996. *Introduction to Graph Theory*. Prentice Hall.

J. Wobbrock, A. Wilson, and L. Yang. 2007. Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. In *20th Annual ACM Symposium on User Interface Software and Technology*. ACM.

A. Wolin, M. Field, and T. Hammond. 2011. Combining corners from multiple segmenters. In *8th Eurographics Symposium on Sketch-Based Interfaces and Modeling*.

L. Yeung, B. Plimmer, B. Lobb, and D. Elliffe. 2008. Effect of fidelity in diagram presentation. In *22nd British HCI Group Annual Conference on People and Computers: Culture, Creativity, Interaction*, Vol. 1. British Computer Society, 35–44.

Received September 2013; revised May 2014; accepted June 2014