A Reading Algorithm for Constraint Diagrams

Andrew Fish, Jean Flower and John Howse
Visual Modelling Group
University of Brighton, Brighton, UK
`{Andrew.Fish, J.A.Flower, John.Howse}@brighton.ac.uk`
http://www.cmis.brighton.ac.uk/research/vmg/

Abstract

Constraint diagrams *are a visual notation designed to complement the Unified Modeling Language in the development of software systems. They generalize Venn diagrams and Euler circles, and include facilities for quantification and navigation of relations. Their design emphasizes scalability and expressiveness while retaining intuitiveness. The formalization of constraint diagrams is non-trivial: previous attempts have exposed subtleties concerned with the ordering of symbols in the visual language. Consequently, some constraint diagrams have more than one intuitive reading. We develop the concept of the* dependence graph *for a constraint diagram. From the dependence graph we obtain a set of* reading trees. *A reading tree provides a partial ordering for some syntactic elements of the diagram. Given a reading tree for a constraint diagram, we present an algorithm that delivers a unique semantic reading.*

Keywords Visual formalisms, software specification, formal methods, constraint diagrams, diagrammatic reasoning.

1 Introduction

The Unified Modeling Language (UML) [11] is the Object Management Group's industrial standard for software and system modelling. This has accelerated the uptake in the software industry of diagrammatic notations for designing systems.

In this paper, we describe a notation, *constraint diagrams*, which was introduced in [9] for use in conjunction with UML for object-oriented modelling. Constraint diagrams provide a diagrammatic notation for expressing logical constraints, such as invariants and operation preconditions and postconditions. In UML constraints are expressed using the Object Constraint Language (OCL) [14], which is essentially a textual, stylised form of first order predicate logic (FOPL).

Constraint diagrams were developed to enhance the visualization of object structures. Class diagrams show relationships between objects as associations between classes. Annotating, with cardinalities and aggregation for example, enables one to exhibit some properties of these relationships between objects. However, frequently one wishes to exhibit subtler properties, such as those of composite relations. Whereas this is impossible using class diagrams, the inherent visual structure of constraint diagrams makes this, and many other constructions, easy to express. So constraint diagrams fit in with the diagrammatic theme of UML and may be easier for software engineers to use than non-diagrammatic OCL or FOPL.

Constraint diagrams build on a long history of using diagrams to visualize logical or set-theoretical assertions. They generalize Venn diagrams [13] and Euler circles [1], which are currently rich research topics, particularly as the basis of visual formalisms and diagrammatic reasoning systems [12, 5, 6, 7]. Constraint diagrams are vastly more expressive than these systems because they can express relations, whilst still retaining the elegance, simplicity and intuitiveness of the underlying diagrammatic systems. For constraint diagrams to be used effectively in software development, it is necessary to have strong tool support. Such tools are currently being developed [10].

In §2 we give a concise description of constraint diagrams and consider examples with multiple intuitive readings. Formal notions required for the reading algorithm are set up in §3 and §4. In §5 we define dependence between certain syntactic elements of the diagram. This enables us to associate a unique dependence graph to a diagram, in §6. Using this graph, we construct a set of reading trees in §7. Given a constraint diagram and a reading tree for that diagram, we construct a building sequence (§8). A building sequence corresponds to a choice of construction of the diagram. In order to give formal meaning to the diagrams we move from the syntactic realm to the semantic realm in §9. The reading algorithm, which produces a unique semantic reading for a diagram with respect to a reading tree is given in §10. Finally, in §11, we highlight the usefulness of this (and further) work, especially with regard to tool creation.

2 Constraint diagrams

In this section we give a concise description of constraint diagrams. A *contour* is a simple closed curve in the plane. The area of the plane which constitutes the whole diagram is a *basic region*. Furthermore, the bounded area of the plane enclosed by a contour c is the *basic region of* c. A *region* is defined recursively: a basic region is a region, and any non-empty union, intersection, or difference of regions is a region. A *zone* is a region which contains no other region. A zone may be *shaded*. A region is *shaded* if all its zones are shaded.

A *spider* is a tree with nodes (called *feet*) in distinct zones. It *touches* any region which contains (at least) one of its feet. The union of zones that a spider touches is called the spider's *habitat*. A spider is either an existential spider, whose feet are drawn as dots, or a universal spider, whose feet are drawn as asterisks.

The source of a labelled *arrow* may be a contour or a spider. The target of a labelled arrow may be a contour or a spider. A contour is either a *given* contour, which is labelled, or a *derived* contour, which is unlabelled and is the target of some arrow.

Given contours represent sets, arrows represent relations and derived contours represent the image of a relation. Existential quantification is represented by existential spiders, and universal quantification is represented by universal spiders. The scope of quantification is a set and this set is represented by a region which is called the scope of the spider. Distinct spiders represent distinct elements. A shaded region with n existential spiders touching it represents a set with no more than n elements.

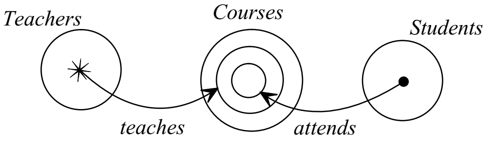

Fig. 1: Ordering of quantifiers

In [4], a number of issues were raised concerning the interpretation of constraint diagrams. For example, because there is no linear ordering of syntactic elements in a constraint diagram and the ordering of universal and existential quantifiers is important, the diagram in figure 1 can have different interpretations. Starting with the universal spider we obtain: *For each teacher there is a student who attends only courses taught by that teacher*. Whereas, starting with the existential spider we get: *There is a student who attends only courses taught by all teachers*. This issue arises when considering any diagram which includes both an existential spider and a universal spider.

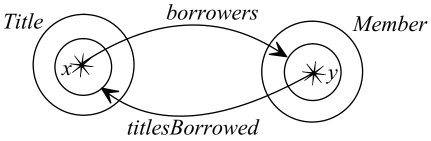

Fig. 2: Scope of quantifiers

Also of interest is the scope of quantifiers. A spider represents a quantifier, but the spider's habitat may be smaller than its scope.

For example, in figure 2 there are two sensible readings, depending upon whether one starts at x or at y.

If one starts at x, then its scope is the region inside the contour *Title*, even though its habitat is the smaller region inside the derived contour contained in *Title*. We read: "Each title, x, is borrowed by a set of members." The scope of the other spider, y, is the region inside the derived contour contained in Member. The habitat of y is equal to its scope. We read: "Each of these members is borrowing titles including x."

On the other hand, assume that one starts reading at y. Then the scope of x is equal to its habitat, but the scope of y is larger than its habitat. One obtains the semantically different reading: "Each member, y, is borrowing a set of titles. Each of these titles is borrowed by members including y."

3 Formal Syntax

In this section we give the abstract syntax of constraint diagrams, that is, a formal definition which is independent of any topological and visual representations.

A *constraint diagram* consists of contours, zones, spiders and arrows, formally defined as follows.

1. $\mathcal{C}$ is a finite set of *contours* which is partitioned into subsets $\mathcal{C}^G$, the set of *given contours*, and $\mathcal{C}^D$, the set of *derived contours*.

   $\mathcal{L}^G$ is a set of *contour labels* and the bijection $\ell_G : \mathcal{C}^G \to \mathcal{L}^G$ returns the label of a given contour.
2. The set $\mathcal{Z} \subseteq \mathcal{PC} \times \mathcal{PC}$ is the set of *zones*, where $\mathcal{PC}$ is the power set of $\mathcal{C}$. Each $z \in \mathcal{Z}$ is a pair $(C^+(z), C^-(z))$, where $C^+(z)$ and $C^-(z)$ partition $\mathcal{C}$ and represent the contours containing and excluding z respectively. The set $\mathcal{Z}^* \subseteq \mathcal{Z}$ is the set of *shaded zones*. $\mathcal{R} = \mathcal{PZ} - \{\emptyset\}$ is the set of *regions*.
3. $\mathcal{S}$ is a finite set of *spiders* which is partitioned into $\mathcal{S}^e$, the set of *existential spiders*, and $\mathcal{S}^u$, the set of *universal spiders*.

The function $\eta : \mathcal{S} \to \mathcal{R}$ returns the habitat of a spider.

4. $\mathcal{A}$ is a finite set of *arrows*. $\mathcal{L}^A$ is a set of *arrow labels*. The functions $\ell_A : \mathcal{A} \to \mathcal{L}^A$, $S : \mathcal{A} \to \mathcal{C} \cup \mathcal{S}$, $T : \mathcal{A} \to \mathcal{C} \cup \mathcal{S}$ return the *label*, *source* and *target* of an arrow, respectively.

We will use the standard object-oriented notation $x.r$ to represent textually the relational image of element x under relation r, that is, $x.r = \{y : (x, y) \in r\}$. Thus $x.r$ is the set of all elements related to x under relation r. The expression $x.r$ is a *navigation expression*, so called because we can *navigate* from x along the arrow r to the set $x.r$. The relational image of a set S is then defined by $S.r = \bigcup_{x \in S} x.r$.

4 Formal concepts

In this section we set up formal concepts required for the reading algorithm. A complete understanding of the details is unnecessary in order to appreciate the idea of the algorithm.

If P and Q are disjoint subsets of $\mathcal{C}$ then the *zonal region* [8] is the region which is contained in all contours in P and excluded from all contours in Q. This region is a zone in any diagram with contour set $P \cup Q$. That is,

$$(P, Q) = \{z \in \mathcal{Z} : P \subseteq C^+(z) \wedge Q \subseteq C^-(z)\}.$$

Any region r is a set of zones and hence can be *represented* as a union of zonal regions. We can minimize the number of contours in a description of r by choosing different zonal region representations. A *minimal representation* for r is a representation with the fewest number of contours. Such a representation is not necessarily unique.

The union of the sets of derived contours that appear in minimal representations of r is called the *derived contour set*, and is denoted by $DC(r)$.

An arrow, f, is said to *hit the boundary* of a region, r, if $T(f) \in DC(r)$. An arrow, f, is said to *hit* a derived contour c if $T(f) = c$. In figure 3, arrow f hits the boundary of the shaded region r. Informally this is because $T(f)$ is a derived contour and forms part of the boundary of r. Arrow g does not hit the boundary of r, because $T(g)$ is given and not derived, and arrow h does not hit the boundary of r.

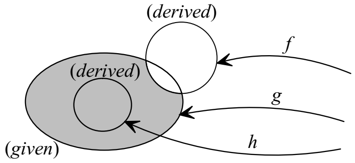

Fig. 3: Arrows *hitting* region boundaries and contours

For a contour, c, we will want to describe the placement of c relative to other contours in the diagram.

Informally, think of $R_{out}(c)$ as the smallest region of the diagram properly enclosing the basic region of c, whose boundary does not touch c. Also, think of $R_{in}(c)$ as the largest region which is properly enclosed by the basic region of c, whose boundary does not touch c. Then we may describe the location of c as "containing $R_{in}(c)$" and "contained in $R_{out}(c)$".

For example, in figure 4, $R_{in}(C)$ is the shaded region in the first diagram, whereas $R_{out}(C)$ is the shaded region in the second diagram.

Formally, for each zone $z = (C^+(z), C^-(z))$ in the basic region of c, consider the zonal region $(C^+(z) - \{c\}, C^-(z))$. Then $R_{out}(c)$ is the union of these zonal regions. Define $R_{in}(c)$ by removing from $R_{out}(c)$ those zonal regions for which $(C^+(z) - \{c\}, C^-(z) \cup \{c\})$ is a zone. Note that $R_{in}(c)$ may be empty.

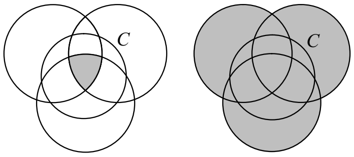

Fig. 4: Illustrating R_{in} and R_{out}

5 Dependence between components

The first step towards generating a semantic reading of a constraint diagram is to determine the dependences that exist between spiders and arrows. Let an *object* in d be a spider or an arrow.

For example, if we have an arrow f with a spider s as its source s then we say that arrow f *depends upon* spider s. In some other cases dependence is symmetric, and we will refer to *dependence between* objects.

Dependence is defined by the six criteria listed below. The dependence relation is not transitive. This means that it is possible to have s dependent upon t and t dependent upon f without concluding that s is dependent upon f.

1. There is a *dependence between* spiders s and t if their habitats have non-empty intersection.
2. If the source of an arrow f is a derived contour c then f is *dependent upon* any arrow which hits c.
3. If the source or the target of an arrow f is a spider s then f is *dependent upon* s
4. There is a *dependence between* spider s and arrow f if f hits the boundary of the habitat of s.

5. The shaded zones of a diagram can be collected into regions which are connected components.[1] For each such shaded region r, construct the set of spiders which touch r. Add to this set the arrows which hit the boundary of the region r. There is pairwise *dependence between* the objects in this set.
6. There is *dependence between* arrows f and g, whose targets are derived contours, if $T(f)$ and $T(g)$ are equal or if $T(f) \in DC(r)$ where r is $R_{in}(T(g))$ or $R_{out}(T(g))$, or vice versa.

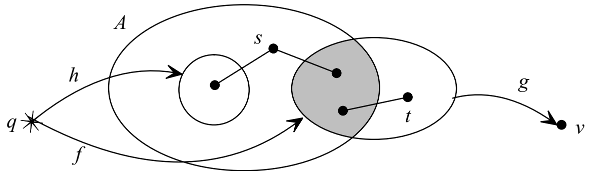

Fig. 5: Object dependence

These six dependence criteria can be illustrated with reference to the diagram in figure 5. This example was chosen to exhibit many different types of dependence, and is much more complicated than a typical diagram used in modelling software requirements.

1. Spiders s and t both touch the (shaded) zone $(\{A, T(f)\}, \{T(h)\})$, so there is dependence between s and t. Similarly, there is dependence between q and v.
2. Arrow f hits the source of g, so g is dependent upon f.
3. Arrows f and h are dependent upon their source q, and g is dependent upon its target v.
4. Arrow f hits the boundary of the habitat of t so there is dependence between t and f. Similarly, there is dependence between q and f and between v and f.
5. The shaded region is touched by s and t, and its boundary is hit by f, so there is pairwise dependence between all three of these objects.
6. The contour $T(h)$ has $R_{out}(T(h)) = (\{A\}, \{T(f)\})$. This is a minimal representation of $R_{out}(T(h))$, so there is dependence between f and h.

6 Dependence graphs

The dependences between objects can be described collectively using a *dependence graph* $G(d)$, whose edges represent the dependence of one object upon another (directed edges) and dependence between objects (undirected edges).

Given a diagram, d, construct $G(d)$ as follows:

- the node set of $G(d)$ is the set of objects of d,
- if object o_1 is dependent upon object o_2 then add a directed edge from the node o_2 to the node o_1 and
- if there is a dependence between objects o_1 and o_2 then add an undirected edge between the corresponding nodes.

Identify duplicate edges between nodes. Remove undirected edges between nodes which also have a directed edge between them.

The graph generated from dependences for figure 5 is shown in figure 6. Duplicate edges were identified between s and t and again between f and t. An undirected edge was removed between q and f because of the directed edge from q to f.

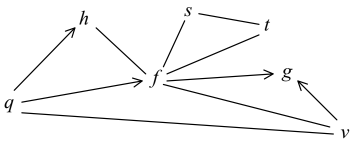

Fig. 6: The dependence graph for figure 5

A simpler example, with its dependence graph is shown in figure 7.

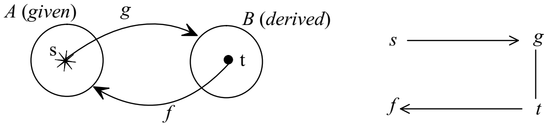

Fig. 7: A diagram and its dependence graph

A diagram is defined to be *unreadable* if its dependence graph has a directed cycle, and *readable* otherwise.

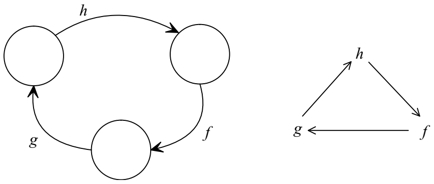

Fig. 8: An unreadable diagram

Equivalently, a diagram is unreadable if and only if there exists a sequence of arrows, $f_1, \ldots, f_n$, with $n \geq 1$, such that $T(f_i)$ is a derived contour which is equal to $S(f_{i+1})$, for each i (reading f_1 for f_{n+1}).

---

[1] When we say that a region is *connected* we are not strictly referring to the notion of a connected topological set, but rather to the notion of connected subgraphs of the abstract dual graph see [3].

7 Reading Trees

From the dependence graph of a diagram we construct a *reading tree*. This tree is directed and choices made in its construction will yield different semantic readings of the diagram.

Let $G(d)$ be the dependence graph of a readable diagram d. Construct a directed forest, $F(d)$, as follows. The node set of $F(d)$ equals the node set of $G(d)$. Add directed edges to $F(d)$ according to the following conditions.

- No two directed edges in $F(d)$ end at the same node.
- If there is a directed edge from node n_1 to n_2 in $G(d)$, then there is a directed path in $F(d)$ from n_1 to n_2.
- If there is an undirected edge between nodes n_1 and n_2 in $G(d)$, then the nodes n_1 and n_2 must lie in a common directed path in $F(d)$.

As a consequence of these conditions placed on $F(d)$, each tree component has a unique *starting* node, n, with no incoming edges.

Add a node, labelled by PTC (for *plane tiling condition* see equation 1 in section 9), to $F(d)$, and a directed edge from this node to each starting node. Any rooted, directed tree thus obtained is called a *reading tree* of d, denoted by $RT(d)$.

Figure 9 shows two examples of reading trees obtainable from the dependence graph in figure 7.

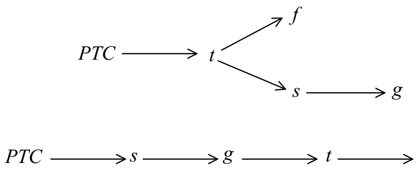

Fig. 9: Two reading trees for the example in figure 7

In the following sections we construct a semantic interpretation for a diagram and reading tree which is unique up to the logical equivalence $P \wedge Q \equiv Q \wedge P$.

8 Building Sequence

Given a reading tree, $RT(d)$, we construct a sequence of diagrams, called a *building sequence* of *building diagrams*. The initial diagram in a building sequence includes the given contours of d and some shading, and the final diagram in the sequence is d itself. Intermediate diagrams in the sequence are obtained by adding diagrammatic elements.

A building sequence is constructed from the reading tree as follows. Perform a depth-first search of the reading tree, generating an ordering on its nodes n_0, n_1,. . . ,n_m. Use this to define the building sequence diagrams d_0,. . . ,d_m. Then d_{j+1} is obtained from d_j by adding an object n_{j+1} and any of its syntactic consequences (see below).

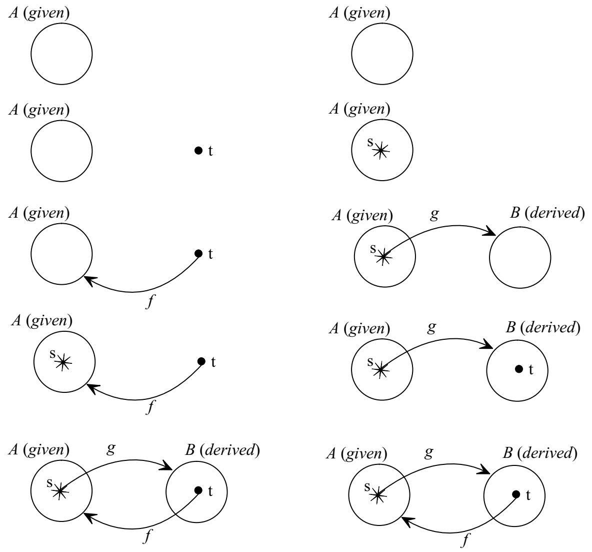

Fig. 10. Building sequences for the two reading trees in figure 9

The initial diagram d_0 consists of all of the given contours of d, together with the connected components of shading whose boundaries are not hit by any arrows and have no spiders touching them.

There are two types of syntactic consequence of adding an object. First, adding an arrow whose target contour is not included in d_j, requires the addition of the target contour in d_{j+1}. If adding this derived contour splits a zone of d_j which had a spider's foot in it, and the spider's habitat in the final diagram meets both new zones (zonal regions in d) then create feet for that spider in the two new zones (sometimes called *splitting* spiders' feet).

In the second case, adding a spider or adding an arrow may complete the set of spiders which touch, or arrows which hit the boundary of, a connected component of shading. In this case, the addition of the shading is a syntactic consequence of adding the object.

When all nodes of the reading tree have been visited, all objects have been added and $d_m = d$, where m is the total number of objects in d.

Figure 10 shows the building sequences for the two reading trees in figure 9. A depth first search of the top tree in figure 9 reads *PTC, t, f, s, g* which corresponds to the leftmost building sequence.

9 Semantics

Up to this point, we have been analysing and manipulating the syntax of diagrams. Diagram semantics have only been mentioned informally. In this section, we begin the creation of a predicate logic statement which is the semantics of a constraint diagram.

Begin by defining an *interpretation* for a constraint diagram, which assigns sets to given contour labels and relations to arrow labels. An *interpretation* for d is a tuple $\langle \mathbf{U}, \Psi, \phi \rangle$ where $\mathbf{U}$ is a set and Ψ and ϕ are functions, $\Psi : \mathcal{L}^G \rightarrow \mathcal{P}\mathbf{U}$ maps given contour labels to subsets of $\mathbf{U}$ and $\phi : \mathcal{L}^A \rightarrow \mathcal{P}(\mathbf{U} \times \mathbf{U})$ maps arrow labels to relations on $\mathbf{U}$.

Using the assignment of labels to arrows, we will extend ϕ to map arrows to relations on $\mathbf{U}$, and write $\phi(a)$ instead of $\phi(\ell_A(a))$ for an arrow a. Similarly, write $\Psi(c)$ in place of $\Psi(\ell_G(c))$ for a given contour c.

Extend the definition of Ψ to interpret zones and regions which are built from given contours – if zone z is the contour partition (P, Q) of $\mathcal{C}^G$ then define

$$\Psi(z) = \bigcap_{c \in P} \Psi(c) \cap \bigcap_{c \in Q} \overline{\Psi(c)}$$

where $\overline{\Psi(c)} = \mathbf{U} - \Psi(c)$.

Recall that d_0 is the subdiagram of d comprising the given contours and some shading, and define the *plane tiling condition (PTC)*:

$$\bigcup_{z \in \mathcal{Z}(d_0)} \Psi(z) = \mathbf{U}. \tag{1}$$

This condition says, for d_0, explicitly that the sets assigned to the zones comprise the universal set $\mathbf{U}$ and also, implicitly, that any zones absent from the diagram are interpreted as the empty set.

The plane tiling condition for the example in figure 7 just states that $\Psi(A) \cup \overline{\Psi(A)} = \mathbf{U}$, because there is only one given contour. The plane tiling condition is included in the semantics of d. Other conditions are harder to describe, and for these we require the apparatus from earlier sections of this paper.

For each diagram in the building sequence, we construct text statements called *semantic consequences*. Each diagram d_j includes a new object n_j, and the semantic consequence is denoted $Con(n_j)$. For example, if we add a spider s in given contour A, then $Con(s)$ is $s \in \Psi(A)$. The semantic consequence of an object expresses in text form the meaning that we derive from the presence of the object in the diagram.

An example to show the semantic consequences in a building sequence is given in figure 11. The rest of this section is devoted to a complete formal description of how the semantic consequences are obtained.

As well as defining the consequence $Con(o)$ for each object o in d, we will also define $text(e)$ for *diagram elements* e: contours (given or derived), spiders, arrows, and regions. The text function is defined inductively, using the building sequence. At the start of the induction:

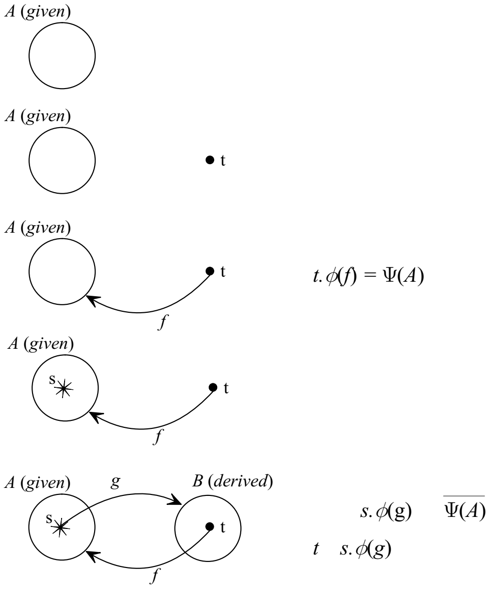

Fig. 11: Semantic consequences

- for each given contour c, define $text(c) = \Psi(c)$,
- for each arrow a, define $text(a) = \phi(a)$,
- for each spider s, define $text(s) = s$.

The *text* function extends to a region $r = \bigcup_i (P_i, Q_i)$ provided that it is defined for those contours in P_i and Q_i. Define $text(r)$ to be

$$\bigcup_i \left[\bigcap_{c \in P_i} text(c) \cap \bigcap_{c \in Q_i} \overline{text(c)} \right].$$

If r corresponds to a zone in d_0, that is P_i and Q_i partition $\mathcal{C}^G$, then $text(r)$ is a textual version of the extension of Ψ described earlier and uses $\Psi(c)$ for each occurrence of $text(c)$. However, $\Psi(c)$ is undefined for a derived contour c, so our construction of the text function is necessary.

We constructed the building sequence with three purposes: we assign text to new diagram elements, construct $Con(o)$ for new objects o, and find the scope of quantification for spiders.

Let d_{j+1} be obtained from d_j by adding object n_j (spider or arrow). Assume inductively that the *text* function assigns text to all elements and regions in d_j.

1. n_j is an arrow, f, whose syntactic consequences include the addition of contour $T(f)$. The source $S(f)$ is present in diagram d_j and so we inductively know $text(S(f))$. Define $text(T(f))$ to be $text(S(f)).\phi(f)$.

To describe the placement of $T(f)$ relative to existing contours in d_j, find the regions $in = R_{in}(T(f))$ and $out = R_{out}(T(f))$ in d_{j+1}. Construct the *semantic consequence*

$$text(in) \subseteq text(T(f)) \subseteq text(out).$$

2. n_j is an arrow, f, whose syntactic consequences do not include the addition of a contour. Construct the *semantic consequence*
$$text(S(f)).\phi(f) = text(T(f)).$$
3. n_j is an arrow, f, and spider s has non-corresponding habitats $\eta_j(s)$ in d_j and $\eta_{j+1}(s)$ in d_{j+1} (this means that these habitats differ if they are compared in the final diagram $d = d_m$). Let r be the habitat of s in d_{j+1}. Construct the *semantic consequence*
$$s \in text(r).$$
4. n_j is a spider, s, whose habitat which is touched by a spider t. Construct the *semantic consequence*
$$s \neq t.$$
5. n_j is an object which is the last object to hit the boundary of, or touch, a connected component of shading r. Let $spiderSet$ denote the text representation of the (possibly empty) set of spiders which touch r. Construct the *semantic consequence*
$$text(r) - spiderSet = \emptyset.$$

If there are no consequences of an object, o, then define $Con(o) = true$. Otherwise $Con(o)$ is defined to be the conjunction of consequences constructed above.

For the example in figure 11, we have $Con(t)$ is "*true*", $Con(f)$ is "$t.\phi(f) = \overline{\Psi(A)}$", $Con(s)$ is "*true*" and $Con(g)$ is "$\emptyset \subseteq s.\phi(g) \subseteq \overline{\Psi(A)} \wedge t \in s.\phi(g)$".

We need one final concept before proceeding to construct a semantic reading of a diagram and reading tree. Spiders denote quantification over sets. The *scope* of a spider s is the region representing the set over which quantification occurs and is defined to be the habitat of s in the first building diagram d_j in which s appears.

Note that the scope of a spider is not necessarily equal to its habitat in $d = d_m$. In figure 11, the scope of t is the zonal region $(\emptyset, \{A\})$ whereas its habitat is the zone $(\{T(g)\}, \{A\})$. The scope of s is equal to its habitat: $(\{A\}, \emptyset)$.

10 Reading Algorithm

We have now set up the framework which enables us to write down the semantic reading of a constraint diagram, given a choice of reading tree.

Repeat the depth-first search of the reading tree traversing nodes $n_0, \ldots, n_m$, followed by the unique path back to the root node. Read the semantics of the diagram, with respect to the reading tree, according to the following rules

- The initial encounter of
  - the root node gives the Plain Tiling Condition
  - a universal spider, s: "$\forall s \in text(scope(s))$ $(Con\,(s)$"
  - an existential spider, s: "$\exists s \in text(scope(s))$ $(Con\,(s)$"
  - an arrow a labelled, f: "$(Con\,(f)$"
- Traversal of edges matching orientation: "$\wedge$"
- Traversal of edges opposing orientation: ")"

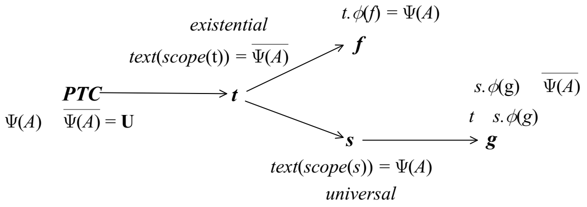
$t.\phi(f) = \Psi(A)$
existential
$text(scope(\mathrm{t})) = \overline{\Psi(A)}$
f
$s.\phi(\mathrm{g})$ $\overline{\Psi(A)}$
PTC
t
$\Psi(A)$ $\overline{\Psi(A)} = \mathbf{U}$
t $s.\phi(g)$
s
g
$text(scope(s)) = \Psi(A)$
universal

Fig. 12: Information used in the algorithm

To illustrate the algorithm we will derive the semantic reading using the first tree in figure 9 and the building sequence in figure 11. For clarity, in figure 12, we have redrawn the reading tree, placing information next to the nodes which is obtainable from the building sequence. Begin at the root node, PTC:

$$\Psi(A) \cup \overline{\Psi(A)} = \mathbf{U} \ldots$$

Navigate along the first edge and arrive at t:

$$\Psi(A) \cup \overline{\Psi(A)} = \mathbf{U} \wedge \exists t \in \overline{\Psi(A)}\ (true \ldots$$

Continue the depth-first search along the edge to f:

$$\Psi(A) \cup \overline{\Psi(A)} = \mathbf{U} \wedge \exists t \in \overline{\Psi(A)}\ (true$$
$$\wedge (t.\phi(f) = \Psi(A) \ldots$$

Traverse the edge back to t, adding a close bracket. Then traverse to s:

$$\Psi(A) \cup \overline{\Psi(A)} = \mathbf{U} \wedge \exists t \in \overline{\Psi(A)}\ (true$$
$$\wedge (t.\phi(f) = \Psi(A)) \wedge \forall s \in \Psi(A)\ (true \ldots$$

Finally traverse to g and back to the root node, adding close brackets to complete the semantic reading:

$$\Psi(A) \cup \overline{\Psi(A)} = \mathbf{U} \wedge \exists t \in \overline{\Psi(A)}\ (true$$
$$\wedge (t.\phi(f) = \Psi(A)) \wedge \forall s \in \Psi(A)\ (true$$
$$\wedge (\emptyset \subseteq s.\phi(g) \subseteq \overline{\Psi(A)} \wedge t \in s.\phi(g)))).$$

After removing excess syntax omitting Ψ and ϕ for readability this example gives the reading:

$$\exists t \in \overline{A}\ (t.f = A \wedge \forall s \in A\ (s.g \subseteq \overline{A} \wedge t \in s.g)).$$

The other reading tree shown in figure 9 gives an alternative semantic reading:

$$\forall s \in A\ (s.g \subseteq \overline{A}\ \wedge\ \exists t \in s.g\ (t.f = A)).$$

The distinction between these two readings is a typical example of the need to order quantifiers (see also the example in figure 1).

A *model* for d is an interpretation (which assigns sets to given contours and relations to arrows) satisfying the semantic statement constructed. A model conforming to one reading tree of d may not be a model for a different reading tree.

Different choices of depth-first search of the reading tree will give different semantic readings, but the readings will only differ by the logical equivalence $P \wedge Q \equiv Q \wedge P$.

Consider the interpretation for the diagram in figure 7 given by $\mathbf{U} = \{1,2,3,4\}$, $\Psi(A) = \{1,2\}$, $\phi(f) = \{(3,1),(3,2)\}$ and $\phi(g) = \{(1,3),(2,4)\}$. This is a model for the first reading

$$\exists t \in \overline{A}\ (t.f = A \wedge \forall s \in A\ (s.g \subseteq \overline{A} \wedge t \in s.g)).$$

but it is not a model for the second reading

$$\forall s \in A\ (s.g \subseteq \overline{A} \wedge \exists t \in s.g\ (t.f = A)).$$

The first reading begins with the existence of t. Choosing $t = 3$ gives $t.f = \{1,2\}$ and then if $s = 1$ then $s.g = \{3\}$ and if $s = 2$ then $s.g = \{4\}$. The second reading begins by taking any s in A. If $s = 1$ then $s.g = \{3\}$ and $3.f = A$, but if $s = 2$ then $s.g = \{4\}$ and $4.f \neq A$ – the interpretation fails to conform to the second semantics predicate.

11 Conclusion and further work

Formal semantics for constraint diagrams are a prerequisite for their safe use in software specification. In this paper, we have presented a reading algorithm which constructs the semantics of a constraint diagram, with respect to a reading tree. Certain computational aspects of the work, such as the development of efficient algorithms to generate all possible reading trees for a diagram have been investigated in [2].

We envisage the production of an environment (a collection of software tools) which will produce the semantics of a diagram, automating the resolution of possible ambiguities. To maintain the inherent usability of the notation, the environment should provide a single default reading of a diagram, but allow more confident users to specify alternate readings. Such an environment is already under construction [10]. Current work also involves investigating the notation both logically and empirically so that a unique intuitive default reading can be obtained.

Unambiguous semantics are an essential foundation for the development of a set of diagrammatic reasoning rules. These rules can be applied to reason about system specifications. For example, one can check structural subtyping: that the invariants of a subclass are stricter than the invariants of a superclass. The environment should guide a modeller through a reasoning process.

The formalisms required to underpin such an environment have been completed for simpler systems, and this paper constitutes a significant advance towards the required formalisms for constraint diagrams.

Acknowledgements This research was partially supported by UK EPSRC grant GR/R63516. We thank Gem Stapleton and John Taylor for comments on earlier drafts of this paper. We would also like to thank the anonymous referees for their very helpful comments.

References

[1] L. Euler. Lettres a une princesse d'allemagne. Letters Vol 2, No. 102-108, 1761.

[2] A. Fish and J. Howse. Computing Reading Trees for constraint diagrams. Submitted to AGTIVE03, 2003.

[3] J. Flower and J. Howse. Generating Euler diagrams. In *Proc. Diagrams 2002* pp 61–75. Springer-Verlag, 2002.

[4] J. Gil, J. Howse, and S. Kent. Towards a formalization of constraint diagrams. In *Proc. Symp. on Human-Centric Computing*, pp 72–79. IEEE Press, 2001.

[5] E. Hammer. *Logic and Visual Information.* CSLI Publications, 1995.

[6] J. Howse, F. Molina, and J. Taylor. SD2: A sound and complete diagrammatic reasoning system. In *Proc. Symp. on Visual Languages (VL2000)*, pp 127–136. IEEE Press, 2000.

[7] J. Howse, F. Molina, J. Taylor, S. Kent, and J. Gil. Spider diagrams: A diagrammatic reasoning system. *JVLC* 12, pp 299–324, 2001.

[8] J. Howse, G. Stapleton, J. Taylor, and J. Flower. Corresponding regions in Euler diagrams. In *Proc. Diagrams 2002*, pp 146–160. Springer-Verlag, 2002.

[9] S. Kent. Constraint diagrams: Visualising invariants in object oriented models. In *Proceedings of OOPSLA97, ACM SIGPLAN Notices*, 1997.

[10] Visual Modelling Group (VMG) at Brighton home page: www.cmis.brighton.ac.uk/Research/vmg.

[11] OMG. UML specification, version 1.4. Available from www.omg.org.

[12] S.-J. Shin. *The Logical Status of Diagrams.* Cambridge University Press, 1994.

[13] J. Venn. On the diagrammatic and mechanical representation of propositions and reasonings. *Phil.Mag*, 1880.

[14] J. Warmer and A. Kleppe. *The Object Constraint Language.* Addison-Wesley, 1998.