Transforming Constraint Diagrams

Jim Burton* Gem Stapleton†

Ali Hamie‡

Visual Modelling Group

University of Brighton, Brighton, UK

Abstract

Constraint diagrams were proposed by Kent for the purposes of formal software specification in a visual manner. They have recently been formalized and generalized, making them more expressive. This paper presents a collection of transformations that can be applied to the so-called unitary α fragment of constraint diagrams. The transformations can be used to define inference rules in a more succinct manner than in earlier systems. We establish that the transformations are sufficient to transform any given unitary α-diagram into any other unitary α-diagram. Therefore, they are sufficient for formalizing any inference rules between such diagrams.

1 Introduction

Visual languages play an important role in the design and implementation of software. For example, the Unified Modelling Language (UML) [20] is now an industry standard visual notation designed specifically for use by software engineers and is used throughout the software development process, from capturing domain requirements through to implementation. Under some circumstances (such as in a safety critical environment; see, for example, [19]) it is desirable, perhaps even essential, to produce formal models of software. In part, such application areas serve to motivate the need for the precise specification of the UML at both a syntactic and semantic level; the pUML group was set up with this goal in mind [18].

Part of the creation of a formal model is likely to involve specifying constraints such as system invariants and operation contracts which, within the UML, is achieved by using the Object Constraint Language (OCL) [22]. The OCL is the only purely textual part of the UML and, therefore, does not fit with the UML's diagrammatic theme. Building on the formal diagrammatic reasoning systems of Shin [13], Hammer [1] and

* `j.burton@brighton.ac.uk`

† `g.e.stapleton@brighton.ac.uk`

‡ `a.a.hamie@brighton.ac.uk`

others, Kent introduced constraint diagrams [11] which are designed to complement the visual components of the UML and to specify constraints like the (symbolic) OCL. Constraint diagrams can also be used independently of the UML.

In figure 1 there is a constraint diagram which expresses an invariant that we might wish to place on a video rental store system: there is a member that can only borrow films that are in the collections of the stores which they have joined. The semantics of constraint diagrams will be explained more fully later, but the blob acts as an existential quantifier, the arrows allow us to make statements about binary relations and the closed curves represent sets (or classes).

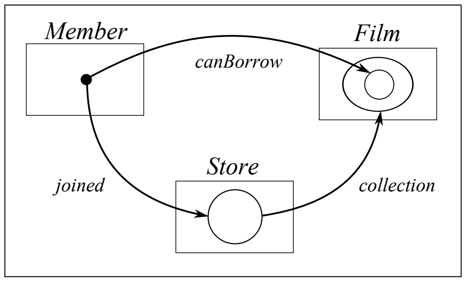

Figure 1: A constraint diagram.

At first glance, constraint diagrams appear intuitive and, perhaps, unambiguous, but it was not until a formalization of their semantics was attempted that a range of ambiguities was noticed [8]. Indeed, only when a formalization was eventually obtained [5] did the complexity of interpreting these diagrams become apparent. Whilst in many examples constraint diagrams are “well-matched to meaning” [9] there are also many situations where their intuitiveness breaks down. This led to the development of generalized constraint diagrams [14]. Both of these constraint diagram notations share a common fragment, which is considered in this paper. For this fragment, we set up a transformation system which forms the basis of a reasoning system for both constraint diagrams and generalized constraint diagrams.

There are various ways in which reasoning will need to be performed when using formal methods. First, there is reasoning about the model; for example, when one wishes to show that the model is consistent or that the post-condition of one operation implies the precondition of another. Secondly, a programmer will need to use some informal reasoning to determine an appropriate implementation that conforms to the specification. Thirdly, at a later stage, one might also wish to formally prove that the implementation does indeed conform to the model. Formal reasoning has been investigated for constraint diagrams [4, 16] but as yet no inference rules have been defined for generalized constraint diagrams.

An aim of this paper is to define a transformation system for so-called unitary α-diagrams which can be used to subsequently define inference rules for either constraint diagrams or generalized constraint diagrams. Section 2 provides a brief overview of unitary diagrams. We also present a formalization of the syntax of unitary diagrams in section 2. Our transformations are defined in section 3, focusing separately on those which remove syntax and those which add syntax. Finally, in section 4, we show how the transformations can be used as a basis for inference rules.

2 Unitary Diagrams

We follow a typical approach of formally defining the syntax at an abstract level [10]. In this way, we disregard the many aspects of drawn diagrams that are irrelevant to their semantic meaning, such as the shape and relative location of curves. To aid intuition, we include a informal presentation of the concrete syntax, since this is used to guide the work, but all formal aspects are conducted at the abstract level.

2.1 Concrete Syntax

The concrete syntax of a visual language defines, in our case, diagrams as drawn images. We proceed to sketch the concrete syntax of so-called unitary constraint diagrams. We make occasional reference to semantics to aid the readers' understanding. For the purposes of this paper, the semantics are not particularly important, which is why we do not include their precise formalization.

Unitary constraint diagrams consist of closed curves (some of which may be labelled) drawn in the plane and which represent sets. The spatial relationships between the curves makes assertions about the relationships between the represented sets. For example, the diagram in figure 1 contains six curves, three of which are labelled. The placement of one curve inside another makes a subset assertion, whilst non-overlapping curves make a disjointness assertion. So, $Member$ and $Film$ are disjoint, for example.

In the regions formed by the curves we can place graphs, whose nodes are either all dots or all asterisks; these graphs are called existential spiders and universal spiders respectively. Existential spiders represent the existence of an element. In figure 1, there is one existential spider that has exactly one node placed inside $Member$. For simplicity of presentation, we will assume there are no universal spiders, although the transformations we define can easily be extended to cope with their inclusion. Similarly, we also assume that the existential spiders are placed in single zones; this constraint to single zones gives what are called α-diagrams [16]. The curves in a diagram subdivide the plane into minimal regions: such a region is a connected component of the plane less the images of the curves. In figure 1, there are seven minimal regions. Of particular importance is the notion of a *zone* in a diagram, d. A zone is a set of minimal regions that can be described as being inside some (possibly no) curves but outside the rest of the curves in d. Semantically, a zone represents the set which is the intersection of the sets represented by the curves it is inside less the union of the sets represented by the curves that it is outside. In figure 1, every minimal region is also a zone and there are no zones that are not also minimal regions. However, this need not be the case: sometimes, zones consist of more than one minimal region and such zones are said to be disconnected. Zones can be shaded. The use of shading places an upper bound on the cardinality of the represented sets: in a shaded zone, all of the elements must be represented by spiders.

Finally, arrows are used to make statements about binary relations: the set of elements (or element) represented by the arrow's source is related to precisely the set of elements represented by the arrow's target under the relation represented by the arrow's label. For example, in figure 1 the arrow labelled *joined* sourced on the existential spider, e, asserts that the set of elements to which e is related under the relation *joined* is a subset of *Store*. In addition, if we restrict the domain of *collection* to *Store* then we obtain a subset of *Film* which includes all of the films that can be borrowed by e.

So far, we have described unitary diagrams which consist of curves, spiders placed in zones or sets of zones, shading, and arrows. Further examples of unitary diagrams can be seen throughout the paper; we discuss the syntax of d_1 when presenting the formalization below. We refer the reader to [5] for further examples and more precise details on the concrete syntax and the semantics of constraint diagrams, and to [14] for similar information on generalized constraint diagrams. For the purposes of this paper, it is the formal, abstract, syntax that is important and the next section includes those details necessary for our transformations to be defined. Unitary diagrams can be joined together using logical connectives, such as $\wedge$, to make compound diagrams; it is unitary diagrams for which we define transformations.

We have placed a restriction on spiders so that they can only have one node, meaning that they are placed inside single zones. This restriction yields α-diagrams which form a fragment of (non-unitary) generalized constraint diagrams that is not reduced in expressive power: given any generalized constraint diagram there exists a semantically equivalent diagram that contains only spiders placed inside single zones. However, there are constraint diagrams that are not semantically equivalent to any α-diagram, but only if they contain universal spiders. That is given a constraint diagram containing only existential spiders, one can reduce it to an α-diagram,as in [16]. We note that excluding universal spiders does decrease the expressive power but, as stated above, our work easily adapts to the case where they are permitted.

2.2 Abstract Syntax

Our formal definition of the syntax of unitary diagrams adapts that in [14]. In the abstract syntax we identify labelled curves with their labels; curve labels are drawn from the set $\mathcal{LC}$. Further, at the abstract level, the unlabelled curves are formalized as elements of an arbitrary (but specified) set $\mathcal{UC}$. We consider the elements of $\mathcal{UC}$ to correspond directly to the unlabelled curves of drawn diagrams. In a drawn diagram, a zone can be described by the curves that contain it and the curves that do not contain it. We use this insight to formalize zones at an abstract level.

Definition 2.1. *A* ***zone*** *is a pair,* (in, out) *where* $in \cap out = \emptyset$ *and* $in \cup out \subseteq \mathcal{LC} \cup \mathcal{UC}$.

The set of all zones is denoted $\mathcal{Z}$. To illustrate the concept, the shaded zone in figure 2 can be described by $z = (\{A\}, \{B, uc\})$ where uc denotes the unlabelled curve. There are two spiders placed in this zone; we cannot formalize a spider by identifying it with the zone in which it is placed. However, this provides the basis of their formalization: a spider will essentially be defined as a number together with a zone. In our example, the two spiders are written as $s_1(z)$ and $s_2(z)$.

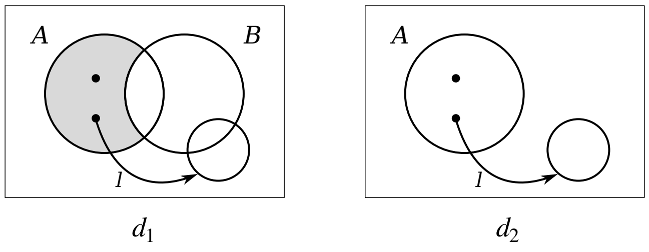

Figure 2: Formalizing the syntax.

Definition 2.2. *A **spider** is of the form $s_i(z)$ where i is a natural number and z is a zone. The **habitat** of $s_i(z)$ is z and we say that $s_i(z)$ **inhabits** z.*

The set of all spiders is denoted $\mathcal{S}$. We now proceed to formalize arrows. To identify the arrows in a drawn diagram, it is sufficient to state their source and target, together with their label. For example, in figure 2, the arrow can be described by the triple $(l, s_1(z), uc)$ (recall, uc is the unlabelled curve and $z = (\{A\}, \{B, uc\})$). We draw arrow labels from a fixed set $\mathcal{AL}$.

Definition 2.3. *An **arrow end** is either a curve drawn from $\mathcal{LC} \cup \mathcal{UC}$ or a spider drawn from $\mathcal{S}$. An **arrow** is an ordered triple (l, s, t) where $l \in \mathcal{AL}$, and s and t are arrow ends called the **source** and **target** respectively.*

Definition 2.4. *A **unitary diagram** is a tuple, $d = (Z, Z^*, S, A)$, which satisfies the following:*

1. *$Z = Z(d)$ is a finite set of zones such that for each pair of zones (in_1, out_1) and (in_2, out_2) in $Z(d)$ we have $in_1 \cup out_1 = in_2 \cup out_2$. That is, the zones are all described using the same curves. We define $C(d) = in_1 \cup out_1$.*

2. *$Z^* = Z^*(d)$ is a set of shaded zones such that $Z^*(d) \subseteq Z(d)$. That is, all of the shaded zones are in the diagram.*

3. *$S = S(d)$ is a finite set of spiders such that for each spider $s_i(z) \in S(d)$, $z \in Z(d)$. That is, spiders are placed in zones of the diagram.*

4. *$A = A(d)$ is a set of arrows such that for each arrow (l, s, t) in $A(d)$, s and t are in $S(d) \cup C(d)$. That is, arrows are sourced and targeted on components of the diagram.*

So, d_1 in figure 2 is formalized as the tuple (Z, Z^*, S, A) where:

1. Z is comprised of the following zones.
   - $(\{A\}, \{B, uc\})$,
   - $(\{A, B\}, \{uc\})$,
   - $(\{B\}, \{A, uc\})$,

- $(\{B, uc\}, \{A\})$,
- $(\{uc\}, \{A, B\})$,
- $(\emptyset, \{A, B, uc\})$

2. $Z^* = \{(\{A\}, \{B, uc\})\}$,
3. $S = \{s_1(\{A\}, \{B, uc\}), s_2(\{A\}, \{B, uc\})\}$, and
4. $A = \{(l, s_1(\{A\}, \{B, uc\}), uc)\}$.

Semantically, d_1 asserts that the set $A - B$ contains at least two elements, x and y, through the use of the two existential spiders, the shading asserts that there are no more elements in that set (i.e. $|A - B| = 2$), and that x is related to some set of elements, say $x.l$, under the relation l such that $x.l \cap A = \emptyset$. The diagram d_2 makes a weaker statement, asserting that there are at least two elements in A, at least one of which is related to some set of elements, under l, that is disjoint from A. In fact, we can deduce d_2 from d_1 and, if we had a set of sound and (possibly) complete inference rules then we could prove that d_2 does indeed follow semantically from d_1.

3 Transformations

To facilitate elegant definitions of inference rules for unitary diagrams, we define *diagram transformations*, which are purely syntactic and represent the addition or removal of a piece of syntax. For example, we can remove the curve B from d_1 in figure 2, transforming it into d_2; this remove curve transformation will be formalized below. The transformations defined will be applicable under specified syntactic conditions, which are not related to sound reasoning, but are intended to merely constrain the transformation to ensure the result of its application is a diagram. The benefit of making transformations which are purely syntactic and unrelated to reasoning is that this facilitates their use in a wide number of (reasoning) contexts.

3.1 Transformations that remove syntax

We start with the simplest transformation, that which removes an arrow.

Transformation 1. Remove arrow

We can transform a diagram by removing an arrow. In figure 3, the arrow, a, labelled r is removed from d_1 to give d_2.
Formal definition Let d_1 be a unitary diagram such that there exists an arrow a in $A(d_1)$. The diagram d_2 can be obtained from d_1 removing a using the remove arrow transformation, denoted $d_1 \xrightarrow{-a} d_2$, where $d_2 = (Z(d_1), Z^*(d_1), S(d_1), A(d_1) - \{a\})$.

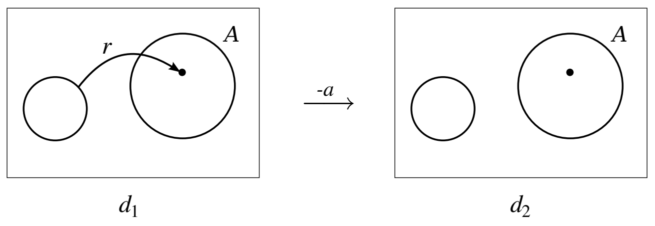

Figure 3: Transforming a diagram by removing an arrow.

Transformation 2. Remove shading

We can transform a diagram by removing the shading from a zone. In figure 4, the shading is removed from the zone $(\{B\}, \{A\})$ in d_1 to give d_2.

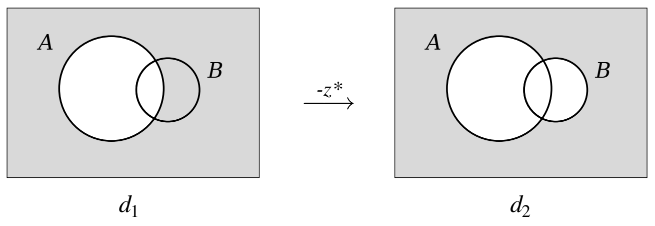

Figure 4: Transforming a diagram by removing shading from a zone.

Formal definition Let d_1 be a unitary diagram and let z be a zone such that $z \in Z^*(d_1)$. The diagram d_2 can be obtained from d_1 using the remove shading transformation, denoted $d_1 \xrightarrow{-z^*} d_2$, where $d_2 = (Z(d_1), Z^*(d_1) - \{z\}, S(d_1), A(d_1))$.

Transformation 3. Remove spider

Our next transformation removes a spider from a unitary diagram. We need to provide a constraint (i.e. a precondition) on when this transformation can be applied in order to ensure that the result is a diagram. To formally define the remove spider transformation, we need to refer to the set of arrows sourced on, or targeting, a spider. Later, we also need to identify curves that are the source or target of an arrow. Here, we provide some notation that is convenient for identifying these sets.

Definition 3.1. *Let d be a unitary diagram and let s be a spider in $S(d)$. The set of arrows which are either sourced or targeted on s in d, denoted $A(s, d)$, is*

$$A(s, d) = \{(l, \sigma, \tau) \in A(d) : \sigma = s \vee \tau = s\}.$$

*If an arrow a is sourced or targeted on s then we say a **touches** s. Similarly, we define the set of arrows which touch a curve c in a diagram d, denoted $A(c, d)$:*

$$A(c, d) = \{(l, \sigma, \tau) \in A(d) : \sigma = c \vee \tau = c\}.$$

We can transform diagrams by removing a spider provided it is not touched by an arrow. In figure 5 s is removed from d_1 to give d_2.

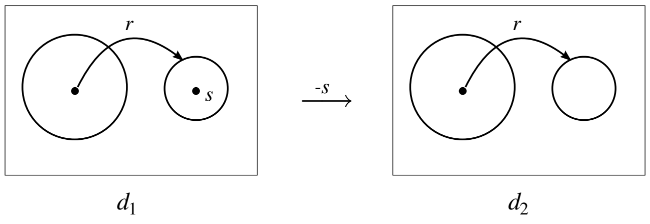

Figure 5: Transforming a diagram by removing a spider.

Formal definition Let d_1 be a unitary diagram such that there exists a spider $s \in S(d_1)$ which is not touched by any arrow, that is $A(s,d) = \emptyset$. The diagram d_2 can be obtained from d_1 by removing s under the remove spider transformation, denoted $d_1 \xrightarrow{-s} d_2$, where $d_2 = (Z(d_1), Z^*(d_1), S(d_1) - \{s\}, A(d_1))$.

Transformation 4. Remove zone

We can transform a diagram by removing any zone which is not the habitat of any spider. In figure 6 the zone $(\{A, B\}, \emptyset)$ is removed from d_1 to give d_2.

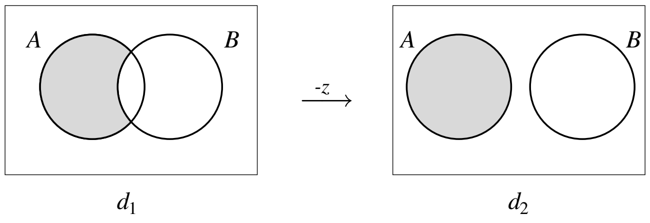

Figure 6: Transforming a diagram by removing a zone.

Formal definition Let d_1 be a unitary diagram such that there exists a zone z in $Z(d_1)$ which is not the habitat of any spider. Then the diagram d_2 can be obtained from d_1 by removing z under the remove zone transformation, denoted $d_1 \xrightarrow{-z} d_2$, where $d_2 = (Z(d_1) - \{z\}, Z^*(d_1) - \{z\}, S(d_1), A(d_1))$.

Transformation 5. Remove curve

We can remove a curve provided it is not touched by any arrow. In figure 7 the curve labelled B is removed from d_1 to give d_2.

In diagram d_1 in figure 7, the region inside the curve labelled A is partially shaded. We could choose to define the transformation which removes B so that it removes this partial shading or leaves as shaded all zones which were shaded in the original. Actually, there are various choices for how to define a remove curve rule. We want to

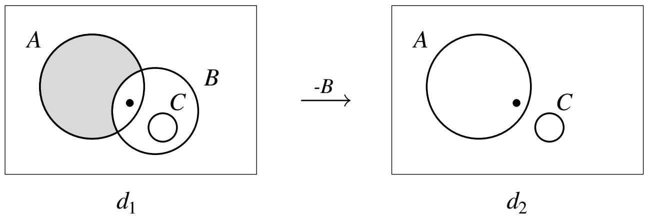

Figure 7: Transforming a diagram by removing a curve.

use the transformations as the basis for (useful) inference rules, so we have chosen to define this transformation in such a manner that it removes partial shading: we know that $B - A$ represents the empty set through the use of this shading and, when deleting B, we forget this information. In figure 8, the curve B can be removed, but this time we retain the shading in A. We need our formalization to reflect when we must lose shading and when we can retain it; for this purpose we need to appeal to *missing zones*.

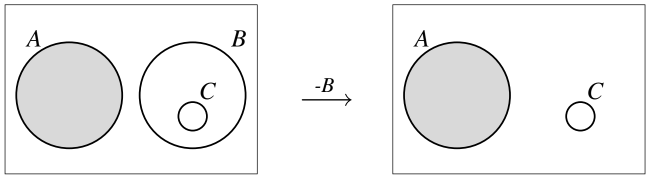

Figure 8: Accounting for missing zones.

Definition 3.2. *Let d be a unitary diagram. A zone $z = (in, out)$ where $in \cup out = C(d)$ that is not in $Z(d)$ is said to be* ***missing*** *from d. The set of zones missing from d is denoted $MZ(d)$, so $MZ(d) = \{(in, out) \in \mathcal{Z} : in \cup out = C(d)\} - Z(d)$.*

Formal definition Let d_1 be a unitary diagram and let c be a curve such that $c \in C(d_1)$ and $A(c, d_1) = \emptyset$. The diagram d_2 can be obtained from d_1 by removing c using the remove curve transformation, denoted $d_1 \xrightarrow{-c} d_2$, where d_2 has the following components.

1. $Z(d_2) = \{(in - \{c\}, out - \{c\}) : (in, out) \in Z(d_1)\}$,
2. $Z^*(d_2)$ is the union of the sets of zones $Z_{i,o}$, $Z_{i,m}$, and $Z_{m,o}$ where
   (a) $Z_{i,o}$ is formed by removing c from the shaded zones of d_1 that were split by c into two zones (one inside and one outside):
   $$Z_{i,o} = \{(in, out) : (in \cup \{c\}, out) \in Z^*(d_1) \wedge (in, out \cup \{c\}) \in Z^*(d_1)\},$$
   (b) $Z_{i,m}$ is formed by removing c from the shaded zones of d_1 that c was entirely within:
   $$Z_{i,m} = \{(in, out) : (in \cup \{c\}, out) \in Z^*(d_1) \wedge (in, out \cup \{c\}) \in MZ(d_1)\},$$

(c) $Z_{m,o}$ is formed by removing c from the shaded zones of d_1 that c was entirely outside:

$$Z_{m,o} = \{(in, out) : (in \cup \{c\}, out) \in MZ(d_1) \wedge (in, out \cup \{c\}) \in Z^*(d_1)\},$$

3. $S(d_2) = \{s_i(in - \{c\}, out - \{c\}) : s_i(in, out) \in S(d_1)\}$,
4. $A(d_2) = A(d_1)$.

3.2 Transformations that add syntax

The transformations that we define for adding syntax are counterparts of those which remove syntax. For the first two transformations no examples are given since they are very similar to their remove syntax counterparts.

Transformation 6. Add arrow

Formal definition Let d_1 be a unitary diagram and let (l, s, t) be an arrow such that $s, t \in S(d) \cup C(d)$ and $(l, s, t) \notin A(d_1)$. The diagram d_2 can be obtained by adding (l, s, t) to d_1 using the add arrow transformation, denoted $d_1 \xrightarrow{+a} d_2$, where $d_2 = (Z(d_1), Z^*(d_1), S(d_1), A(d_1) \cup \{(l, s, t)\})$.

Transformation 7. Add spider

Formal definition Let d_1 be a unitary diagram such that there exists a zone $z \in Z(d_1)$ and a spider $s_i(z) \notin S(d_1)$ where $z \in Z(d_1)$. The diagram d_2 can be obtained by adding s to d_1 using the add spider transformation, denoted $d_1 \xrightarrow{+s} d_2$, where $d_2 = (Z(d_1), Z^*(d_1), S(d_1) \cup \{s\}, A(d_1))$.

Transformation 8. Add shading

We can transform a diagram by adding shading to a zone. In figure 9, shading is added to the zone $(\{A, C\}, \{B\})$ in d_1 to give d_2.

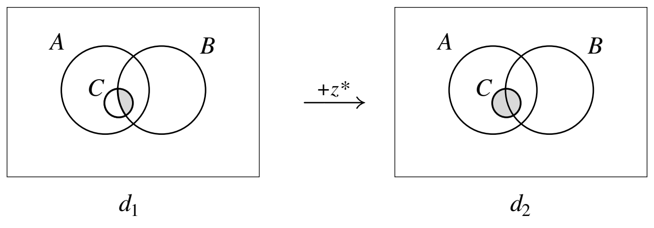

Figure 9: Transforming a diagram by adding shading to a zone.

Formal definition Let d_1 be a unitary diagram and $z \in Z(d_1) - Z^*(d_1)$. The diagram d_2 can be obtained from d_1 by adding shading to z using the add shading transformation, denoted $d_1 \xrightarrow{+z^*} d_2$, where $d_2 = (Z(d_1), Z^*(d_1) \cup \{z\}, S(d_1), A(d_1))$.

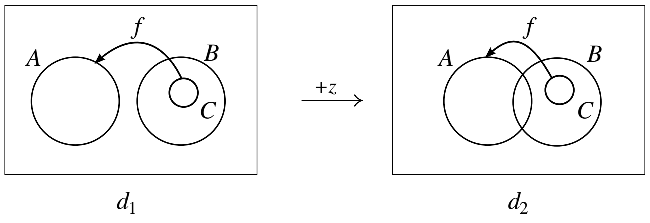

Figure 10: Transforming a diagram by adding a zone.

Transformation 9. Add zone

We can transform a diagram by adding a missing zone. Figure 10 shows the addition of the missing zone $(\{A, B\}, \{C\})$ to d_1 to give d_2.
Formal definition Let d_1 be a unitary diagram and z be a zone such that $z \in MZ(d_1)$. The diagram d_2 can be obtained from d_1 by adding z using the add zone transformation, denoted $d_1 \xrightarrow{+z} d_2$, where $d_2 = (Z(d_1) \cup \{z\}, Z^*(d_1), S(d_1), A(d_1))$.

Transformation 10. Add curve

There are a number of ways of adding a curve to a diagram: the new curve can be added in such a way that it is entirely outside of all existing curves, or is entirely contained within one other curve, and so on. The relationship between the new curve and the existing curve can be captured by appealing to its relationship with the existing zones: the existing zones are either completely inside the new curve, completely outside the new curve, or split by the new curve. Thus, we parametrise the transformation of adding a curve to a diagram d_1 using two subsets of zones which we call Z_{in} and Z_{out}, where $Z_{in} \cup Z_{out} = Z(d_1)$; those zones which will fall inside the new curve are in Z_{in}, those outside in Z_{out} and those that will be split are in $Z_{in} \cap Z_{out}$. The case of adding a curve which splits every zone in d_1, for instance, is that of choosing $Z_{in} = Z_{out} = Z(d_1)$. Figure 11 shows an example of adding a curve with $Z_{in} = \{(\emptyset, \{A, B\})\}$ and $Z_{out} = Z(d_1) - Z_{in}$.

In addition, each spider can be inside or outside the new curve. Thus, we also supply a two-way partition of the spider set, S_{in} and S_{out}, allowing us to specify the habitats of the spiders after the curve addition. We must place constraints on the choice of S_{in} and S_{out} to ensure consistency with the manner in which the curve is added. For instance, we cannot place a spider $s_i(z)$ in the set S_{in} if $z \in Z_{out} - Z_{in}$, since the 'new' habitat of the spider will not be present in the diagram after the curve addition. Consequently, we only have a choice about whether a spider, $s_i(z)$, is in S_{in} or S_{out} if $z \in Z_{in} \cap Z_{out}$. Note that this is a syntactic constraint and not related to soundness. An appropriate choice of Z_{in}, Z_{out}, S_{in} and S_{out} allows the user to add curves in any of the possible ways.

Recall that the set $\mathcal{LC} \cup \mathcal{UC}$ is the abstract set that corresponds to the labelled and unlabelled curves that can appear in any diagram at the concrete syntax level. The set $C(d)$, for any unitary diagram d, is a subset of $\mathcal{LC} \cup \mathcal{UC}$.

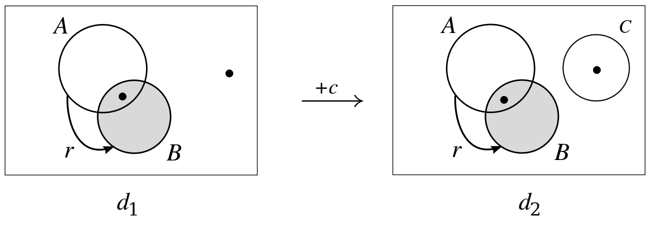

Figure 11: Transforming a diagram by adding a curve.

Formal definition Let d_1 be a unitary diagram and let c be a curve that is not in d_1, that is $c \in (\mathcal{LC} \cup \mathcal{UC}) - C(d)$. Let Z_{in} and Z_{out} be subsets of $Z(d_1)$ such that $Z_{in} \cup Z_{out} = Z(d_1)$. Let S_{in} and S_{out} be a two-way partition of $S(d_1)$ such that

1. for all $s_i(z)$ in S_{in}, $z \in Z_{in}$ and
2. for all $s_i(z)$ in S_{out}, $z \in Z_{out}$.

The diagram d_2 can be obtained by adding the curve c to d_1 using the add curve transformation, denoted $d_1 \xrightarrow{+P} d_2$, where $P = (c, Z_{in}, Z_{out}, S_{in}, S_{out})$ and d_2 has the following components:

1. $Z(d_2) = Z_{in+c} \cup Z_{out+c}$ where
   (a) Z_{in+c} is formed by adding c to the zones of d_1 that we wish to contain c in d_2: $Z_{in+c} = \{(in \cup \{c\}, out) : (in, out) \in Z_{in}\}$,
   (b) Z_{out+c} is formed by adding c to the zones of d_1 that we wish to exclude c in d_2: $Z_{out+c} = \{(in, out \cup \{c\}) : (in, out) \in Z_{out}\}$.
2. $Z^*(d_2) = Z^*_{in+c} \cup Z^*_{out+c}$ where
   (a) $Z^*_{in+c} = \{(in \cup \{c\}, out) : (in, out) \in Z_{in} \cap Z^*(d_1)\}$,
   (b) $Z^*_{out+c} = \{(in, out \cup \{c\}) : (in, out) \in Z_{out} \cap Z^*(d_1)\}$.
3. $S(d_2) = S_{in+c} \cup S_{out+c}$ where
   (a) $S_{in+c} = \{s_i(in \cup \{c\}, out) : s_i(in, out) \in S_{in}\}$,
   (b) $S_{out+c} = \{s_i(in, out \cup \{c\}) : s_i(in, out) \in S_{out}\}$.
4. $A(d_2) = A(d_1)$.

3.3 Completeness of the Transformations

The set of transformations defined above is complete because we can use them to transform any unitary diagram into any other unitary diagram, although the resulting system is (intentionally) not sound. The completeness of the transformation system means that these transformations are sufficient for describing a set of sound and complete inference rules for the unitary α-diagram fragment of both constraint diagrams and generalized constraint diagrams.

Theorem 1. *Let d_1 and d_n be unitary diagrams. Then there exists a sequence of diagrams, $(d_1, d_2 ..., d_n)$ such that for each i, where $1 < i \leq n$, the diagram d_i can be obtained from d_{i-1} by the application of one of the above transformations. In other words, the given set of transformations is complete.*

Sketch. The transformations which remove syntax can be used repeatedly to transform d_1, regardless of its content, into the diagram $(\emptyset, \emptyset, \emptyset, \emptyset)$. The transformations which add syntax can then be used to build d_2. $\square$

Although there will often be faster ways to transform one diagram into another, we can rely on the 'brute force' method of removing all diagrammatic elements from d_1 to produce the empty diagram then adding the elements of d_2. This relies on choosing the right order in which to apply the transformations, depending on their pre-conditions of syntactic well-formedness; for instance, before using remove spider (transformation 3) to remove a spider s which is the source of an arrow a in d_1, we must first use remove arrow (transformation 1) to remove a from d_1.

4 Using Transformations to Define Inference Rules

We are able to use the transformations to define inference rules in a variety of ways. The motivation for using transformations in this way is by analogy with software engineering. Functional and modular abstraction leads to systems with less code duplication and which are easier to understand and maintain. In a similar way, the shorter inference rule definitions that result from abstracting syntactic details into transformations are easier to state, reason about and check for errors. We are able to compose transformations in the definition of rules which make several changes to a diagram in a similar way to composing referentially transparent functions in a functional programming language. To use the transformations when defining inference rules, we may need to place further conditions on when the transformation can be applied to ensure soundness. In the case of the erasure of a spider, such a condition would be that the habitat is not shaded.

We have defined a set of sound inference rules for the unitary α fragment of generalized constraint diagrams, and present three of their definitions below as examples. Work on establishing a complete set for this fragment is ongoing. Four transformations can immediately be used as sound inference rules: remove arrow, remove shading, remove curve and add zone. The other transformations need not result in a semantic consequence of the diagram to which they are applied. Some transformations are used by several rules; for instance, (at least) five of inference rules that we have so far defined use the add arrow transformation. An *add shaded zone* inference rule makes use of two transformations. We have also begun work on defining inference rules whose application results in a compound diagram, and expect a significant proportion of these inference rules to use more than one of the transformations defined here, particularly as compared to the unitary fragment, because the compound inference rules tend to be more complex.

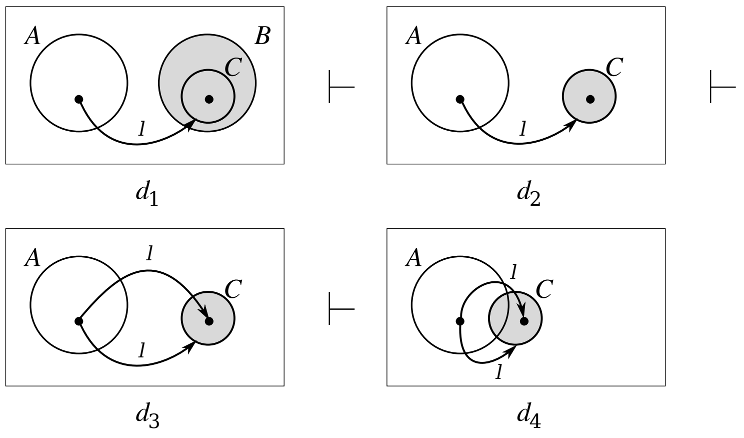

Figure 12: Using transformations to define inference rules.

To illustrate how we use the transformations to define sound inference rules, we consider an example. Figure 12 shows a proof of d_4 from d_1. First, we apply the remove curve transformation to d_1 giving d_2. We note that the remove curve transformation can be used directly as an inference rule; that is, applying the remove curve transformation always results in a semantic consequence of the diagram to which the transformation is applied. Next, we apply an add arrow rule to give d_3. Unlike the remove curve transformation, we cannot add arrows in arbitrary ways and obtain a semantic consequence. The information provided by the new arrow must be deducible from the information already present in the diagram and, as stated, we have defined a number of rules which add arrows. The rule used to obtain d_3 from d_2 is called *Add arrow: contour to spider*. Before defining the rule, we define the *empty* curves of a diagram, or those within which every zone is shaded.

Definition 4.1. *Let d be a unitary diagram. Define the* ***empty curves*** *of d, denoted $EC(d)$, as follows.*

$$EC(d) = \{c \in C(d) : \forall (in, out) \in Z(d)\ c \in in \Rightarrow (in, out) \in Z^*(d)\}.$$

Definition 4.2. ***Add arrow: contour to spider.*** *Let d_1 be a unitary diagram such that:*

1. *there is an arrow (l, s, t) in $A(d_1)$ such that $t \in EC(d_1)$, and*
2. *there is a spider $\sigma \in S(d_1)$ such that $S(t, d_1) = \{\sigma\}$.*

Let d_2 be the diagram obtained by adding the arrow (l, s, σ) to d_1 using the add arrow transformation. Then we may replace d_1 with d_2.

As an example of the reuse of transformations, we include a second inference rule which adds an arrow to a diagram. Informally, diagram d_1 in figure 13 tells us that the sum of the images of the relation r when restricted to the elements of A is the empty set. This information is provided by the arrow labelled r which is sourced on A and targets the shaded and unlabelled curve. It follows that we can add an arrow with the same source and label which targets any other empty curve without changing the meaning of the diagram. The inference rule *Add arrow: empty set* allows us to do this and is used in d_2 to add an arrow with the same source and label as the arrow in d_1 but which targets B.

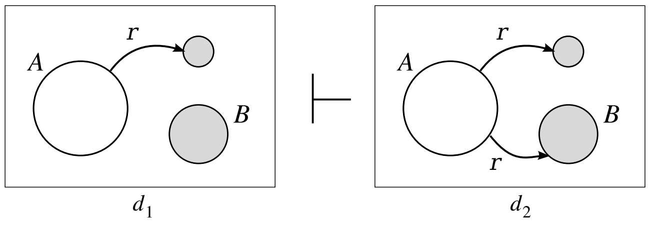

Figure 13: An application of the inference rule *Add arrow: empty set*.

Definition 4.3. ***Add arrow: empty set.*** *Let d_1 be a unitary diagram which satisfies:*

1. *there is an arrow (l, s, t) in $A(d_1)$ where $t \in EC(d_1)$,*
2. *there is a curve $t_1 \in EC(d_1)$ where $(l, s, t_1) \notin A(d_1)$.*

Let d_2 be the diagram obtained by adding the arrow (l, s, t_1) to d_1 using the add arrow transformation. Then we may replace d_1 with d_2.

Returning to figure 12, we obtain d_4 from d_3 using a combination of two transformations: add zone and add shading. The diagram d_3 asserts that $A \cap C = \emptyset$ since A and C do not overlap, but we can assert this disjointness using a shaded zone, namely $(\{A, C\}, \emptyset)$, justifying that d_4 is a semantic consequence of d_3. This intuition is formalized in the following inference rule.

Definition 4.4. ***Add shaded zone.*** *Let d_1 be a unitary diagram and z be a zone such that $z \in MZ(d_1)$. Let d_2 be the diagram obtained by applying the add zone transformation to add z to d_1 obtaining d_1', then applying the add shading transformation to shade z in d_1' obtaining d_2. Then we can replace d_1 by d_2.*

5 Extending to Compound Diagrams

The defined transformations focus on unitary diagrams. In both constraint diagrams and generalized constraint diagrams, logical operators are used to form so-called compound diagrams, albeit in rather different manners in the two notations. Our transformations can also be used in the context of compound constraint diagrams and generalized constraint diagrams. For instance, figure 14 shows two compound constraint diagrams, the second of which is a consequence of the first. In the first diagram we have two unitary diagrams joined by a $\wedge$ connective and, in one of those diagrams, d_1, an arrow labelled r is shown. The diagram d_2 includes the source and target of the arrow in d_1 but not the arrow itself. Applying the add arrow transformation to add this arrow to d_2 is a sound inference step and results in the second compound diagram.

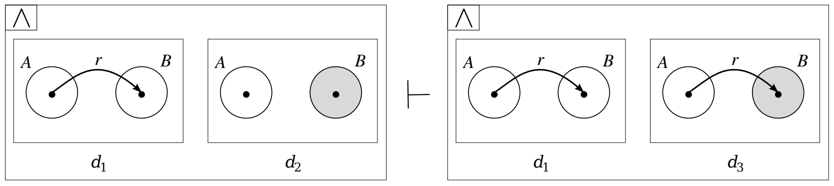

Figure 14: A constraint diagram and the add arrow transformation.

Since non-unitary inference rules sometimes have more complex postconditions, they are more likely to make use of several syntactic transformations than are unitary inference rules. An illustration of this is is given by the rule *excluded middle for zones*, adapted from [16]. This rule states that an unshaded zone either contains exactly the number of spiders depicted, or the zone must contain at least one more spider than depicted. Therefore, the conclusion of the rule is a disjunction of two diagrams, where the add spider transformation had been applied to the first, and the add shading transformation applied to the second. An example is shown in figure 15. The diagrams d_2 and d_3 are copies of d_1 except that transformations have been used to add a spider to zone $\{A, B\}$ in the first and to shade the same zone in the second.

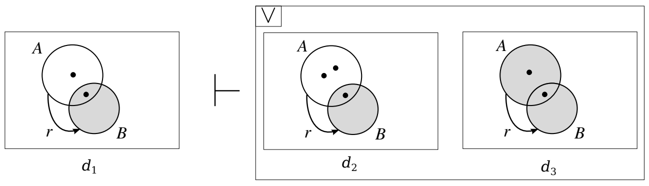

Figure 15: Illustrating *Excluded middle for zones*.

Definition 5.1. ***Excluded middle for zones.*** *Let d_0 be a unitary constraint diagram, let z be a non-shaded zone in d_0, or $z \in Z(d_0) - Z^*(d_0)$, and let s be a spider not in $S(d_0)$. Let d_1 be the diagram obtained by using the add spider transformation to add s to z in d_0, and let d_2 be the diagram obtained by using the add shading transformation to add shading to z in d_0. Then d_0 can be replaced by $d_1 \vee d_2$.*

6 Conclusion

In this paper we have presented a series of transformations that can be applied to unitary diagrams (either constraint diagrams or their generalized form) and established that they are complete. They provide a basis for defining inference rules, as illustrated in section 4, for both constraint diagrams and generalized constraint diagrams. Defining transformations with the right level of generality allows us to use them flexibly in the definition of inference rules. In the future, we plan to use these transformations to build a sound and, ideally, complete reasoning system for generalized constraint diagrams.

We are in the process of creating a proof assistant for reasoning with the abstract syntax of constraint diagrams [2] which is intended to form a flexible basis for graphical tools. The implementation of the tool uses the notion of modular, purely syntactic transformations combined with preconditions to form inference rules in a manner very close to the abstract definitions of the rules. This might in fact be called a "traditional" software engineering solution to the problem of creating such a tool. This close symmetry and the fact that the tool uses a dependently typed language to create types which correspond directly to abstract diagrams, transformations and inference rules, helps when establishing the correctness of the tool.

Also in the context of tool support, significant research has been directed towards the automated generation and layout of Euler diagrams, which form the bases of constraint diagrams, including [3, 6, 15, 21]. Moreover, other work has focused on how to add spiders to the drawn Euler diagrams [12]. Thus, much work has been conducted on how to automatically draw concrete diagrams from their abstract syntax. This diagram drawing functionality provides a basis for making interactive proof assistants and theorem provers accessible to a range of users, not just those familiar, and confident with using, the abstract syntax. Already, fully automated theorem provers have been developed for Euler diagrams [17] and spider diagrams [7]. Thus, whilst significant further work is required to develop tool support for constraint diagrams, there is already a firm basis on which we can build.

Acknowledgements. Jim Burton thanks the UK EPSRC for support through a DTA studentship. Gem Stapleton acknowledges support from the UK EPSRC, grant number EP/E011160/1, for the Visualization with Euler diagrams project (www.eulerdiagrams.com).

References

[1] Barwise, J. and E. Hammer, *Diagrams and the concept of logical system*, in: G. Allwein and J. Barwise, editors, *Logical Reasoning with Diagrams*, Oxford University Press, 1996 .

[2] Burton, J., *Diagrams and intuitive formal specifications*, in: P. Bottoni, M. B. Rosson and M. Minas, editors, *Visual Languages and Human-Centric Computing*, IEEE (2008), pp. 262–263.

[3] Chow, S. and F. Ruskey, *Drawing area-proportional Venn and Euler diagrams*, in: *Proceedings of Graph Drawing 2003, Perugia, Italy*, LNCS **2912** (2003), pp. 466–477.

[4] Fish, A. and J. Flower, *Investigating reasoning with constraint diagrams*, in: *Visual Language and Formal Methods 2004*, ENTCS **127** (2005), pp. 53–69.

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

[6] Flower, J. and J. Howse, *Generating Euler diagrams*, in: *Proceedings of 2nd International Conference on the Theory and Application of Diagrams* (2002), pp. 61–75.

[7] Flower, J., J. Masthoff and G. Stapleton, *Generating readable proofs: A heuristic approach to theorem proving with spider diagrams*, in: *Proceedings of 3rd International Conference on the Theory and Application of Diagrams*, LNAI **2980** (2004), pp. 166–181.

[8] Gil, J., J. Howse and S. Kent, *Towards a formalization of constraint diagrams*, in: *Proc IEEE Symposia on Human-Centric Computing (HCC '01), Stresa, Italy* (2001), pp. 72–79.

[9] Gurr, C. and K. Tourlas, *Towards the principled design of software engineering diagrams*, in: *Proceedings of 22nd International Conference on Software Engineering* (2000), pp. 509–518.

[10] Howse, J., F. Molina, S. J. Shin and J. Taylor, *On diagram tokens and types*, in: *Proceedings of 2nd International Conference on the Theory and Application of Diagrams* (2002), pp. 146–160.

[11] Kent, S., *Constraint diagrams: Visualizing invariants in object oriented modelling*, in: *Proceedings of OOPSLA97* (1997), pp. 327–341.

[12] Mutton, P., P. Rodgers and J. Flower, *Drawing graphs in Euler diagrams*, in: *Proceedings of 3rd International Conference on the Theory and Application of Diagrams*, LNAI **2980**, pp. 66–81.

[13] Shin, S. J., "The Logical Status of Diagrams," CUP, 1994.

[14] Stapleton, G. and A. Delaney, *Evaluating and generalizing constraint diagrams*, Journal of Visual Languages and Computing **19** (2008), pp. 499–521.

[15] Stapleton, G., J. Howse, P. Rodgers and L. Zhang, *Generating euler diagrams from existing layouts*, in: *Layout of (Software) Engineering Diagrams*, Electronic Communications of the EASST (2008), pp. 16–31.

[16] Stapleton, G., J. Howse and J. Taylor, *A decidable constraint diagram reasoning system*, Journal of Logic and Computation **15** (2005), pp. 975–1008.

[17] Stapleton, G., J. Masthoff, J. Flower, A. Fish and J. Southern, *Automated theorem proving in Euler diagrams systems*, Journal of Automated Reasoning **39** (2007), pp. 431–470.

[18] The Precise UML Group, *Untitled*, http://www.cs.york.ac.uk/puml/index.html (1997).

[19] UK Ministry of Defence, *The procurement of saftey critical software in defence equipment* (1993).

[20] Unified Modelling Language, *Untitled*, http://www.uml.org/ (2006).

[21] Verroust, A. and M. L. Viaud, *Ensuring the drawability of Euler diagrams for up to eight sets*, in: *Proceedings of 3rd International Conference on the Theory and Application of Diagrams*, LNAI **2980** (2004), pp. 128–141.

[22] Warmer, J. and A. Kleppe, "The Object Constraint Language: Precise Modeling with UML," Addison-Wesley, 1998.