SOSYM manuscript No.
(will be inserted by the editor)

Spider Graphs: A Graph Transformation System for Spider Diagrams

Paolo Bottoni[1], **Andrew Fish**[2], **Francesco Parisi Presicce**[1]

[1] Department of Computer Science, "Sapienza" University of Rome, Italy
e-mail: (bottoni,parisi)@di.uniroma1.it

[2] School of Computing, Engineering and Mathematics, University of Brighton, UK
e-mail: Andrew.Fish@brighton.ac.uk

Received: July 9, 2013/ Accepted:

Abstract The use of diagrammatic logic as a reasoning mechanism to produce inferences on subsets of some universe could provide a way to overcome the current limitations of visual modeling methods, which have to be integrated with textual languages to express complex constraints. On the other hand, graph transformations are becoming widespread as a way to express formal semantics for visual modeling languages, so that a mechanisation of diagrammatic logic based on graph transformation would facilitate language integration, based on a common underlying machinery. In this paper, we propose such a mechanisation for Spider Diagrams (SDs), an established language for reasoning with diagrams modeling relations between sets and constraints on their cardinalities. The concrete syntax of SDs extends that of Euler diagrams which use closed curves and the enclosed regions to represent sets and their intersections. The language is augmented with *reasoning rules*, i.e. syntactic transformation rules corresponding to logical inference rules. However, these rules are typically defined in procedural terms, so that a completely formal specification and an adequate mechanisation of them has not been achieved yet. We propose an abstract syntax for SDs in terms of typed graphs, and define the corresponding language of Spider Graphs (SGs), expressing reasoning rules for SDs as graph transformation units. This enables a direct realisation of the reasoning system via graph transformation tools without resorting to ad-hoc implementations, and we provide an implementation in AGG. Techniques for static analysis become available to reason on proof strategies and on possible optimisations.

Keywords Diagrammatic Reasoning - Graph Transformations - Spider Diagrams - Spider Graphs - Reasoning Strategies

1 Introduction

Euler Diagrams (EDs) are a well-known formal notation for modeling sets and their relationships. As a representational device, EDs are a variation of the Euler circles developed to represent syllogistic reasoning [17]. EDs generalise *Venn Diagrams* (VDs) in that they do not require every possible set intersection to be displayed (for a survey on VDs see [43]).

By adopting a model-theoretic semantics and defining transformation (reasoning) rules on a class of diagrams, diagrammatic logics can be exploited as a reasoning mechanism to produce inferences on subsets of some universe. The area was established by Shin, Hammer, Barwise and Etchmendy [2, 29, 44] and has been rapidly expanding in recent years.

Visual reasoning would also be beneficial for the Unified Modeling Language (UML), which is currently limited in its capability of expressing complex constraints, for which one needs the textual Object Constraint Language (OCL) [53], requiring modellers to deal with different languages, rather than with a completely visual representation. Research is currently active on the definition of automatic checks on model satisfaction, as reported on in Section 2, usually involving mapping to some textual representation language for expressing transformations. An advantage of diagrammatic reasoning systems is that they enable the presentation of visual proofs of correctness. As an example, one can express the post-condition of a contract for method `a` as a diagram d_1 and the precondition for a method `b` as d_2. Then a sequence of diagrams connecting d_1 to d_2 where consecutive diagrams differ in a manner corresponding to a logical inference, constitutes a visual proof that the execution of `a` can be followed by that of `b`.

Spider Diagrams (SDs) [32, 33] extend EDs through additional syntax for the representation of constraints on set cardinality, enabling diagrammatic inferences not normally considered in symbolic logics. Variations on SD syntax or rules induce variations of the SD reasoning system, with some choices producing sound and complete systems. For example, the SD system in [48] is expressively equivalent to monadic first order logic with equality. SDs are also the basis of the richer language of Constraint Diagrams (CDs) [35], which has extra syntax to express explicit quantification and relations, and was proposed as a means of visually presenting invariants in object-oriented models, potentially as a visual replacement for the OCL.

The paper makes several contributions. We provide the first translation from SDs to Graph Transformation (GT) systems for the variant discussed in [32], that we adopt as a reference. The resulting system of typed graphs is called Spider Graphs (SGs) and we give constraints to characterise its language and prove its equivalence to the SD system. This formalisation provides a mechanisation of the reasoning system, based on general purpose GT tools, rather than ad-hoc algorithmic solutions. In [32] the inference rules are indeed specified algorithmically via sequences of actions that can be applied to a given diagram to infer a new one, and diagrams which are not syntactically correct can be produced at intermediate steps. The

proposed SG formalisation also offers the possibility to reason in formal terms about the transformations themselves. In particular, the invariants for the transformation steps can be expressed, characterising the language of all graphs which are part of a demonstration, and the relations between the different reasoning rules can be explored in terms of static analysis of collections of rules. We propose a mechanisation of the reasoning system based on the AGG system [37], which enables us to consider conflicts and dependencies between rules via critical pair analysis, to check the applicability of rule sequences on specific graphs, and in general to reason on proof strategies based on the characteristics of the source and target graphs.

To prove the equivalence between the SD and the SG systems, we also provide precise definitions in terms of a Z variant for the system of [32].

Paper organization. After the presentation of related work in Section 2, Section 3 provides an informal background on SDs and an abstract formalization of the SD reasoning system. A short introduction to concepts from GTs is given in Section 4. In Section 5, SGs are formally defined for the first time in terms of a type graph and a collection of positive and negative constraints, while Section 6 shows the realization of (part of) the unitary fragment of the reasoning system in [32] through SG transformation units and their construction from the original rules for SD; the whole construction is completed in Appendix A. A proof of the correctness of the proposed encoding of the reasoning system is given in Section 7, while Section 8 discusses techniques of analysis made possible by the proposed SG system, as well as some alternative approaches to formalisation in the GT area. Finally, Section 9 draws conclusions and points to future work.

2 Related work

To the best of our knowledge, this paper presents the first application of graph transformation techniques to the formalisation of the ED and SD logical reasoning systems. In this section we provide details of the most closely related works; see Section 3 for ED/SD terminology.

For CDs, a formal semantics was provided in [20], formal reasoning was investigated in [19], and a decidable, restricted, reasoning system was developed in [46]. Conceptual Graphs are a system of logic, based on the existential graphs of Charles Sanders Peirce and the semantic networks of artificial intelligence, intended as a readable, but formal design and specification language [45]. Hyperproof [1] is a heterogeneous reasoning system for understanding the information content of proofs rather than the syntactic structure of sentences; it provides access to both graphical and sentential information, together with a set of logical rules for integrating these different forms of information. Diagrammatic theorem proving methods have also been developed for mathematical problems, e.g. in the Diamond system [34]. A very recent development is Speedith, an interactive SD reasoner [52], which integrates an external theorem prover with reasoning rules for com-

pound SDs, allowing a user to observe the development of a proof, but without direct support for reasoning on proof strategies.

Directed acyclic graphs (DAGs) were used in [49] to represent Euler/Venn diagrams, preserving semantic and inferential properties. The DAGs are uniquely characterised by their leaf nodes, corresponding to ED zones; transformations of DAGs corresponding to the reasoning rules are used to check if, given a pair of diagrams, one could be inferred from the other. The focus in [49] was on checking proofs (or individual steps of proofs), as opposed to their construction, and graph rewriting techniques were not used.

There has been a significant amount of work on the automatic construction of concrete Euler diagrams from abstract Euler diagrams, primarily based on the fundamental works of Chow [8] and Flower et al. [22]. Adding syntax (such as spiders) to EDs adds further layout questions but does not fundamentally alter the complexity of the generation problem. In terms of interactive modelling tools one envisions facilities for the presentation and interaction with concrete diagrams, with user choices passed to the abstract level for use by the system, permitting the application of the machinery developed here (via AGG for example) followed by results presented at concrete level utilising generation tools. Therefore, following the standard approach from the logical perspective in this area, we deal here only with abstract syntax and the reasoning system acts at this level; this will also facilitate subsequent analysis on the effects of the choice of reasoning rules. Thus we effectively separate reasoning and generation concerns in order to deal with these issues independently. The investigation of the challenging question of concrete level transformations of EDs via transformation of the "dual graph" of the diagram was begun by Fish [18], although this avenue has yet to make full use of the power of the theory of graph transformations.

Tools to assist [24] and automate [25] reasoning with EDs and SDs use heuristic guidelines for searches. In [47], several ED reasoning systems were presented, using heuristics based on an A^* algorithm to provide a lower bound on the number of proof steps required, based on differences between the premise and the conclusion diagrams in a proof. The heuristics were constructed using difference measures (e.g. curve, zone and shading difference) providing a numerical estimate of the number of rules of a certain type to be applied in a proof, or, in the "restrictive system" to identify if a proof does not exist. In addition to these heuristic functions, two "proof-writing rules" were provided: the *add curve* rule is not applied (i) until all curves that need to be removed (i.e. those in the premise but not in the conclusion) have been removed; (ii) to add curves that are neither in the premise or the conclusion. In this case, the heuristics and proof-writing rules were discovered by inspection and analysis of the diagrammatic systems in question, requiring an in-depth knowledge of the individual systems themselves. Through our approach, we discover analogous heuristics in an automatic way, using formal tools for conflict analysis. Flower *et al.* [25] use less sophisticated heuristics and give an algorithm producing a proof within a sound and complete compound (i.e. enabling logical connectives between

diagrams) SD system if a premise diagram entails a conclusion diagram, or a counterexample (a model for the premise which is not a model for the conclusion) otherwise. In [24], a heuristic approach increases the readability of proofs for a unitary SD system, extended to a compound system in [23].

By mapping SDs to SGs, we transform the question of existence of a proof within the diagrammatic logic into that of reachability of one SG (corresponding to the conclusion SD) from another (for the premise SD). In particular, the properties of the diagrams determined by the heuristics of the restrictive system of [47] indicating the non-existence of a proof between a pairs of diagrams correspond to a characterisation of the non-reachability for the corresponding graph pair. Dependency analysis for the graph based framework presented in this paper may provide hints for similar "proof-writing rules" for the extended SD system.

In [4, 5], Transformation Units provided an operational semantics of OCL and the foundation for Visual OCL, mixing diagrammatic and textual notation. A translation from Visual OCL back to OCL enables round-trips between notations [16]. Graph transformations have been used for verifying properties of models in UML (see e.g. [42]), whereas a visual notation for expressing model to model transformations, according to the Query-View-Transform paradigm, has been incorporated into the QVT standard [39]. In the latter cases, the focus is on inter-model transformations, rather than the sort of intra-model transformations inherent to diagrammatic reasoning.

Several works employ constraints to define classes of graphs, or study classes of graph transformations preserving or enforcing such constraints. Taentzer *et al.* propose to manage inconsistencies among different model views via distributed graph rewriting, for example applying special rules when an inconsistency is identified [26]. The detection of inconsistencies between rules representing different model transformations has been attacked by static analysis methods in [30]. Similarly, Münch *et al.* add *repair actions* to rules in case some post-conditions are violated by rule application [38]. In all these cases, actions were modeled through single rules. Habel and Pennemann [28] have extensively treated the problem of transforming existing rules to make them compatible with (nested) constraints by adding application conditions. Their work unifies theories about application conditions [11] and nested graph conditions [41], lifting them to high-level transformations. In [40], a logic of graph constraints is defined to allow the use of constraints for language specification, and to provide rules for proving satisfaction of clausal forms. Finally, Ehrig *et al.* exploit layered graph grammars to derive a grammar generating (rather than transforming) instances of the language defined by a meta-model with multiplicities [15]. Satisfaction of OCL constraints is checked a posteriori on a generated instance.

An approach based on textual constraint programming has been used to specify forms of reasoning on diagrammatic languages, notably UML, together with OCL constraints (see e.g. [7] for a comparison of different tools). In this case, a mapping from a UML+OCL specification to a Constraint Satisfaction Problem has to be given. This model allows a general

encoding of UML models, on which arbitrary constraints have to be checked, whereas we are interested in coding a restricted number of reasoning rules on a language characterised by fewer constraints and types of elements.

3 Spider Diagrams

We describe the SD system in [32], whilst simplifying the terminology. We briefly describe the concrete syntax and the intuition of the semantics, before giving the formal abstract syntax definition. Then we present the transformation rules of the SD-system, called *reasoning rules*, that correspond to logical inferences. We present examples of SDs within a modelling context to demonstrate their use in specification and reasoning.

A *concrete* ED is a collection of labelled simple closed *curves* in the plane, decomposing it into connected minimal regions. A *zone* is a region inside one set of curves and outside all of the others; zones may be *shaded*. A *concrete* SD is an ED together with a set of *spiders*: trees whose vertices are placed in zones with no two vertices of the same tree lying in the same zone. Strands (wiggly lines) and ties (two parallel lines mimicking an *equals* sign) can be placed between the vertices of distinct spiders within a zone. We adopt the convention that all diagrams have a *boundary* curve, drawn as a rectangle and labelled by U (for the universe of discourse); this is in the set of inside curves for any region. In particular, there is an *outermost* zone, characterised by being inside only the boundary curve.

A concrete diagram represents a collection of logical statements according to the following intuitive semantics: a curve interior represents a set (corresponding to the label); intersection, union and complement on regions represent the corresponding operations on sets; spiders represent elements in the sets determined by their habitat (the set of zones that they have vertices in); if two spiders are connected by a tie within a zone and they represent elements in the set represented by that zone, then these elements are the same; if two spiders are connected by a strand within a zone, then they may be equal; the elements denoted by two distinct spiders are distinct if there is no zone for which both of the spiders lie in the same *strand-tie graph* (formed by taking all of the vertices of the spiders and all of the ties or strands within that zone); the set represented by a shaded zone contains no elements except for those denoted by spiders that inhabit that zone.

Figure 1 shows two examples of sequences of four SDs, demonstrating reasoning processes within the domain of nation composition, in particular for Great Britain. The diagram on the top left has 5 curves (including U), 5 zones (including the outermost zone), one spider (Bob) inhabiting a single zone ($\{U, GreatBritain, England\}, \{Wales, Scotland\}$). This diagram indicates that every member of Great Britain is either English, Welsh or Scottish (due to the shading) and that Bob is English. Deleting the *Wales* curve in the subsequent diagram also removes the shading: it is not true that the sets of English and Scottish people partition the set of people from

Great Britain. Two more curve deletions lead us to conclude that *Bob* is from Great Britain. In the bottom sequence *Alice* (a single spider that inhabits two zones) is either English or Scottish and the same sequence of curve deletions yields the conclusion that *Alice* is from Great Britain.

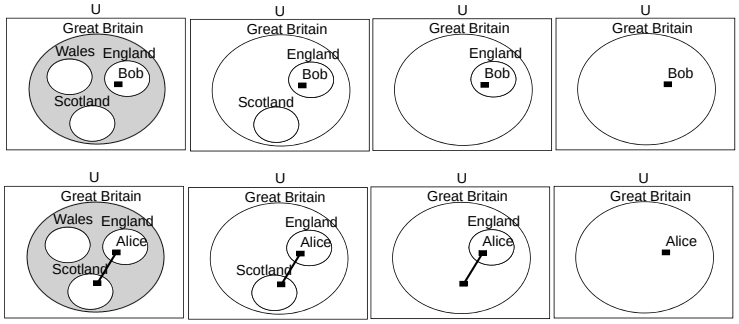


Fig. 1 Two sequences of SDs in which consecutive diagrams differ by erasing a curve (rule 5). We deduce that *Bob*, who is English, is from Great Britain, and *Alice*, who is either English or Scottish, is also from Great Britain.

3.1 Abstract syntax

The abstract syntax of an SD records the semantically important information, leaving out details such as the particular embeddings of the curve in the plane. We provide a formal definition of the abstract syntax of an SD in Definition 1, which is a more detailed formalisation of those in [32, 33], including the boundary curve, which is often omitted. We introduce only the necessary terminology and simplify notation, referring here to a spider diagram instead of a unitary spider diagram; one can also consider diagrams joined by logical connectives, but for simplicity, we only consider the unitary system consisting of single diagrams and their transformations.

In the following, the symbol $\setminus$ denotes set difference, $\mathcal{P}(X)$ denotes the powerset of a given set X, $\mathcal{P}^2(X)$ the set of subsets of cardinality 2 of a given set X, and $\mathcal{P}^f(X)$ the set of subsets of finite cardinality for an infinite set X. We write $\tau \cap \upsilon$ and $\tau \cup \upsilon$ to denote the intersection and union, respectively, of the two relations τ and υ, and $\Pi x, y \in Q$ for $\Pi x \in Q \wedge \Pi y \in Q$, for some quantifier Π and some set Q. The symbol ! indicates uniqueness.

Definition 1 (Spider Diagram.) *Let $\mathcal{L}$ be a fixed, countably infinite set of* ***labels****. A* ***spider diagram*** *is a tuple $d = \langle \mathcal{C}, \mathcal{Z}, \mathcal{Z}^*, \mathcal{S}, h, \tau, \upsilon \rangle$ where:*

1. *$\mathcal{C} = \mathcal{C}(d) \in \mathcal{P}^f(\mathcal{L})$ is a finite set of* ***curve labels****.*
2. *$\mathcal{Z} = \mathcal{Z}(d) \subseteq \{(X, \mathcal{C} \setminus X) \mid X \in \mathcal{P}^f(\mathcal{C})\}$ is a finite set of* ***zones****.*
3. *$\mathcal{Z}^* = \mathcal{Z}^*(d) \subseteq \mathcal{Z}$ is the set of* ***shaded*** *zones.*

4. $\mathcal{S} = \mathcal{S}(d) \in \mathcal{P}^f(\mathcal{L})$ *is a finite set of* ***spider labels****.*
5. $h : \mathcal{S} \to \mathcal{P}(\mathcal{Z})$ *is a function that returns the* ***habitat*** *of each spider (i.e. the set of zones that the spider inhabits).*
6. $\tau \subseteq \mathcal{P}^2(\mathcal{S}) \times \mathcal{Z}$ *is a relation between pairs of spiders and zones which indicates if two spiders are connected by a* ***tie*** *within that zone.*
7. $\upsilon \subseteq \mathcal{P}^2(\mathcal{S}) \times \mathcal{Z}$ *is a relation between pairs of spiders and zones which indicates if two spiders are connected by a* ***strand*** *within that zone.*

satisfying the following traits:

(a) $\exists! U \in \mathcal{C}(d)[(\exists! z_U = (\{U\}, \mathcal{C} \setminus \{U\}) \in \mathcal{Z}) \wedge (\forall z = (X, Y) \in \mathcal{Z}[U \in X])]$.
(b) $\forall c \in \mathcal{C}[\exists z = (X, Y) \in \mathcal{Z}[c \in X]]$.
(c) $\mathcal{C} \cap \mathcal{S} = \emptyset$.
(d) $\forall s \in \mathcal{S}[h(s) \neq \emptyset]$.
(e) $\tau \cap \upsilon = \emptyset$.
(f) $\forall(\{s_1, s_2\}, z) \in \tau \cup \upsilon[(z \in h(s_1) \cap h(s_2))]$.

The collection of all SDs is denoted $\mathcal{SD}$.

If $z = (X, Y)$ is a zone, then X is the set of labels for curves that *contain* z, Y is the set of labels for curves that *exclude* z, and $\{X, Y\}$ is a partition of $\mathcal{C}$. If $\{X, Y\}$ is a partition of $\mathcal{C}$ but $(X, Y) \notin \mathcal{Z}$ then we say that the zone (X, Y) is *missing*. We say that z is *inside* each $x \in X$ and *outside* each $y \in Y$. The label U is reserved; it is called the *universe*, and by condition a) it must be present. There is an *outermost zone* which is outside every curve except for U. Every curve must contain at least one zone by condition b).

Figure 2 shows a representation of the relations of the abstract syntax as a graph, where box nodes correspond to sets $\mathcal{L}$, $\mathcal{C}$ and $\mathcal{S}$ and bullet nodes are designators of relations, with the edges indicating the sets on which they are defined. The *zone* node defines two subsets of curves, whilst the *connection* node (strands or ties) defines a subset of spiders and of zones.

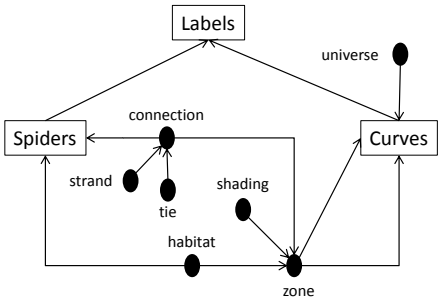


Fig. 2 A diagrammatic representation of the abstract syntax for SDs.

The formal semantics for an abstract SD are commonly provided in a model theoretic manner: a universal set **U** together with an interpretation

function mapping curves to subsets of **U** and spiders to elements of **U** are said to be a model for the diagram if they respect some additional *semantics predicate*. Examples of predicates are found in [32, 33]; we follow [32], which has been informally discussed above. Reasoning rules are transformations of abstract diagrams; they are said to be *sound* if each model for the premise diagram is also a model for the consequent diagram.

3.2 The transformation rules

We present the *SD* rules in the system of [32], rephrased so as to be presented as a clear natural language statement that is consistent with the improved terminology adopted here, whilst retaining the rule numbering to aid comparison. A complete reformulation into a formal Z-based specification is distributed between Section 7 and the Appendix. Besides providing a formalisation of this rule system (which was heretofore missing), this sets the ground for deriving graph transformation rules and transformation units which realise a faithful operationalisation of this system.

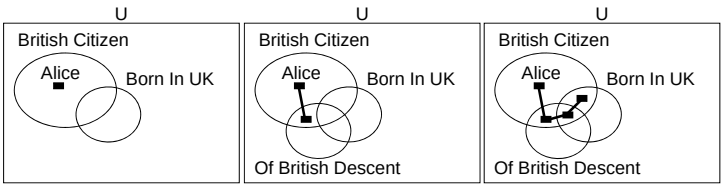


Fig. 3 A sequence of SDs, first introducing a curve (rule 6) and then extending a spider habitat (rule 2). The information about Alice, who is a British citizen, is weakened by removing the fact that she was not born in the UK.

We provide examples modeling the requirements of British Citizenship which demonstrate the application of the reasoning rules described below. Figure 3 shows an example where the curve labelled $OfBritishDescent$ is added to a model instance. The effect of introducing a curve (rule 6) replaces every present zone in the premise diagram (the left hand diagram) with two zones (these will be called twin zones with respect to the curve $OfBritishDescent$). The spider *Alice* inhabits both of the twin zones constructed from the zone inside $BritishCitizen$ and outside $BornInUK$. This essentially says that Alice is a British citizen who was born outside the UK, and so she may or may not be of British descent (i.e. have a British parent). The extension of the habitat of the spider *Alice* (rule 2) gives rise to the conclusion shown on the right hand side of the figure. This relaxes the constraint that *Alice* was not born in the UK, indicating that it is now not known if she was born in the UK or not (in a modelling context this might occur if the validity of the birth record documentation was doubtful).

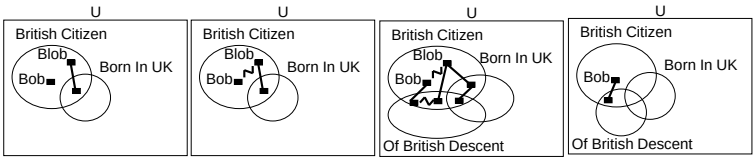


Fig. 4 A sequence of SDs showing the introduction of a strand (rule 1), followed by the introduction of a curve (rule 6) and the erasure of a spider (rule 3).

The left hand diagram in Figure 4 contains two spiders *Bob* (inhabiting one zone) and *Blob* (inhabiting two zones). Here, *Bob* and *Blob* are different British Citizens, and *Bob* was not born in the UK. The introduction of a strand (rule 1) between the *Bob* and *Blob* within the zone $(\{U, BritishCitizen\}, \{BornInUK\})$ removes the necessity that *Bob* and *Blob* are distinct (one might wonder whether partial records were duplicated with spelling errors). Curve introduction (rule 6) is shown next, followed finally by the erasure of the spider *Blob* (rule 3). It could have been decided that *Blob* was indeed the effect of an unfortunate typographical error.

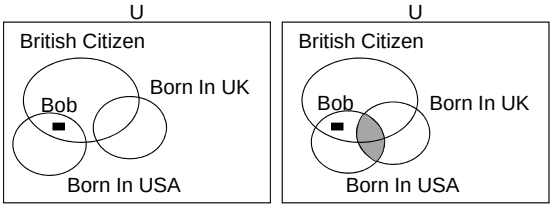


Fig. 5 A pair of SDs showing the application of addition of the reversible rule to add all of the missing zones (rule 7).

The SD reasoning rule, called Equivalence of Venn and Euler form, can be viewed as the addition of missing zones, and the removal of shaded zones which are not inhabited by any spider. Figure 5 shows that *Bob* denotes a British Citizen who was born in the USA. The curves showing *Born in UK* and *Born in USA* are disjoint (have disjoint interiors) and so the zone that would be inside both of these curve is missing (since no one can be born in both the UK and the USA). The application of rule 7 adds the missing zone as shaded zone in the conclusion diagram. This rule is reversible, and viewing the right hand diagram as the premise diagram, one can remove the shaded zones that have no spiders inhabiting them, returning the left hand diagram as conclusion. We present the set of rules in natural language, together with some definitions necessary to state the rules precisely.

Rule 1 (Introduction of a strand): If spiders s and t inhabit a zone z, but are not connected within z, then a strand connecting them within z can

be added. If spiders s and t inhabit a zone z and are connected by a tie within z then the tie can be replaced by a strand within z.

Rule 2 (Extend Habitat): If spider s does not inhabit zone z then the habitat of s can be extended to z, and if t is another spider inhabiting z then any connection (strand or tie) between s and t within z can be added.

For the statement of the next rule we introduce the notion of strand-tie graph, which will also be used in the formalisation via SGs.

Definition 2 (Strand-tie graph.) *Let d be a SD with zone z. Then the* strand-tie graph *of z is the graph whose vertices correspond exactly to the set of spiders that inhabit z, with edges connecting vertices if and only if those spiders are connected by a strand or tie within z.*

Rule 3 (Erase Spider): If spider s does not inhabit any zone that is shaded then s, together with all of its connections, can be erased. If this erasure disconnects the strand-tie graph of any zone z, then strands are added so that the strand-tie graph is connected again.

Rule 4 (Erase Shading): the shading can be erased from any shaded zone.

For the erase curve rule (used in Figure 1) we require the concept of twin zones from [21], formalised in the current style: zones z_1 and z_2 are *twins* with respect to a curve c when z_1 is inside c, z_2 is outside c and they have identical relationships (being inside or outside) with all other curves. This rule is a more precise version of the statement given in [32].

Definition 3 (Twin relation.) *With $z_1, z_2 \in \mathcal{Z}(d)$, $c \in \mathcal{C}(d)$, we have:* $twins_c(z_1, z_2) \Leftrightarrow \exists X, Y \in \mathcal{P}(\mathcal{C} \setminus \{c\})[X \cup Y \cup \{c\} = \mathcal{C} \wedge z_1 = (X \cup \{c\}, Y) \wedge z_2 = (X, Y \cup \{c\})]$.

Rule 5 (Erase Curve): a curve c can be erased. If either of $(X \cup \{c\}, Y)$ or $(X, Y \cup \{c\})$ is present then it is replaced by $z = (X, Y)$. If there exist zones z_1 and z_2 which are twins with respect to c then they are both replaced by z, and: (i) z is shaded *iff* both z_1 and z_2 were shaded; (ii) a spider s will inhabit z *iff* s inhabited at least one of z_1 and z_2; (iii) two spiders s and t are connected by a strand in z *iff* there was a connection (strand or tie) between s and t within either z_1 or z_2.

Rule 6 (Introduce Curve): a new curve c can be added. If $z = (X, Y)$ was present then z is replaced with the twins $z_1 = (X \cup \{c\}, Y)$ and $z_2 = (X, Y \cup \{c\})$, and: (i) z_1 and z_2 are both shaded *iff* z was shaded; (ii) a spider s inhabits z_1 and z_2 *iff* s inhabited z; (iii) two spiders s and t are connected by a strand (respectively, tie) within both z_1 and z_2 *iff* there was a strand (respectively, tie) between s and another spider t within z.

The following rule has an extra clause which was omitted in [32]: one cannot remove all of the zones inside any curve since this would break one of the constraints on the diagram; an alternative approach would be to adopt a cascading effect, removing any curves that would cause this problem.

Rule 7 (Equivalence of Venn and Euler forms): If Z_M is the set of missing zones, then Z_M may be added to $Z^*(d)$. If $K \subseteq Z^*(d)$, such that no spider inhabits any zone in K, and $U \notin K$, then K can be removed, provided that for each $c \in \mathcal{C}(d)$, there is at least one zone $z = (X \cup \{c\}, Y) \in \mathcal{Z}$ such that $z \notin K$.

4 Graph Rewriting

We set our study in the context of graph rewriting for typed graphs. A graph $G = (V, E, s, t)$ consists of a set[1] of *nodes* $V = V(G)$, a set of *edges* $E = E(G)$, a pair of *source* and *target* functions, $s, t : E \to V$. A (partial) *graph morphism* between two graphs $G_1 = (V_1, E_1, s_1, t_1)$ and $G_2 = (V_2, E_2, s_2, t_2)$ is a pair of (partial) functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ preserving source and target relations on their images, as expressed by Condition (1).

$$(e_2 = f_E(e_1)) \implies ((f_V(s_1(e_1)) = s_2(e_2)) \wedge (f_V(t_1(e_1)) = t_2(e_2)) \quad (1)$$

In a *type graph* $TG = (V_T, E_T, s^T, t^T)$, V_T and E_T are sets of node and edge types, while the functions $s^T : E_T \to V_T$ and $t^T : E_T \to V_T$ define source and target node types for each edge type. A typed graph on TG is a graph $G = (V, E, s, t)$ equipped with a (total) graph morphism $tp\colon G \to TG$, composed of functions $tp_V : V \to V_T$ and $tp_E\colon E \to E_T$, preserving the typing of the *source* and *target* functions, i.e. $tp_V(s(e)) = s^T(tp_E(e))$ and $tp_V(t(e)) = t^T(tp_E(e))$. We consider *simple* typed graphs, where at most one edge of a given type can exist for a given (*source*,*target*) pair [10].

A morphism between typed graphs (typed on the same TG) adds to condition (1) conditions (2) and (3) for type preservation.

$$(e_2 = f_E(e_1)) \implies (tp_E(e_2) = tp_E(e_1)) \quad (2)$$

$$(v_2 = f_V(v_1)) \implies (tp_V(v_2) = tp_V(v_1)) \quad (3)$$

In the rest of the paper we use only injective morphisms between graphs and omit the indication whether a morphism is a typing morphism or a morphism between typed graphs, when it is clear from the context.

An *atomic universal constraint* is a morphism $ac : P \to C$ [28]. A graph G *satisfies* ac, denoted $G \vDash ac$, if for each morphism $m : P \to G$ there exists a morphism $c : C \to G$ such that[2] $c \circ ac = m$. An atomic constraint is recursively defined to be either an atomic universal constraint or a construct $\neg ac$, with ac an atomic constraint. For satisfaction of $\neg ac$ one has: $G \vDash \neg ac \Leftrightarrow G \nvDash ac$. The set of *models* for ac is $M(ac) = \{G \mid G \vDash ac\}$. A constraint of the form $c : \emptyset \to C$ is called an *existential* constraint as it amounts to requiring the presence of a subgraph isomorphic to C within G.

[1] In this paper we consider only finite graphs, i.e. with V and E finite sets.

[2] The left argument of a composition $\circ$ is the last morphism to be applied.

The negation of an existential constraint ($\neg c : \emptyset \to C$) defines a *forbidden graph*, as it states that a subgraph isomorphic to C cannot appear in G.

General constraints can be built on top of atomic constraints via atomic formulae of the form $F = (F_1\ op\ F_2)$ where op is one of $\wedge$ or $\vee$ and F_1 and F_2 are atomic formulae or atomic constraints. Moreover, we also admit general nested constraints characterised by nested formulae of the form $N = OP(N_2)$ where OP is one of $\forall$ and $\exists$ and N_2 is either a nested formula or an atomic one. The notion of satisfaction for atomic constraints is suitably adapted to satisfaction of general nested constraints.

A graph rewriting rule is given by a morphism between typed graphs, together with application conditions expressed as (nested) formulae. We adopt here the Single-PushOut (SPO) approach to graph rewriting [14], where rules are expressed by presenting two graphs, called left- and right-hand sides (L and R) with a partial morphism $r\colon L \to R$ between them, stating which elements from L are preserved by the transformation. The left of Figure 6 shows an SPO direct derivation diagram, where the *target* graph H produced by the application of $L \xrightarrow{r} R$ to a *host* graph G is defined as the cospan $G \xrightarrow{g} H \xleftarrow{m^*} R$ for the span $G \xleftarrow{m} L \xrightarrow{r} R$, such that the resulting square (1) is a pushout (i.e. H is the union of G and R through their common pre-images in L). Informally, the application of a rule to a host graph G according to the *match* $m : L \to G$ deletes the elements in $m(L \setminus r^{-1}(R))$ and creates in the *target* graph H the elements in the image $m^*(R \setminus r(L))$. We indicate by $G \Longrightarrow_r^m H$ the relation between G and a target graph H obtained by applying rule r along the match m. The notation $G \Longrightarrow H$ defines the *derivation relation*, i.e. H can be produced by transforming G according to some rule along some match.

In the SPO approach, unlike the DPO approach [13], it is not required that the *dangling condition* be satisfied. Hence, a node in $L \setminus r(L)$ can be matched to a node in G even if its removal would leave some dangling edges. The pushout construction then ensures that in this case the application of the rule would remove the node together with all its adjoining edges. The right of Figure 6 shows that (atomic) constraints can be associated with a rule in the form of an *application condition* AC: $\{ac_i\colon L \to P_i, \{p_{ij}\colon P_i \to C_{ij}\}_{j\in J_i}\}_{i\in I}$, for a match $m\colon L \to G$ of the LHS of a rule. The rule is then applicable at the match m if the associated AC is satisfied by m, i.e. for each $n_i\colon P_i \to G$ for which $n_i \circ ac_i = m$, there exists $c_{ij}\colon C_{ij} \to G$ such that $c_{ij} \circ p_{ij} = n_i$. In a *negative application condition* (NAC) C_{ij} is empty, hence the NAC cannot be satisfied if n_i exists. Therefore, for r to be applicable, a subgraph isomorphic to P_i must not be present in G. In particular, NACs can be derived from a forbidden graph F according to the construction in [27], so as to avoid that the application of r creates a match for F.

In a similar way General Application Conditions (GACs) are expressed by formulae with the same structure as general nested constraints. A rule is applicable at a match m only if the associated formula is satisfied for m.

All these concepts are lifted to attributed typed graphs following the approach of [11]. Intuitively, we partition V into V_G and V_D (D for *domain*),

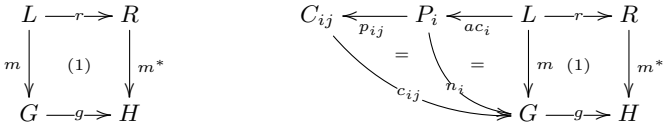


Fig. 6 SPO Direct Derivation Diagram for rules (left) and with AC (right).

the sets of *graph* and *value* nodes, respectively, while E is partitioned into E_G and E_A (A for *attribute*). *Graph edges* in E_G are equivalent to those for non-attributed graphs, while an *attribute edge* in E_A defines the assignment of a value to an attribute of a node. Moreover, we have $s = s_G \cup s_A$, with $s_G \colon E_G \to V_G$ and $s_A \colon E_A \to V_G$, and $t = t_G \cup t_A$, with $t_G \colon E_G \to V_G$ and $t_A \colon E_A \to V_D$. In a similar way, the type graph TG has distinct sets V_T^G and V_T^D of graph and value node types respectively, as well as distinct sets E_T^G and E_T^A for graph and attribute edge types. Given $t \in V_T^G$, all nodes of type t are associated with the same subset $A(t) \subset E_T^A$ of edge types, corresponding to the set of attribute names for t. Values in V_D range over the disjoint union of the set of sorts in a *data signature* $DSIG$. In this paper we consider that nodes have a single, immutable, attribute.

The final component of our setting are *Transformation Units* (TUs), by which one can express control conditions over rule application [36]. Let:

- $\mathcal{G}$ be the class of typed graphs;
- $\mathcal{R}$ be the class of SPO rules on typed graphs with application conditions;
- $\Longrightarrow$ be the derivation relation for the SPO approach;
- $\mathcal{E}$ be a class of graphs , where the *semantics* $sem(e)$ of an expression $e \in \mathcal{E}$ is a subclass of $\mathcal{G}$. In this paper we define $\mathcal{E}$ through the combination of a type graph and a set of graph constraints;
- $\mathcal{C}$ be a class of control conditions over identifiers of rules in $\mathcal{R}$ built on a grammar allowing identifiers, the sequential construct ';', the alternative choice '|', and the loop construct **asLongAsPossible** c **end**, with $c \in \mathcal{C}$.

A Transformation Unit is a construct $TU = (e_1, e_2, P, imp, c)$, with $e_1, e_2 \in \mathcal{E}$ initial and terminal graph class expressions (defining valid input and output graphs), $P \in \mathcal{P}^f(\mathcal{R})$ a finite set of SPO rules, imp a set of references to other, *imported*, TUs, whose rules can be used in the current one, and $c \in \mathcal{C}$ a control condition enabling rules from P and units from imp to be applied. A TU can only be applied to a graph $G \in sem(e_1)$ to produce a graph $H \in sem(e_2)$; if $H \notin sem(e_2)$, it is considered to fail.

TUs have a transactional behaviour, i.e. a unit succeeds only if its rules and imported units can be executed according to the control condition; it fails otherwise. In this case, all the graphs produced during its execution are discarded and the graph G is restored. For rules, failure corresponds to the absence of a match for L which satisfies the application condition. A sequence fails if any rule in the sequence fails. Control conditions of type **asLongAsPossible** r **end**, where r is a single rule, always succeed (i.e. even if r cannot be applied) and terminate as soon as no match is found for r.

However, if $r = r_1, \ldots, r_n$ is a sequence of rules, then the iteration on r fails if, after a successful execution of r_1 the execution of any rule r_i as prescribed by the remaining sequence $r_2, \ldots, r_n$ fails. Otherwise, **asLongAsPossible** r **end** terminates with success as soon as no further match is found for r_1.

In this paper, we take $e_1 = e_2$ to specify the class of Spider Graphs, as defined in Section 5 and P to be the set of rules in Section 6. We relate rule expressions to graph rules by naming rules and passing parameters to them. We use parameters to identify nodes, thus restricting rule application to specific diagram elements with specific properties. Hence, the rules presented in the TUs in Section 6 are actually rule schemata to be instantiated to actual rules, with the parameter values defining application conditions.

5 Spider Graphs

We introduce *Spider Graphs* (SGs) as a language of typed graphs, specified by a type graph and a system of constraints, providing an equivalent representation for the abstract syntax of SDs presented in Section 3.

The definition of the type graph for SGs starts from the diagrammatic representation of the abstract syntax of SDs in Figure 2, where nodes designating subsets can be construed as hyperedges. In order to use simple typed graphs, we employ a construction analogous to König's one for reducing hypergraphs to bipartite graphs (see e.g. [54]). That is, a hyperedge type can be reified into a node type together with a corresponding edge type for every hyperedge tentacle directed towards a different type of node. The construction starts by recognising curves and spiders as primary types of nodes, to which the types `Curve` and `Spider` correspond. With these two types, we associate the attribute `label`, with values of type `String`, to provide a reference to the original label for the corresponding element. After that, zones are identified as hyperedges connecting the curves including them. Hence, we define the node type `Zone` and the edge type *inside*, relating zones to curves. We dispense with the representation of the outside relation as it can be derived from the inside one. For the habitat relation, the same procedure would give a `Habitat` node type with edge types *spider* and *zone* to connect it to the corresponding node types. However, each habitat node would have an edge toward a single spider and a collection of edges to a set of zones. Since we do not need to identify specific instances of habitat, we replace this type by directly connecting spiders and zones via the *inhabit* edge type from `Spider` to `Zone`. Finally, connections are hyperedges relating spiders inhabiting the same zone, so we provide a `Connection` node type with edges of type *connects*, relating connections to spiders, and of type *within*, relating connections to the zone for which the relation exists. The remaining designators in Figure 2 correspond to unary predicates on single elements of type `Zone` or `Connection`. Hence, we represent them as self-edge types, where the type *shading* indicates that the corresponding zone is shaded; the presence of a *universe* self-edge denotes the boundary curve, and self-edges of type

tie or *strand* define the nature of the connection. The resulting type graph **SG** is shown in Figure 7, where multiplicities limit the number of edges of the same type leaving from or entering a given node type (self-edges are always restricted to have multiplicity 0 or 1).

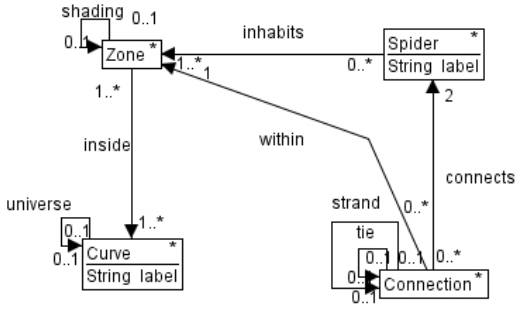


Fig. 7 The type graph for SGs.

Under this typing, Definition 4 introduces a notion of representation of an SD as a graph typed on **SG**. For G a graph typed on **SG**, we call T-node any node n of G of type T and T-edge any edge of G of type T. We write n_T to indicate that n is a T-node, and e_T to indicate that e is a T-edge.

Definition 4 (Representation of SDs.) *Let $d = \langle \mathcal{C}, \mathcal{Z}, \mathcal{Z}^*, \mathcal{S}, h, \tau, \upsilon \rangle \in \mathcal{SD}$ and let $G = (V, E, s, t)$ be a simple typed graph on **SG**. Then we say that G is a* representation *of d if and only if:*

1. *there is a bijection b_c from $\mathcal{C}$ to the set of* Curve*-nodes.*
2. *there are exactly one* Curve*-node n_U and one* universe*-edge e_U such that $s(e_U) = t(e_U) = n_U$.*
3. *there is a bijection b_z from $\mathcal{Z}$ to the set of* Zone*-nodes, such that: $z = (X, Y) \in \mathcal{Z} \iff$ there is a partition of the set of* Zone*-nodes of G into X and Y (permitting X or Y to be empty) such that:*
   (a) *there is an* inside*-edge from $b_z(z)$ to $b_c(x)$, for every $x \in X$, and*
   (b) *there is not an* inside*-edge from $b_z(z)$ to $b_c(y)$, for any $y \in Y$.*
4. *$z \in \mathcal{Z}^* \iff$ there is a* shading*-edge from $b_z(z)$ to itself.*
5. *there is a bijection b_s from $\mathcal{S}$ to the set of* Spider*-nodes, such that: $z \in h(s)$ with $s \in \mathcal{S}$ and $z \in \mathcal{Z} \iff$ there is an* inhabit*-edge from $b_s(s)$ to $b_z(z)$.*
6. *there is a bijection b_{con} from the set of all connections $\tau \cup \upsilon$ to the set of* Connection*-nodes, such that:*
   (a) *$t = (\{s_1, s_2\}, z) \in \tau \cup \upsilon \iff$ there is a* within*-edge from $b_{con}(t)$ to $b_z(z)$ and there are* connects*-edges from $b_{con}(t)$ to $b_s(s_1)$ and $b_s(s_2)$, both of which have* inhabits*-edges to $b_z(z)$.*
   (b) *$t = (\{s_1, s_2\}, z) \in \tau \iff$ there is a* tie*-edge from $b_{con}(t)$ to itself.*
   (c) *$t = (\{s_1, s_2\}, z) \in \upsilon \iff$ there is a* strand*-edge from $b_{con}(t)$ to itself.*

Lemma 1 *Let G_1 and G_2 be representations of the same Spider Diagram d. Then G_1 is isomorphic to G_2.*

Proof We observe that the conditions in Definition 4 fix all choices for graphs typed over the type graph in Figure 7. Any representation G of d has a fixed number of nodes of each type, determined by the bijections in conditions 1, 3, 5, and 6 of Definition 4. The self-edges are completely determined by conditions 2, 4 and 6(b,c), noting that condition 2 ensures that there is a universe self-edge present in G corresponding to the boundary curve in d even though this is not enforced by the type graph for SG. Conditions 3 and 5 fix the *inside* and *inhabits* relationships. Finally, condition 6(a) fixes the choices for the *connects-inhabits-within* relationship triangle of the type graph: the bijection from the connections of d to the `Connection` nodes of G with the required relationships, together with the cardinality constraint on the *within* and *connects* relationship, means that there can be no extra edges besides those involved in the imposed relationship. □

Since the graph representation of an SD d is unique up to isomorphism, we can refer to it as *the* representation of d; the existence of one such graph is guaranteed. Theorem 1 characterises the properties of graphs representing SDs enabling the identification of a set of constraints completing the definition of the class of Spider Graphs.

Theorem 1 *Let $d = \langle \mathcal{C}, \mathcal{Z}, \mathcal{Z}^*, \mathcal{S}, h, \tau, \upsilon \rangle \in \mathcal{SD}$ and let G be the representation of d. Then the following properties hold for G, where edge types are omitted if they are derivable from the context (i.e. from the type graph).*

1. $\forall e_1, e_2 \in E[s(e_1) = s(e_2) \wedge t(e_1) = t(e_2) \implies e_1 = e_2]$
2. $\forall n_1, n_2 \in V[(n_1 \neq n_2 \wedge tp_V(n_1) = tp_V(n_2) = \texttt{Zone}) \implies$
   $(\exists n_c \in V[(tp_V(n_c) = \texttt{Curve}) \wedge$
   $(((\exists e_1 \in E[s(e_1) = n_1 \wedge t(e_1) = n_c]) \wedge (\nexists e_2[s(e_2) = n_2 \wedge t(e_2) = n_c])) \vee$
   $((\exists e_2 \in E[s(e_2) = n_2 \wedge t(e_2) = n_c]) \wedge (\nexists e_1[s(e_1) = n_1 \wedge t(e_1) = n_c])))])]$
3. $\exists! n_U \in V[(tp_V(n_U) = \texttt{Curve}) \wedge$
   $(\forall n_z \in V[(tp_V(n_z) = \texttt{Zone}) \implies (\exists e_U \in E[(s(e_U) = n_z \wedge t(e_U) = n_U)])]) \wedge$
   $(\exists n_{zU} \in V[(tp_V(n_{zU}) = \texttt{Zone}) \wedge (\forall n_c \in V[(n_c \neq n_U \wedge tp_V(n_c) = \texttt{Curve}) \implies (\nexists e_n \in E[s(e_n) = n_{zU} \wedge t(e_n) = n_c])])])]$
4. $\forall n_c \in V[tp_V(n_c) = \texttt{Curve} \implies \exists n_z \in V, e_n \in E[s(e_n) = n_z \wedge t(e_n) = n_c]]$
5. $\forall n_s \in V[tp_V(n_s) = \texttt{Spider} \implies \exists n_z \in V, e_s \in E[s(e_s) = n_s, t(e_s) = n_z]]$
6. $\forall n_x \in V[tp_V(n_x) = \texttt{Connection} \implies \exists! e_x \in E[(s(e_x) = t(e_x) = n_x) \wedge (tp_E(e_x) = \text{strand} \vee tp_E(e_x) = \text{tie})]]$
7. $\forall n_x \in V[(tp_V(n_x) = \texttt{Connection}) \implies (\exists! n_1, n_2, n_z \in V[(n_1 \neq n_2) \wedge (tp_V(n_1) = tp_V(n_2) = \texttt{Spider} \wedge tp_V(n_z) = \texttt{Zone}) \wedge$
   $(\exists e_1, e_2 \in E[tp_E(e_1) = tp_E(e_2) = \text{connects} \wedge t(e_1) = n_1 \wedge t(e_2) = n_2 \wedge s(e_1) = s(e_2) = n_x]) \wedge$

$$(\exists e_3, e_4 \in E[(tp_E(e_3) = tp_E(e_4) = \text{inhabits}) \wedge (t(e_3) = t(e_4) = n_z) \\ \wedge (s(e_3) = n_1) \wedge (s(e_4) = n_2)]) \wedge \\ (\exists e_x \in E[tp_E(e_x) = \text{within} \wedge s(e_x) = n_x \wedge t(e_x) = n_z])])]$$

Proof We analyse each of the *properties* in Theorem 1 using the *conditions* and *traits* on d from Definition 1. These translate to conditions on the representation G of d due to the bijections in Definition 4, ensuring that there is a unique node in G for each curve, spider and zone in d, as required by conditions (1), (2), (4) and trait (c) of Definition 1. Property (1) of Theorem 1 says that there is at most one edge between any two nodes of G. Since G is a simple typed graph by Definition 4, non-unique edges between a pair of nodes can only occur as a *tie*-edge and a *strand*-edge on the same `Connection` node; this is prevented by bijection b_{con}. Property (2) says that any two distinct `Zone` nodes have distinct sets of *inside*-edges; this follows from condition (2) of Definition 1 and bijection b_z. Property (3) says that there is a unique `Curve`-node for which every `Zone`-node is *inside*, and there is a `Zone`-node which is *inside* this particular `Curve`-node only; this follows from trait (a) in Definition 1 and bijections b_c and b_z. Property (4) says that each `Curve`-node has a `Zone`-node which is *inside* it; this follows from trait (c) in Definition 1 and bijections b_c and b_z. Property (5) says that each `Spider`-node *inhabits* at least one `Zone`-node; this follows from condition (5) and trait (d) in Definition 1, and bijections b_s and b_z. Property (6) says that each `Connection`-node has exactly one self-edge, of type either *strand* or *tie*, trait (e) in Definition 1, and bijections b_{con} and b_s. Property (7) says that each `Connection`-node *connects* exactly two `Spider`-nodes *within* the `Zone`-node that both spiders *inhabit*; this follows from conditions (6), (7) and trait (f) in Definition 1, and bijections b_{con}, b_s and b_z. □

We observe that the only condition of Definition 1 that was not considered in the above proof was condition (3); the bijection between shading in zones and shading self-edges was already imposed in Definition 4.

5.1 Graph constraints for SGs.

We express the properties derived from Theorem 1 as graph constraints, characterising the class of SGs independently from their being a representation of an SD. Hence, the forbidden graphs shown in Figure 8 enforce uniqueness of connections and of the *universe* self-edge and ensure that each occurrence of *strand*, *tie*, *shading*, *universe* is a self-edge, while $F8$ prevents a connection from being both a strand and a tie. The forbidden graph $F2$, ensuring uniqueness of the *inhabits* edges, is representative of the class $\mathcal{F}$ of constraints enforcing simplicity of typed graphs.

The positive constraints in Figure 9 correspond to other properties of Theorem 1: property 7 about connections existing only between spiders which both inhabit the region that the connection is within ($P1$); property 5 about each spider inhabiting at least one zone ($P2$); property 4 about not

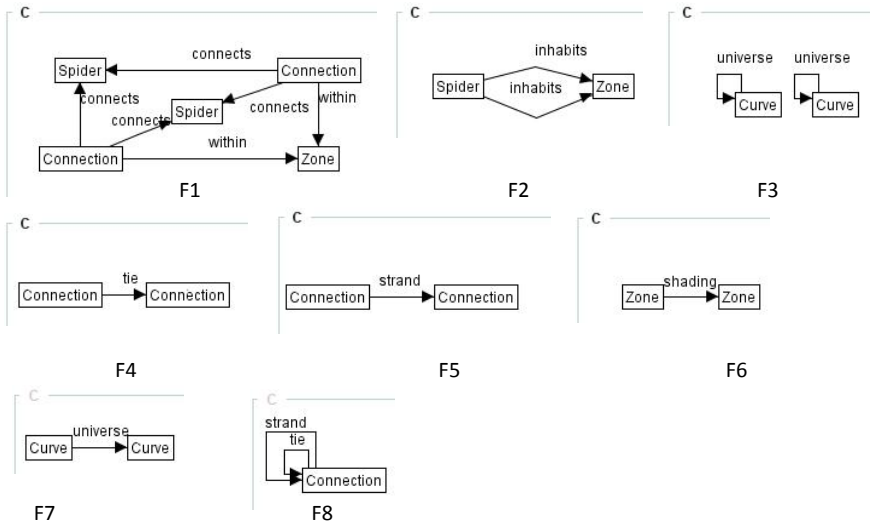


Fig. 8 Forbidden graphs for uniqueness constraints.

having a curve without zones inside ($P3$); and the first part of property 3 about each zone being inside the boundary curve ($P4$) (marked with the *universe* edge). These are all universal constraints of the form $P \to C$, while constraint $P5$ in Figure 10, requiring the existence of the boundary curve, is an existential constraint of the form $\emptyset \to C$. The nested constraint $P6$ at the bottom of Figure 10 requires the existence of an outermost zone lying inside of the boundary curve only. Here and henceforth, identical numbers in different components of a constraint or of a rule identify corresponding elements in the morphisms. The constraint $P7$ in Figure 11 realises property 6: each connection is either a tie or a strand. Finally, Figure 12 presents a nested constraint ($P8$) stating that for each pair of zones there is at least one curve such that the two zones have different relations with that curve, while the constraints $P9$-$P11$ on values in Figure 13 ensure uniqueness of the node associated with a label.

Definition 5 gives the formal characterisation of the language of Spider Graphs ($\mathcal{SG}$), while Theorem 2 proves that elements of $\mathcal{SG}$ are in a bijective correspondence with elements of $\mathcal{SD}$.

Definition 5 (Spider Graph.) *A simple typed graph G on $\boldsymbol{SG}$, is a* Spider Graph *if and only if $G \not\models F_i$, for $i \in \{1, \ldots, 8\}$, $G \not\models f$, for $f \in \mathcal{F}$[3] and $G \models P_i$, for $i \in \{1, \ldots, 11\}$. Let $\mathcal{SG}$ denote the class of all SGs.*

Theorem 2 *Let $G \in \mathcal{SG}$ be a Spider Graph. Then there exists $d \in \mathcal{SD}$ such that G is a representation of d.*

[3] $\mathcal{F}$ is the set of forbidden graphs enforcing simplicity of the typed graphs.

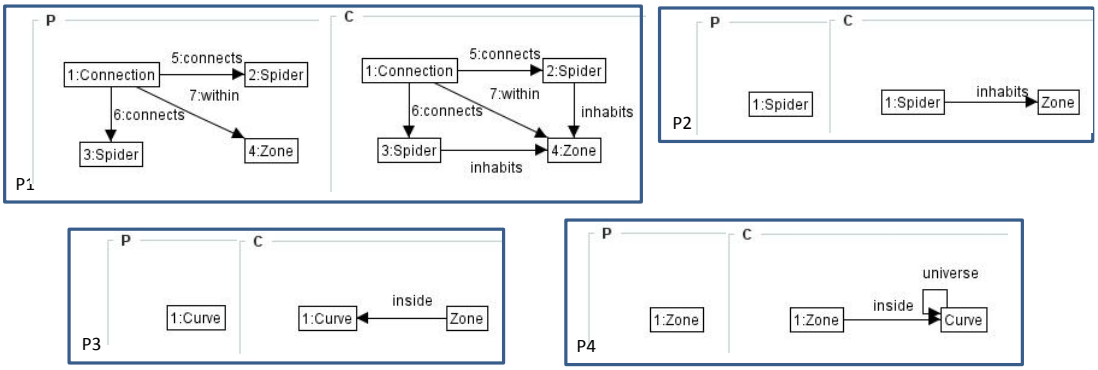


Fig. 9 Positive constraints for SGs.

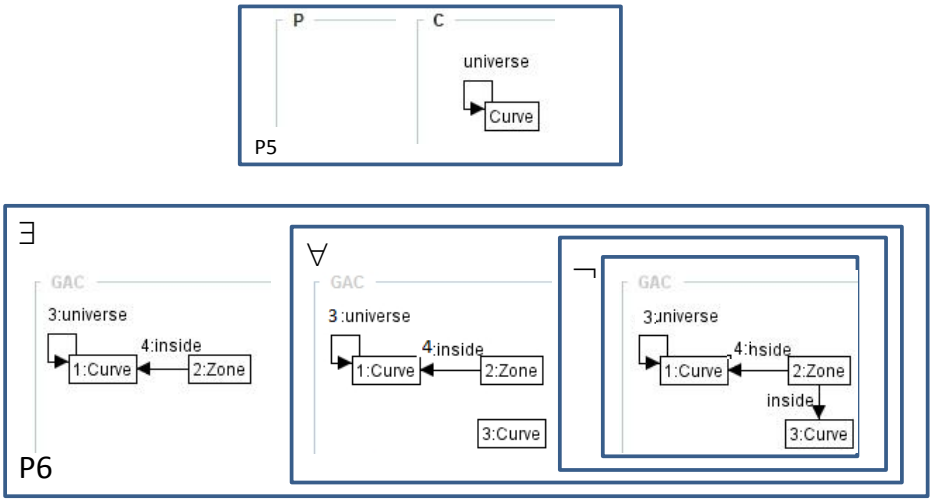


Fig. 10 There exists a boundary curve and an outermost zone.

Proof We construct $d = \langle \mathcal{C}, \mathcal{Z}, \mathcal{Z}^*, \mathcal{S}, h, \tau, \upsilon \rangle \in \mathcal{SD}$ such that there exist bijections as in Definition 4, indicating the *conditions* and traits of Definition 1 that must hold due to the construction together with the constraints defining the SG language. This ensures that the construction yields a SG. We construct d from the nodes of G and then the edges for curves, spiders and connections, pointing to the relevant conditions from Definition 1 satisfied at each stage.

For each node of G of type `Curve`, `Zone` and `Spider`, construct exactly one $c \in \mathcal{C}$, $z \in \mathcal{Z}$ and $s \in \mathcal{S}$ respectively. For each $z \in \mathcal{Z}$, let $z = (X, Y)$, where X is the set of curves constructed for the `Curve`-nodes that the `Zone` node was *inside* (i.e. it is adjacent to, via an edge typed as *inside*), and Y is the remaining set of curves. No two `Zone`-nodes in G have the same set

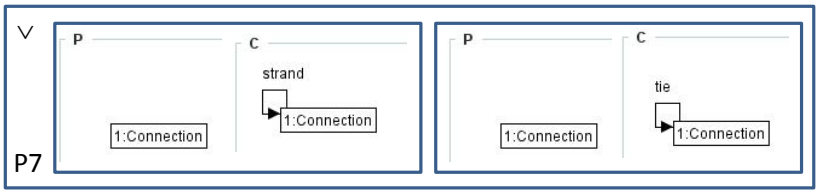


Fig. 11 Connections are either ties or strands.

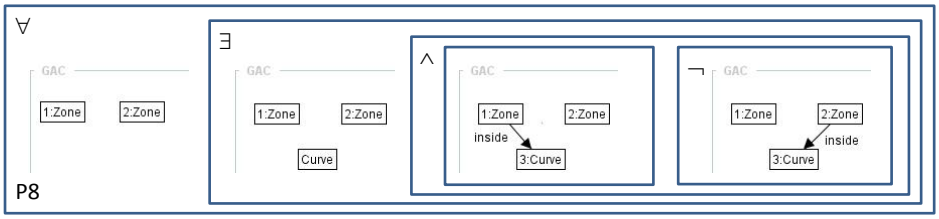


Fig. 12 Two distinct zones have different *inside* relationships with some curve.

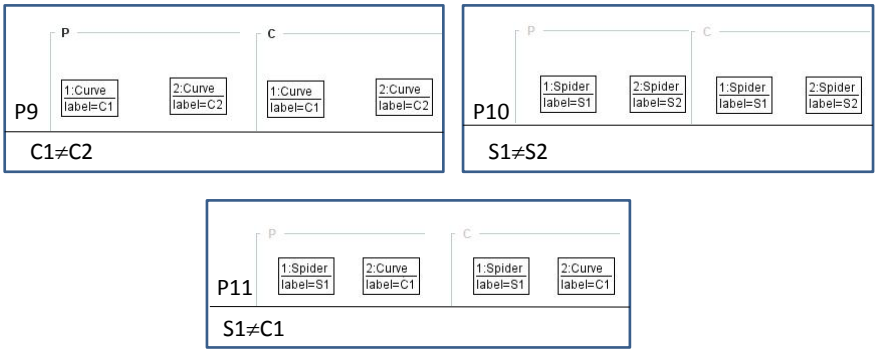


Fig. 13 Labels are unique for spiders and curves.

of adjacent `Curve`-nodes as per the constraint in Figure 12. For each $z \in \mathcal{Z}$, take $z \in \mathcal{Z}^*$ if and only if the corresponding `Zone`-node had a *shading* self-edge; there are no *shading* edges between distinct zones by $F6$. Thus, conditions 1-4, and trait (c) of Definition 1 hold.

Exactly one `Curve`-node has a universe self-edge because of $F3$ and $P5$; no *universe* edge is adjacent to two distinct `Curve`-nodes due to $F7$. Every zone is *inside* the boundary curve due to $P4$, and there is a zone that is only inside the boundary curve due to $P6$. So, trait (a) of Definition 1 holds. Every curve has a zone *inside* it, due to $P3$. So trait (b) of Definition 1 holds.

For each spider $s \in \mathcal{S}$, the habitat function is constructed by associating with each spider s the set of zones corresponding to the set of `Zone`-nodes adjacent (via an *inhabits* edge) to the `Spider`-node corresponding to s. This is non-empty due to $P2$. Thus condition 5 and trait (d) of Definition 1 hold.

For each `Connection`-node n of G, if n has a *strand* or *tie* self-edge then construct $(\{s,t\},z) \in \upsilon$ or $(\{s,t\},z) \in \tau$, respectively, where $s,t \in \mathcal{S}$ are the spiders defined for the *Spider*-nodes which have a *connects* edge from n, and $z \in \mathcal{Z}$ is the zone defined for the *Zone* node with a *within* edge from n; s,t and z exist due to the type graph constraints. The incident edges with any `Connection` node must be self-edges due to $F4$, $F5$ and the type graph constraints. Every `Connection` node has either a *strand* or a *tie* self edge by $P7$ and $F8$. Thus conditions 6 and 7 of Definition 1 hold. We must have that s and t inhabit z, due to $P1$. So trait (f) holds. The sets υ and τ are disjoint due to $F1$, $F8$ and $P7$. So trait (e) holds.

Since all of the conditions and traits of Definition 1 are satisfied, d is a Spider Diagram. Since the construction ensures that the bijections of Definition 4 exist, G is a representation for d, as required. □

6 A Graph Transformation System for Spider Graphs

We define a collection $\mathcal{TU}_{SG}$ of TUs realizing the reasoning system in Section 3. We first outline the approach followed and the conventions adopted. Figure 14 shows the two SGs corresponding to the first two SDs in the sequence of Figure 3, illustrating the effect of the TU corresponding to the *Introduce curve* rule.

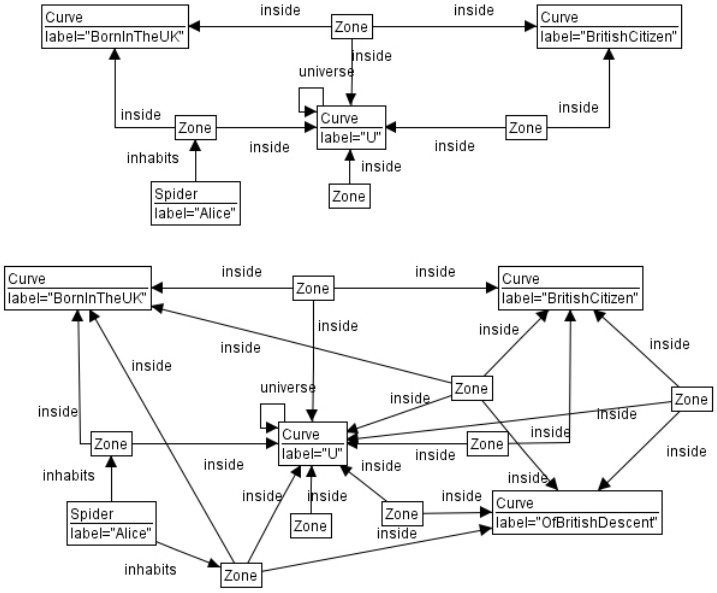


Fig. 14 SG representation of the first reasoning step of Figure 3.

6.1 A procedure for deriving transformation rules from SD rules

For each SD rule, we define the corresponding TU in the SG system and the individual GT rules in the unit. In general, for each SD rule, the derived TU in $\mathcal{TU}_{SG}$ presents one *principal GT rule*. This is the last GT rule to be applied in a TU derived from a SD rule erasing some element, or the first one in a TU from a SD rule introducing some element. The remaining GT rules in the TU derive from two requirements. First, rules are needed to ensure that the TU terminates producing a syntactically correct SG, i.e. complying with the type graph and the graph constraints. Then, rules are needed to preserve semantic properties of the transformed models. Such rules might vary if different semantics were involved. By breaking down the global effect of a SD reasoning rule into GT rules in a TU, the preconditions for each GT rule can be ensured by the effects of previously applied GT rules.

In order to operate on curves or spiders with specific labels, we use parameters which allow the selection of the correct element. Auxiliary markers are used in some rules to force looping on all the required elements, or to reify some complex relation between elements involved in a transformation.

The application of a principal rule may require some preparation (for erasing rules) or repair action (for introducing rules). In general these actions involve arbitrary contexts, so they have to be applied in situations which might lead to inconsistent graphs. Context can be managed by parallel application on all non conflicting matches, or by iterating rule application on all of them, via the `asLongAsPossible` construct. For this presentation, we adopt this latter option for all cases. The order in which these actions have to be taken depends on the specific inconsistencies which may arise. For individual rules, if the absence of an element is stated in the pre-condition, then this is included in a NAC. An additional set of NACs is used to prevent iteration on the same elements. In general, we explicitly show the NAC as part of the pre-condition only where necessary. The negative constraints expressed by the forbidden graphs of Figure 8 are used to generate NACs preventing the application of a rule which would produce such a graph [27].

6.2 The transformation rules

We use a reformulation of the SD rules in terms of pre- and post-conditions to derive graph transformation rules and TUs complying with these conditions. We present only the rules for erasing and introducing a curve, which are the most complex ones, and complete the presentation of the SD system in Appendix A. In these TUs, some rules require the use of additional edges, which appear only temporarily while executing a TU and are removed when it terminates. In particular, spiders and zones can be *mark*ed in the context of an iteration on all instances of a type, while a *twin* edge type is used to relate zones in the $twin_c$ relation for some curve c.

Rule 5. ***[Erasure of a curve.]*** Given a curve c, let S_c be the set of zones which are twins, w.r.t. c, of zones inside c. Deleting c entails an update of: 1) the relations of spiders with S_c; 2) the connections in zones of S_c; 3) their shading properties. Once these operations are completed, all the zones inside c are removed, before deleting c itself in the principal rule for the TU. In the resulting control condition, `C` identifies the curve to be erased.

EraseCurve(`C`) =
asLongAsPossible `associateTwin(C)` **end**;
asLongAsPossible `extendInhabitRelation` **end**;
asLongAsPossible `copyStrandToSurvivorZone` **end**;
asLongAsPossible `ensureConnectionsAreStrandInSurvivorZone` **end**;
asLongAsPossible `removeConnectionFromCondemnedZone` **end**;
asLongAsPossible `ensureCoherentShadingInSurvivorZone` **end**;
asLongAsPossible `removeCondemnedZone(C)` **end**;
(`deleteCurveWithTwin(C)` | `deleteCurveWithoutTwin(C)`)

The rules in the TU are presented in Figures 15-24. The first rule, in Figure 15, is iterated to identify zones which are twins with respect to the curve c specified by the parameter `C` (i.e. with label equal to `C`). An edge of type *twin* is created for each such pair, with source in the *condemned* zone inside c (to be eliminated by the TU) and target in the *survivor* zone outside c (preserved by the TU). The rule presents three NACs, respectively ensuring that: 1) the boundary curve cannot be deleted; 2) the zone node that will become the target of the twin edge is not inside c; 3) the twin relation has not already been established between the same pair of zones.

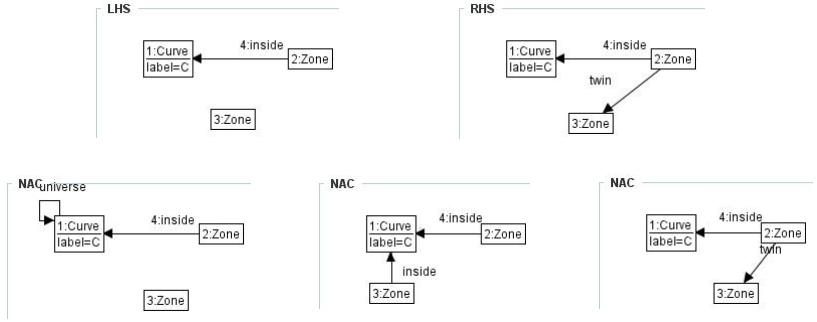


Fig. 15 Rule `associateTwin(C)`: identifying twins.

For conciseness, we define a predicate $inside(z, c) \iff \exists e \in E[tp_E(e) = inside \wedge s(e) = z \wedge t(e) = c]$. The GAC in Figure 16 assesses the presence of a twin relation between two zones z_1, z_2 (identified by a match for the LHS of `associateTwin(C)`) such that $inside(z_1, c)$ and $\neg inside(z_2, c)$ (from the rule NAC). More precisely, it checks the property $\forall c_1 \neq c[(inside(z_1, c_1) \wedge inside(z_2, c_1)) \vee (\neg inside(z_1, c_1) \wedge \neg inside(z_2, c_1))]$, i.e. the zones have different relations with c but the same relations with all of the other curves.

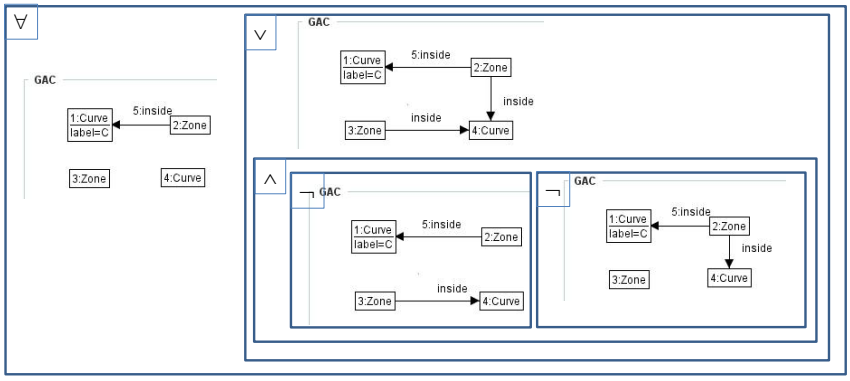


Fig. 16 A graphical representation of the GAC to identify twins, to be checked on each match m for the LHS of the rule `associateTwin(C)` in Figure 15. The formula reads "for each match $m_1 : g_1 \to G$ of the graph g_1 in the $\forall$-box extending m, either the graph in the $\vee$-box has a match extending m_1 or neither of the graphs in the $\wedge$-box has a match extending m_1".

The rules in Figures 17-20 are individually iterated to transfer to survivor twins the information about spiders and connections within condemned zones. The rule `extendInhabitRelation` in Figure 17 extends the *inhabit* relation for each spider in a condemned zone to its survivor twin (a NAC, not shown here, checks that the spider does not already inhabit the survivor).

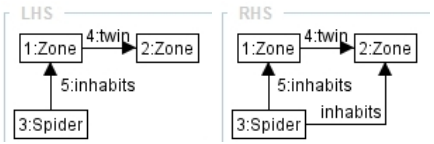


Fig. 17 Rule `extendInhabitRelation`.

The rule `copyStrandToSurvivorZone` (see Figure 18) creates a strand in the survivor zone for each existing connection (of any type) in the condemned zone. A NAC (not shown here) in this rule prevents application if a connection already exists between the same spiders within the survivor zone. To ensure that any connection in the survivor zone is a strand, any previously existing tie within it is converted by looping on the rule `ensureConnectionIsStrandInSurvivorZone` in Figure 19.

The connections in a condemned zone can now be removed by looping on rule `removeConnectionFromCondemnedZone` in Figure 20.

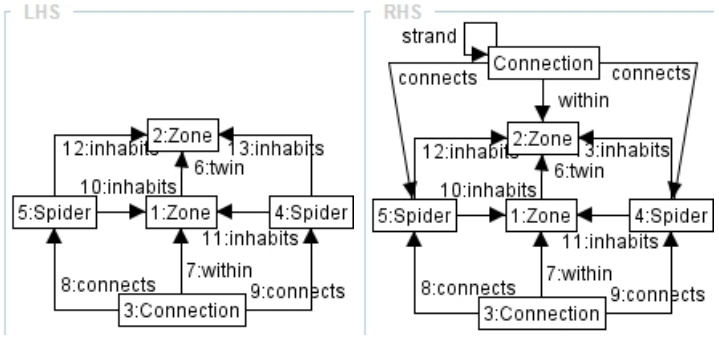


Fig. 18 Rule `copyStrandToSurvivorZone`.

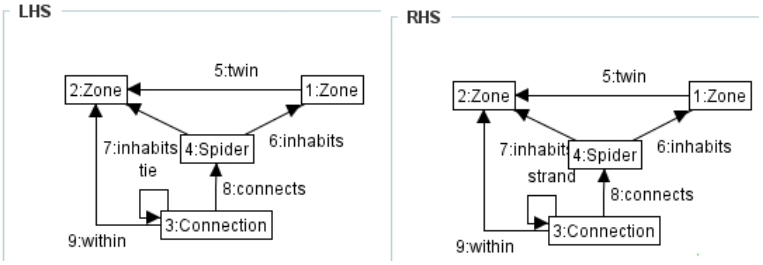


Fig. 19 Rule `ensureConnectionIsStrandInSurvivorZone`.

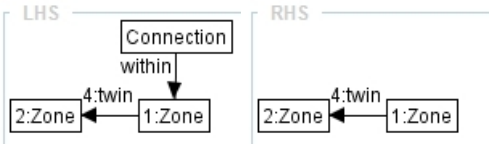


Fig. 20 Rule `removeConnectionFromCondemnedZone`.

The condition (i) from the specification of Rule 5 in Section 3 is enforced by `ensureCoherentShadingInSurvivorZone` (Figure 21), which removes the shading from the survivor zone if the condemned zone was not shaded.

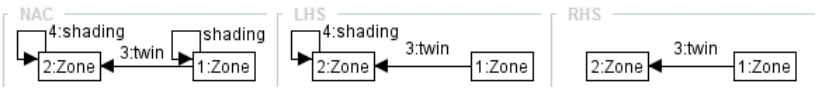


Fig. 21 Rule `ensureCoherentShadingInSurvivorZone`.

The iteration of `removeCondemnedZone` in Figure 22 and a final alternative between `deleteCurveWithTwin` or `deleteCurveWithoutTwin` complete the unit. The iteration removes all condemned zones, identified by having a twin, up to the last one. The first alternative (in Figure 23) removes the curve together with the last twinned zone inside it. If no zone had a twin, then the second alternative is used (Figure 24). The NAC for the second rule ensures that only one of the rules can be applied, while both NACs are consistent with the rules being applied only at the very end of the process. Together, these rules preserve constraint $P3$, so that while the curve is present, a zone is inside it. Note that, as we adopt the SPO approach, all the remaining *inhabits* edges are deleted together with the curve.

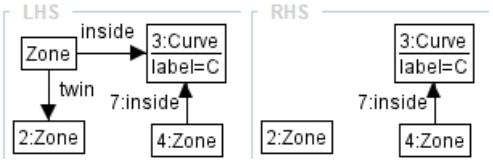


Fig. 22 Rule `removeCondemnedZone`.

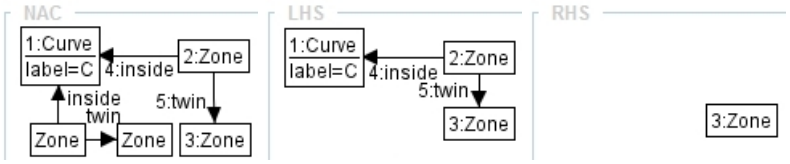


Fig. 23 Rule `deleteCurveWithTwin`.

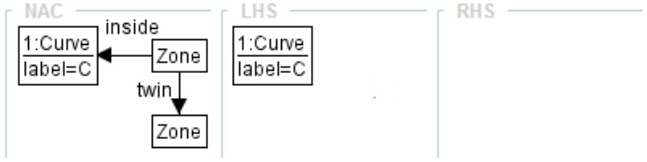


Fig. 24 Rule `deleteCurveWithoutTwin`.

Rule 6. ***[Introduction of a curve.]*** The control condition for the TU is:

IntroduceCurve(`C`) =
`addCurve(C);`

asLongAsPossible `extendZone(C)` **end**;
asLongAsPossible `completeExtension(C)` **end**;
asLongAsPossible `copyInhabitInNewZone` **end**;
asLongAsPossible `copyTieInNewZone` **end**;
asLongAsPossible `copyStrandInNewZone` **end**;
asLongAsPossible `ensureCoherentShadingInNewzone` **end**;
asLongAsPossible `removeTwin` **end**

After applying the unit to introduce a curve c, each of the *old* zones which were in the start graph will be the twin (w.r.t. c) of a *new* zone inside c in the target graph. First, rule `addCurve(C)` adds the new curve (see Figure 25) and creates a zone which has the outermost zone as its twin, preserving constraints $P3$ and $P4$. In the rest of the unit, the existence of a *twin* edge indicates the relation between an old zone and a new one, which, at the end of the process, will satisfy the twin relation.

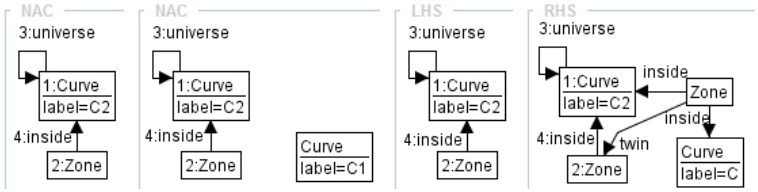


Fig. 25 Rule `addCurve(C)`. The additional checks `C2`$\neq$`C` and `C1`$\neq$`C`$\wedge$`C2`$\neq$`C` on the two NACs (from left to right) are not shown graphically.

A pair of NACs is used to check that the value of `C` differs from that of any other curve. The GAC of Figure 26 identifies the zone `2:Zone` inside the boundary curve `1:Curve` as the outermost one if for each other curve (`5:Curve` in the compartment labelled $\forall$) it does not happen that `2:Zone` is inside `5:Curve` (compartment labelled $\neg$).

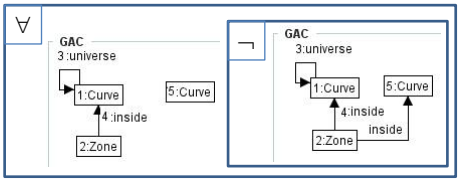


Fig. 26 A graphical representation of the GAC to identify the outermost zone, to be checked on each match m for the LHS of the rule `addCurve(C)` in Figure 25. The formula reads "for each match $m_1 : g_1 \to G$ of the graph g_1 in the $\forall$-box extending m, the graph in the $\neg$-box has no match extending m_1".

The two following iterations generate a number of twins for the old zones, and copy the *inside* relations between old zones and other curves, to their

new twins. In particular, rule `extendZone` of Figure 27, creating new zones which are twins to old ones with respect to c, is guarded by two NACs, preventing repeated application of the same rule on the same match.

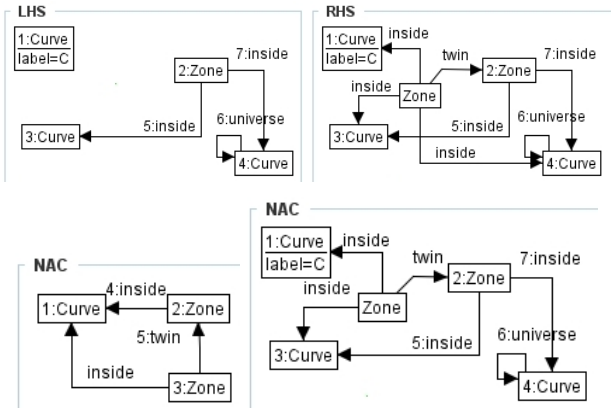


Fig. 27 Rule `extendZone`.

After this, the iteration of rule `completeExtension` in Figure 28 establishes the new zones as inside the same set of other curves as their twins with respect to c. A NAC, not shown here, prevents the application of this rule if it would duplicate an *inside* edge. The nested constraint of Figure 12, stating that no two zones have the same relations with respect to all curves, may not hold after some application of `completeExtension`, but is guaranteed to hold at the end of the iteration of this rule.

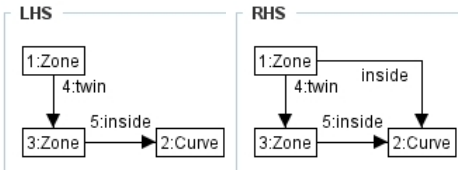


Fig. 28 Rule `completeExtension`.

The following three iterations (see Figures 29-30) take care of the spiders inhabiting the old zones and their connections. All of these have to be duplicated in the new zones. The first rule ensures that the new zone is inhabited by the same spider as its twin zone, while the next two copy ties and strands to the new zones. We show only the version for ties in Figure 30.

To complete the process, rule `ensureCoherentShadingInNewZone` in Figure 31 makes the new zone shaded, if its twin was shaded. The unit is concluded by removing all the auxiliary twin edges, as shown in Figure 32.

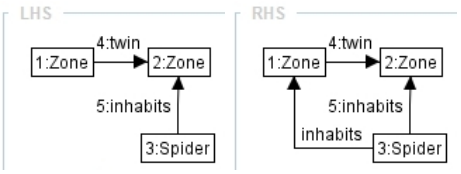


Fig. 29 Rule `copyInhabitInNewZone`.

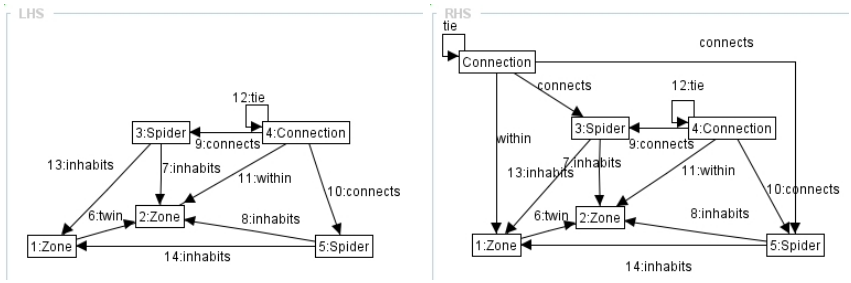


Fig. 30 Rule `copyTieInNewZone`.

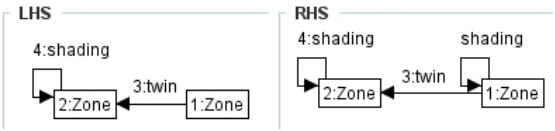


Fig. 31 Rule `ensureCoherentShadingInNewZone`.

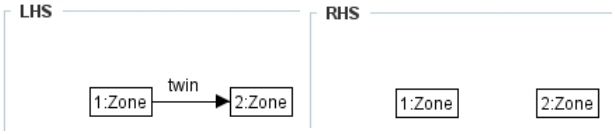


Fig. 32 Rule `removeTwin`.

7 Correctness of translation

From Definition 4 and Theorems 1 and 2, each SD has a representation as an SG, and for each SG there is an SD that SG represents. We now prove the correctness of the construction of $\mathcal{TU}_{SG}$ showing, via Definition 6 and Theorem 3, that each $TU \in \mathcal{TU}_{SG}$ transforms an SG into an SG, realising the specification of an SD rule on the corresponding SD.

Definition 6 *Let T denote a transformation unit and R an SD-rule. Then we say that T* realises *R if $\forall d \in \mathcal{SD}$:*

1. *If d does not satisfy the preconditions of R then the application of T to $G(d)$ fails, where $G(d)$ is the SG representing d.*

2. *If d does satisfy the preconditions of R then the application of T to $G(d)$ produces a Spider Graph G' such that the Spider diagram d' which G' represents satisfies the post-conditions of R.*

Theorem 3 *Let rule R be an SD rule, and let R^* denote the corresponding $TU \in \mathcal{TU}_{SG}$. Then the application of R^* to any element of $\mathcal{SG}$ returns an element of $\mathcal{SG}$. Furthermore, R^* realises R.*

Proof We adopt the following proof strategy for each of the rules:

1. Provide a formal specification for SD-rule R.
2. Assume that $d \in \mathcal{SD}$ satisfies the preconditions of R and let $G(d) \in \mathcal{SG}$ be the SG representation of d.
3. The ordered rules within R^* alter $G(d)$. We indicate, via a sequence of transformations, the corresponding effects of each of these ordered rules within R^* on the components of d. The application of these individual transformations is not guaranteed to preserve the class of SDs (i.e. they may perform some intermediary step whose result is not a SD).
4. Argue, via case analysis when necessary, that the application of this sequence of transformations applied to d yields $d' \in \mathcal{SD}$.
5. Deduce that G', the result of applying R^* to $G(d)$, represents d'.

The details for each of the Rules 5 and 6 are provided in lemmata 2 and 3, whilst those for the remaining rules are provided in the Appendix. □

In order to ensure a precise definition of the effect of each of the SD rules, we provide a specification in a variant of Z. Pre- and post-conditions of the rules are given within the context of a system invariant, the SD definition in Definition 1, since before and after rule application we must have a valid SD. The rules may use parameters indicating named objects (labels for curves or spiders) for directed rule application. Within the specifications, variables that are not passed as parameters or explicitly defined or quantified are implicitly universally quantified. A SD-rule specification has: (i) signature indicating system name, rule names and parameters; (ii) pre- and post-condition contracts placed in boxes separated by a double line (boxes can be omitted if vacuous). Pre-boxes are indicated with a $\triangle$ symbol, post-boxes with a $\triangledown$ symbol. Boxes can be nested, where containment of boxes indicates a conjunction of conditions, whilst disjoint boxes at the same level of nesting indicate alternative cases; two such disjoint boxes with equal pre-conditions indicate a non-deterministic choice. Lemma 2 shows the SD-specification for erasing a curve, utilising nested boxes, for example.

We consider the effect that iterating individual SG rules as long as possible would have on the components of the corresponding SD, and we record this in an abbreviated form in order to save space. That is, we present the effects on the corresponding SDs via an expression of the pre- and post-states, in terms of components of the abstract SD syntax. To emphasize the change from pre- to post-state, we add a prime to the sets in the post-state (returning to non-primed use in the pre-state of the subsequent rule to avoid

notation cluttering). We use the symbol of entailment ($\vdash$) to indicate that if the preconditions (on the left of the $\vdash$) hold in the (SD represented by the) graph before the (iterated) application of the rule, the post-conditions (on the right of the $\vdash$) hold after this application. The individual rule specifications can be obtained by considering the state of the SD corresponding to LHS of rules before the application and the state of the SD corresponding to RHS of rules afterwards. Forbidden graphs and their corresponding NACs, as well as other NACs in the rule, give rise to negated clauses in the pre-condition. For example, the clause $z_2 \notin h(s)$ in entailment (4) below occurs since the `extendInhabitRelation` rule (in Figure 17) is not applied if $z_2 \in h(s)$ in the pre-state (because this would cause the forbidden creation of two *inhabits* edges from the spider to the zone).

The proof of correctness follows by showing that the combined effect of the whole TU on an initial SG, g_1, yields an SG, g_2, and that d_2, the SD corresponding to g_2, satisfies the post-conditions of the SD rule provided d_1, the SD corresponding to g_1, satisfied the precondition of that rule.

Lemma 2 *The TU for $R =$ eraseCurve realises Rule 5 of the SD system.*

Proof We first provide the SD-specification of Rule 5.

$SD!eraseCurve(c \in \mathcal{C})$

$\triangle\ z_1 = (X \cup \{c\}, Y) \in \mathcal{Z} \vee z_2 = (X, Y \cup \{c\}) \in \mathcal{Z}$

$\triangle\ twins_c(z_1, z_2)$

$\triangle\ z_1, z_2 \in \mathcal{Z}^*$

$\triangledown\ (X, Y) \in \mathcal{Z}^*$

$\triangle\ \exists s \in \mathcal{S} \bullet z_1 \in h(s) \vee z_2 \in h(s)$

$\triangledown\ (X, Y) \in h(s) \wedge z_1, z_2 \notin h(s)$

$\triangle\ \exists s, t \in \mathcal{S} \bullet (\{z_1, z_2\}) \cap h(s) \cap h(t) \neq \emptyset$

$\triangle\ (\{s, t\}, z_1) \in \tau \cup \upsilon\ \vee\ (\{s, t\}, z_2) \in \tau \cup \upsilon$

$\triangledown\ (\{s, t\}, (X, Y)) \in \upsilon$

$\triangledown\ c \notin \mathcal{C} \wedge (X, Y) \in \mathcal{Z} \wedge z_1, z_2 \notin \mathcal{Z}$

Next, we specify the effects that looping on rule applications within the TU on the SG would have on the corresponding abstract syntax of SDs, in the order that they appear within the TU: `associateTwin`, `extendInhabitRelation` (4), `copyStrandToSurvivorZone` (5), `ensureConnectionsAreStrandInSurvivorZone` (6), `removeConnectionFromCondemnedZone` (7), `ensureCoherentShadingInSurvivorZone` (8), `removeCondemnedZone` (9), `deleteCurveWithTwin` (10), `deleteCurveWithoutTwin` (11).

The rule `associateTwin` in the *eraseCurve* TU has no effect on the abstract SD, but ensures that twin zones are correctly identified, whilst the subsequent rules act as indicated in the following entailments.

$$twin_c(z_1, z_2) \wedge \exists s \in \mathcal{S}[z_1 \in h(s) \wedge z_2 \notin h(s)] \vdash z_2 \in h'(s). \tag{4}$$

$$twin_c(z_1, z_2) \wedge (\{s_1, s_2\}, z_1) \in \tau \cup \upsilon \wedge (\{s_1, s_2\}, z_2) \notin \tau \cup \upsilon \vdash (\{s_1, s_2\}, z_2) \in \upsilon'. \tag{5}$$

$$twin_c(z_1, z_2) \wedge (\{s_1, s_2\}, z_2) \in \tau \vdash (\{s_1, s_2\}, z_2) \notin \tau' \wedge (\{s_1, s_2\}, z_2) \in \upsilon'. \tag{6}$$

$$twin_c(z_1, z_2) \wedge (\{s_1, s_2\}, z_1) \in \tau \cup \upsilon \vdash (\{s_1, s_2\}, z_1) \notin \tau' \cup \upsilon'. \tag{7}$$

$$twin_c(z_1, z_2) \wedge z_2 \in \mathcal{Z}^* \wedge z_1 \notin \mathcal{Z}^* \vdash z_2 \notin \mathcal{Z}^{*\prime}. \tag{8}$$

$$twin_c(z_1, z_2) \wedge z_1 = (X \cup \{c\}, Y) \wedge \exists X_3, Y_3 \subset \mathcal{C} [X_3 \neq X \wedge (X_3 \cup \{c\}, Y_3) \in \mathcal{Z}] \vdash z_1 \notin \mathcal{Z}'. \tag{9}$$

$$c \in \mathcal{C} \wedge twin_c(z_1, z_2) \wedge z_1 = (X \cup \{c\}, Y) \in \mathcal{Z} \wedge \nexists z_4, z_5 \in \mathcal{Z} [\{z_4, z_5\} \cap \{z_1, z_2\} = \emptyset \wedge z_4 = (X_4 \cup \{c\}, Y_4) \wedge twins_c(z_4, z_5)] \vdash c \notin \mathcal{C}' \wedge z_1 \notin \mathcal{Z}' \wedge z_2 \notin \mathcal{Z}' \wedge (X, Y) \in \mathcal{Z}'. \tag{10}$$

$$c \in \mathcal{C} \wedge \nexists z_1, z_2 \in \mathcal{Z}[twins_c(z_1, z_2)] \vdash c \notin \mathcal{C}'. \tag{11}$$

We perform a case analysis of the overall effects that the sequence of transformations above have on a diagram d, satisfying the preconditions of the rule R, arguing that the output satisfies the post-conditions of rule R. Firstly, suppose that exactly one of $z_1 = (X \cup \{c\}, Y) \in \mathcal{Z}$ and $z_2 = (X, Y \cup \{c\}) \in \mathcal{Z}$ hold. Then the zone z_1 or z_2 which is present is not a twin (i.e. $twin_c(z_1, z_k)$ and $twin_c(z_k, z_2)$ are false for any $z_k \in \mathcal{Z}$). Since all of the other rule specifications affect only twin zones, only entailment (11) holds, and only if there are no c-twins present. In this case, the curve is removed, altering zones in turn, as stated in the post-condition of *SD!eraseCurve*. Curve deletion in the SG removes the *inside* relation between the curve and all zones, thereby removing the c label from zones in the SD representation. Thus the outermost pre- and post-condition pair is satisfied.

Now suppose that $z_1 = (X \cup \{c\}, Y)$ and $z_2 = (X, Y \cup \{c\}) \in \mathcal{Z}$ exist. Then, in the first nested level of conditions, the precondition $twins_c(z_1, z_2)$ holds, and we consider the remaining sub cases, where $z = (X, Y)$.

Case 1: If $z_1, z_2 \in \mathcal{Z}^*$ then $z = (X, Y) \in \mathcal{Z}^{*\prime}$, where the prime indicates the change to post-state in the global specification (as opposed to the local

case when we referred to the equations), since z_2 was shaded and no rules apply to change this (entailment (8) only holds if z_1 was not shaded, and no other entailment holds). This deduction follows from the construction, making use of the fact that the application of the transformation unit effectively removes the inside twin (corresponding to z_1) whist keeping the outside twin (corresponding to z_2), albeit with all references to curve c removed; thus all other attributes of z_2 are inherited by z. If either of z_1 or z_2, or both, are in $\mathcal{Z}$ but not in $\mathcal{Z}^*$ then $z \notin \mathcal{Z}^{*\prime}$ by construction since, if entailment (8) holds, then the shading is removed from z_2.

Case 2: $\exists s \in \mathcal{S}[z_1 \in h(s) \vee z_2 \in h(s)]$. Entailment (4) ensures that after rule application (if it is applicable) we have that $z_2 \in h(s)$. The subsequent curve deletion ensures that z retains the properties of z_2, as indicated in Case 1, and so $z \in h'(s)$.

Case 3: $\exists s, t \in \mathcal{S}[\{z_1, z_2\} \cap h(s) \cap h(t) \neq \emptyset]$ and either $(\{s, t\}, z_1) \in \tau \cup \upsilon$ or $(\{s, t\}, z_2) \in \tau \cup \upsilon$. By entailment (5) a strand is added between s and t in z_2 if there were no connections in z_2 but there were in z_1. Then by entailment (6) any tie in z_2 is converted to a strand. This ensures there is a strand between s and t in z_2 after entailment (6), so $(\{s, t\}, z) \in \upsilon'$.

Finally, the curve $c \in \mathcal{C}$ is deleted by the last rule which is applied in a given execution of a TU. This happens as specified by entailment (10) if there were twin zones with respect to c, or by entailment (11) if no such twin zones were present. Thus the outermost pre- and post-condition pair are satisfied. Equations (7) and (9) perform housekeeping tasks related to the subsequent removal of the zone z_1 (the inside twin).

Hence, the sequence of transformations of the components of d corresponding to the execution of the TU for $R = EraseCurve$ on the representation of d, yields a diagram d' that is obtainable from d by the application of rule R. Due to the correspondence between SDs and SGs from the construction of Definition 4 and Theorem 2, we have that R^* realises R. □

We provide an example tracking the effects of entailments (4)-(11) in detail; that is, we track the effect on the SD corresponding to each of the application steps in the corresponding TU acting on the Spider Graphs.

Example 1 Let d_1 be the SD shown in Figure 33 (top left), with a to be deleted in the rule application yielding d_2 on the right. Since the habitat of both s and t, in d_1, includes both of the twinned zones $((\{u, a, b\}, \{\})$ and $(\{u, b\}, \{a\}))$, entailment (4) does not affect them. For the twinned zones $((\{u, a\}, \{b\})$ and $(\{u\}, \{a, b\}))$, we have $(\{u, a\}, \{b\}) \in h(t)$, $(\{u\}, \{a, b\}) \notin h(t)$, and so $(\{u\}, \{a, b\})$ is added to the habitat of t (bottom left). Equation (5) has no effect, as the twinned pair $((\{u, a, b\}, \{\}), (\{u, b\}, \{a\}))$, with $(\{s, t\}, (\{u, b\}, \{a\})) \in \tau$, $(\{s, t\}, (\{u, a, b\}, \{\})) \notin \tau \cup \upsilon$ does not satisfy its pre-condition: z_1 must be the zone which is inside the curve a. Next, entailment (6) replaces the tie $(\{s, t\}, (\{u, b\}, \{a\})) \in \tau$, with a strand so that $(\{s, t\}, (\{u, b\}, \{a\})) \in \upsilon$ (bottom middle). Then, entailment (7) has no effect since there are no ties or stands between spiders' feet in any twinned zone which is inside a. Equation (8) also has no effect since the

only shading is in a zone which is inside a. The effect of entailment (9) is to remove any twinned zone which is inside a, provided there exists another zone inside a, thereby non-deterministically removing exactly one of the zones $(\{u, a, b\}, \{\})$, $(\{u, a\}, \{b\})$; the case of the removal of $(\{u, a, b\}, \{\})$ is shown in the bottom right. Finally, entailment (10) holds, with z_1 being the remaining zone inside a (there being no other inside twinned zone z_3), deleting curve a and the zone z_1, yielding d_2.

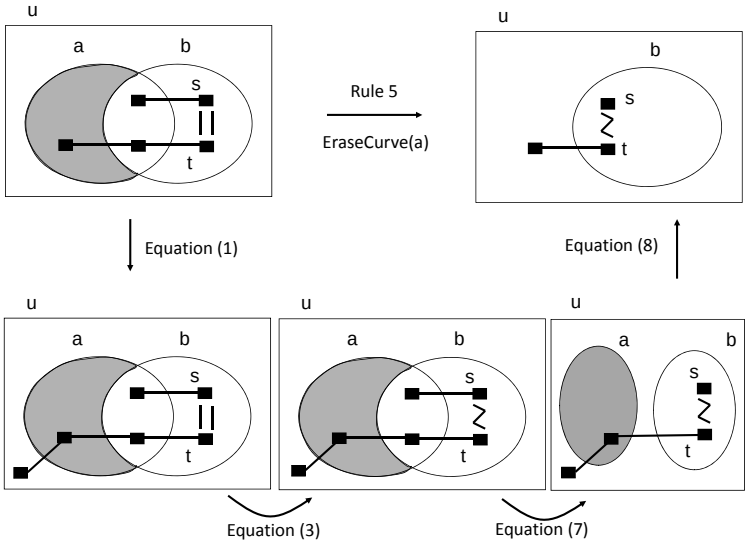
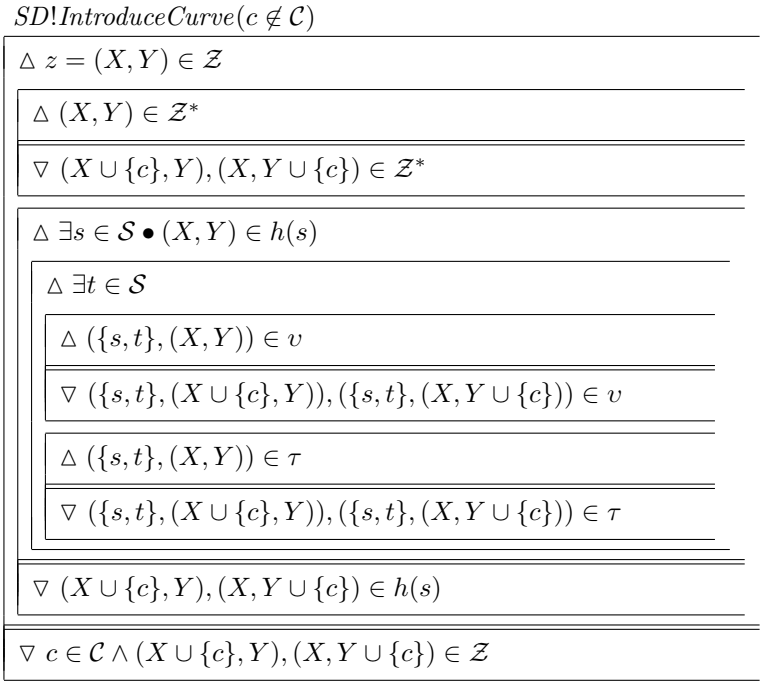


Fig. 33 An example application of the *Erase Curve* rule on a SD. Curve a is erased from diagram d_1 (top left), yielding diagram d_2 (top right). The other path depicts the effective changes that would occur to the SD corresponding to the SG rule applications within the TU for this rule.

Lemma 3 *The TU for* IntroduceCurve *realises Rule 6 of the SD system.*

Proof The SD specification is given by:



The order of rules for the *IntroduceCurve* TU is: `addCurve` (12), `extendZone` (13), `completeExtension` (14), `copyInhabitInNewZone` (15), `copyTieInNewZone` (16), `copyStrandInNewZone`, `ensureCoherentShadingInNewZone` (17), `removeTwin`.

We provide a commentary indicating the effects on the SD interleaved with the entailments presented for the rules. The first rule adds the new curve together with a new zone with which has the outermost zone as a twin; this corresponds to adding the new curve to the SD disjoint from all of the other curves; the extra zones are added later.

$$\begin{aligned} c \notin \mathcal{C} \wedge (\{u\}, \mathcal{C} \setminus \{u\}) \in \mathcal{Z} \wedge (X, Y) \in \mathcal{Z} \vdash \\ c \in \mathcal{C}', (\{c, u\}, \mathcal{C}' \setminus \{c, u\}) \in \mathcal{Z}', \\ (\{u\}, \mathcal{C}' \setminus \{u\}) \in \mathcal{Z}', (X, Y \cup \{c\}) \in \mathcal{Z}'. \end{aligned} \tag{12}$$

The next two entailments collectively define the construction of the set of all zones which are twins to previously existing zones (which were not the outermost zone), while the following one extends to the new zones feet of spiders with feet in the existing twin zones.

$$\begin{aligned} (X \cup \{c_1, u\}, Y \cup \{c\}) \in \mathcal{Z} \wedge (X \cup \{c, c_1, u\}, Y) \notin \mathcal{Z} \vdash \\ (X \cup \{c, c_1, u\}, Y) \in \mathcal{Z}'. \end{aligned} \tag{13}$$

$$\begin{aligned} (X_2 \cup \{c_1\}, Y_2 \cup \{c\}) \in \mathcal{Z} \wedge (X_1 \cup \{c\}, Y_1) \in \mathcal{Z} \wedge X_1 \subseteq X_2 \wedge \\ (X_2 \cup \{c, c_1\}, Y_2) \notin \mathcal{Z} \vdash (X_2 \cup \{c, c_1\}, Y_2) \in \mathcal{Z}'. \end{aligned} \tag{14}$$

$$(X, Y \cup \{c\}) \in h(s) \wedge (X \cup \{c\}, Y) \in \mathcal{Z} \setminus h(s) \vdash (X \cup \{c\}, Y) \in h'(s). \quad (15)$$

The subsequent two rules copy existing connections from old zones to new twin zones (we show the version for ties) and ensure equal shading in both twin zones. Finally, rule `removeTwin` removes the auxiliary twin edges.

$$(\{t_1, t_2\}, (X, Y \cup \{c\})) \in \tau \vdash (\{t_1, t_2\}, (X \cup \{c\}, Y)) \in \tau'. \quad (16)$$

$$(X, Y \cup \{c\}) \in \mathcal{Z}^* \wedge (X \cup \{c\}, Y) \in \mathcal{Z} \vdash (X \cup \{c\}, Y) \in \mathcal{Z}^{*\prime}. \quad (17)$$

The overall effect is to copy every old zone in the pre-state, together with its shading, any *inhabit* relation from spiders, and any connection between such spiders, producing a new twin zone in the post-state with the same properties as the old zone, thus satisfying the SD-rule specification. □

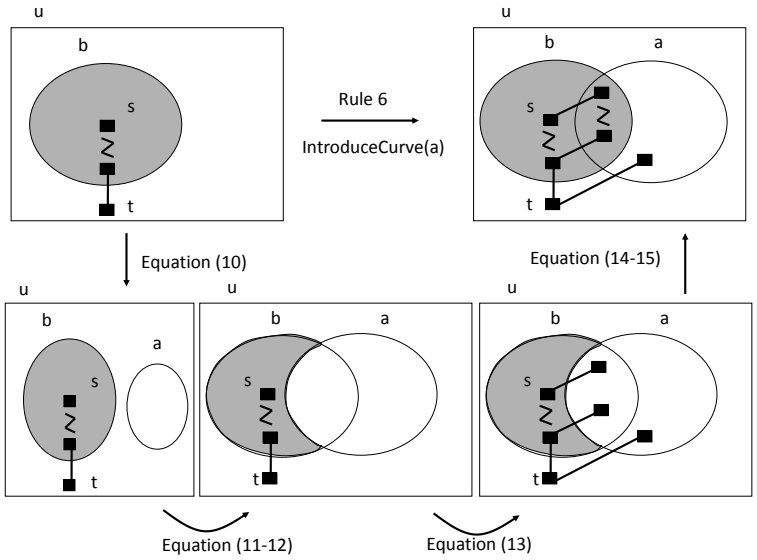


Fig. 34 An example application of the *Introduce Curve* rule on a SD. Curve a is introduced into diagram d_1 (top left), yielding the diagram d_2 (top right). The other path depicts the effective changes that would occur to the SD corresponding to the SG rule applications within the TU for this rule.

Example 2 Let d_1 be the SD shown on the left hand side of Figure 34, with a the curve to be introduced to yield d_2 on the right. Equation (12) adds a and a new zone inside a and u, shown at the bottom left. Equation (13) adds any missing zones outside a but currently with a twin which is inside

a curve different from u or a. Equation (14) adds the twins which are inside a of all zones created by entailment (13); collectively this has the effect of updating any zone z_1 in the old zone set by adding a to its outside curve set, and creating the corresponding z_2 (a twin of z_1 with a in the inside curve set). This produces the zone $(\{u, a, b\}, \{\})$, shown in the bottom middle. Equation (15) copies all feet of the spiders from the old zones into their new a-twin, as shown in the bottom right. Finally, entailments (16) and (17) copy the ties (and analogously strands) and shading from the old zones into the new zones, twins w.r.t. a, yielding d_2, as required.

8 Discussion

We discuss features of the current realisation of SGs and alternatives to it. The current implementation in AGG [37], combining interactive specification of parameters and automatic application of rules, allows us to achieve two main goals. On the one hand, we have a fully formalised procedure, expressed in terms of TUs, which can be exactly followed with AGG. On the other hand, we have access to tools for conflict and dependency analysis based on critical pairs [12], which can be used to reason on applicability of rules and sequences and relations among them. Indeed, an analysis on the principal rules in each TU can be used to guide proof strategies to demonstrate that a certain diagram d' can be derived from a diagram d. For example, rules for deleting and introducing a curve are dependent on one another, as each one creates a condition for the application of the other, but no proof actually requires the application of both of them to the same curve. As another example, the existence of a conflict (meaning that the application of one rule disrupts conditions for the application of the other) between `extendSpiderToZone` and `deleteSpider`, in which the application of the second rule deletes context used by the first one, indicates that proof strategies can be made more efficient by avoiding the use of the *Extend-Habitat* SD rule (in the Appendix) for spiders which must be later deleted.

We have adopted TUs as the general framework for the specification of SG transformations. However, one can observe that different mechanisms can be employed in most of the cases. In particular, in AGG it is possible to describe rule sequences, where rules (or subsequences) have to be applied in the order in which they appear for an indicated number of iterations, including '$*$', which corresponds to iteration as long as possible. This construct can be used to realise TUs which do not include alternative choices. Alternatives can be managed by the layering construct, also available in AGG. In this case, one assigns rules to layers, subject to some compatibility condition [6], ensuring that no dependencies or conflicts arise between rules in the same layer. The rules in one layer are applied as long as possible (at each iteration choosing non-deterministically one of them), before moving to the next layer. This mechanism, however, does not allow nesting of sequences. As no TU employs both nesting and alternative choice, AGG supports a full mechanisation of the process.

In many cases, looping on a rule is needed only to make sure that all possible matches are checked. This can be more simply realised via parallel application and amalgamation of rules [50], which is also available (in an experimental version) in AGG. Another frequent use for looping is to ensure that nodes or edges are copied when preparing or repairing a context for a principal rule. This might be specified through the use of the Sesqui-Pushout approach, based on coupling a pullback and a pushout construction, with the pullback providing cloning of edges between nodes identified in the LHS and kept distinct in the RHS. However, it cannot completely substitute some of the iterations, in which nodes, rather than edges, have to be copied, such as in the case of copying connections. In any case, we have chosen not to follow the Sesqui-Pushout approach in our aim to a mechanisation, as no sufficient tool support for it is currently available.

9 Conclusions

The mechanisation of diagrammatic reasoning (DR) systems opens new possibilities for integrating such systems within visual modeling environments, and restricts the need for complementing the latter with textual constraint languages. In particular, languages based on extensions of Euler Diagrams (EDs), such as Constraint Diagrams (CDs) have been proposed as a way to achieve precise modeling capabilities within a completely visual setting.

In this paper, we presented a first step in this direction, utilising the language of Spider Diagrams (SDs). Besides being of interest in itself, this language provides a foundation for CDs, adding the possibility of expressing explicit quantification and relations. In particular, we defined the language of Spider Graphs (SGs), providing a complete translation of the unitary system of SDs. This enables the formalization of the SD deductive reasoning system in terms of graph transformations (GTs) – namely transformation units (TUs) enabling the ordered application of sequences of graph transformation rules – on a class of graphs modeling the language of SDs.

Bringing together DR and GTs enables the use of a large body of theory and tools, which can assist the development and analysis of diagrammatic reasoning systems and of their proof strategies. In particular, formal tools become available to analyse aspects such as parallel and sequential independence of derivations, or parallel and concurrent deductive rules. Furthermore, when dealing with strict deductive steps (i.e., steps with loss of information, and not just logical equivalences), the notion of critical pairs [31] allows reasoning on appropriate deductive strategies. The approach based on TUs can be extended to support the definition of "derived rules" for diagrammatic reasoning systems [33] as the combination of atomic simple deductive steps into a single complex deductive step.

From the point of view of the SD system itself, we have presented a formal specification of SD rules via pre/post condition pairs in a Z-variant. This also facilitates the analysis of single deductive steps and reasoning sequences, which is made difficult by the usual presentation of SDs and their

inference rules in algorithmic terms. As future work, different syntactic and semantic variants of diagrammatic reasoning systems can be investigated by analysing the corresponding alterations in the graph based system, exploiting the current AGG implementation. By adopting our approach of keeping all actions explicitly controlled within the TUs developed, we facilitate analysis of alteration in semantics of the system or variations of the reasoning rules, permitting a deeper understanding of choices made when designing diagrammatic logical reasoning systems.

ED-based representations are also used for set-based information visualisation, where the representation of elements or items effectively yields a SD, although with different semantics (e.g. [3, 9, 51]). In such applications, one often visualises sequences of diagrams representing some set-based data changes. Hence, besides logical inference rules, also rules indicating data changes become essential. By adopting the same approach, one can specify such transformation rules within the same setting and using the same tools.

Acknowledgments We thank the anonymous referees for many insightful comments on the previous version which have helped us to greatly improve the paper. Thanks to John Taylor for comments on an early draft. Andrew Fish was partially funded by UK EPSRC grants EP/E011160: Visualisation with Euler Diagrams and EP/J010898/1: Automatic Diagram Generation. We also thank the AGG team, in particular Claudia Ermel and Olga Runge, for assistance with the implementation.

References

1. Barwise, J., Etchemendy, J.: Hyperproof. CSLI (1994)
2. Barwise, J., Etchemendy, J.: Visual information and valid reasoning. In: Allwein, G., Barwise, J. (eds.) Logical Reasoning with Diagrams, pp. 3–25. OUP (1996)
3. Bottoni, P., Fish, A.: Coloured Euler diagrams: A tool for visualizing dynamic systems and structured information. In: Proc. Diagrams 2010, *LNCS*, vol. 6170, pp. 39–53 (2010)
4. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: Consistency checking and visualization of OCL constraints. In: Proc. UML 2000, *LNCS*, vol. 1939, pp. 294–308 (2000)
5. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: A visualization of OCL using collaborations. In: Gogolla, M., Kobryn, C. (eds.) Proc. UML 2001, *LNCS*, vol. 2185, pp. 257–271 (2001)
6. Bottoni, P., Schürr, A., Taentzer, G.: Efficient parsing of visual languages based on critical pair analysis (and contextual layered graph transformation. In: Proc. IEEE-VL'00, pp. 59–61. IEEE CS Press (2000)
7. Cabot, J., Clariso, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: Proc. ICSTW '08., pp. 73–80. IEEE CS Press (2008)
8. Chow, S.C.: Generating and drawing area-proportional Euler and Venn diagrams. Ph.D. thesis, University of Victoria (2007)

9. Cordasco, G., De Chiara, R., Fish, A.: Interactive visual classification with Euler diagrams. In: Proc. VL/HCC 2009, pp. 185–192. IEEE CS Press (2009)
10. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: Proc. ICGT 2006, *LNCS*, vol. 4178, pp. 30–45. Springer (2006)
11. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Theory of constraints and application conditions: From graphs to high-level structures. Fundam. Inform. **74**(1), 135–166 (2006)
12. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graphs and graph transformation based on adhesive HLR categories. Fundam. Inform. **74**(1), 31–61 (2006)
13. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
14. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic approaches to graph transformation - Part II: Single Pushout Approach and comparison with Double Pushout Approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, pp. 247–312. World Scientific (1997)
15. Ehrig, K., Küster, J.M., Taentzer, G., Winkelmann, J.: Generating instance models from meta models. In: Proc. FMOODS 2006, *LNCS*, vol. 4037, pp. 156–170. Springer (2006)
16. Ehrig, K., Winkelmann, J.: Model transformation from VisualOCL to OCL using graph transformation. In: Proc. GT-VMT 2006, *ENTCS*, vol. 152, pp. 23–37 (2006)
17. Euler., L.: Lettres a une Princesse d'Allemagne sur divers sujets de physique et de philosophie. Letters **2**, 102–108 (1775). Berne, Société Typographique
18. Fish, A.: Euler diagram transformations. In: Proc. GT-VMT 2009, *ECEASST*, vol. 18. EASST (2009)
19. Fish, A., Flower, J.: Investigating reasoning with Constraint Diagrams. In: Proc. VLFM 2004, *ENTCS*, vol. 127, pp. 53–69. Elsevier (2005)
20. Fish, A., Flower, J., Howse, J.: The semantics of augmented constraint diagrams. JVLC **16**, 541–573 (2005)
21. Fish, A., John, C., Taylor, J.: A normal form for Euler diagrams with shading. In: Proc. Diagrams 2008, *LNCS*, vol. 5223, pp. 206–221. Springer (2008)
22. Flower, J., Fish, A., Howse, J.: Euler diagram generation. Journal of Visual Languages and Computing **19**, 675–694 (2008)
23. Flower, J., Masthoff, J., Stapleton, G.: Generating proofs with spider diagrams using heuristics. In: Proc. DMS-VLC, pp. 279–285. Knowledge Systems Institute (2004)
24. Flower, J., Masthoff, J., Stapleton, G.: Generating readable proofs: A heuristic approach to theorem proving with spider diagrams. In: Proc. Diagrams 2004, *LNAI*, vol. 2980, pp. 166–181. Springer (2004)

25. Flower, J., Stapleton, G.: Automated theorem proving with spider diagrams. In: Proceedings of Computing: The Australasian Theory Symposium, *ENTCS*, vol. 91, pp. 116–132. Elsevier (2004)
26. Goedicke, M., Meyer, T., Taentzer, G.: Viewpoint-oriented software development by distributed graph transformation: towards a basis for living with inconsistencies. In: Proc. IEEE RE'99, pp. 92–99 (1999)
27. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Inform. **26**(3,4), 287–313 (1996)
28. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science **19**(2), 245–296 (2009)
29. Hammer, E., Shin, S.J.: Euler's Visual Logic. History and Philosophy of Logic pp. 1–29 (1998)
30. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In: Proc. ICSE '02, pp. 105–115. ACM Press (2002)
31. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Proc. ICGT '02, *LNCS*, vol. 2505, pp. 161–176. Springer (2002)
32. Howse, J., Molina, F., Taylor, J., Kent, S., Gil, J.: Spider diagrams: A diagrammatic reasoning system. JVLC **12**(3), 299–324 (2001)
33. Howse, J., Stapleton, G., Taylor., J.: Spider diagrams. LMS J. of Computation and Mathematics **8**, 145–194 (2005)
34. Jamnik, M.: Mathematical Reasoning with Diagrams: From Intuition to Automation. CSLI (2001)
35. Kent, S.: Constraint diagrams: visualizing invariants in object-oriented models. In: Proc. OOPSLA '97, pp. 327–341. ACM Press (1997)
36. Kreowski, H.J., Kuske, S., Schürr, A.: Nested graph transformation units. Int. J. on SEKE **7**(4), 479–502 (1997)
37. de Lara, J., Taentzer, G.: Automated model transformation and its validation using AToM3 and AGG. In: Proc. Diagrams'04, *LNCS*, vol. 2980, pp. 182–198 (2004)
38. Münch, M., Schürr, A., Winter, A.J.: Integrity constraints in the multiparadigm language PROGRES. In: Selected Papers from TAGT'98, *LNCS*, vol. 1764, pp. 338–351. Springer (2000)
39. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1. http://www.omg.org/spec/QVT/1.1/PDF/ (2011)
40. Orejas, F., Ehrig, H., Prange, U.: A logic of graph constraints. In: Proc. FASE 2008, *LNCS*, vol. 4961, pp. 179–198. Springer (2008)
41. Rensink, A.: Representing first-order logic using graphs. In: Proc. ICGT 2004, *LNCS*, vol. 3256, pp. 319–335. Springer (2004)
42. Rensink, A., Schmidt, Á., Varró, D.: Model checking graph transformations: A comparison of two approaches. In: Proc. ICGT 2004, *LNCS*, vol. 3256, pp. 226–241 (2004)

43. Ruskey, F.: A survey of Venn diagrams. Electronic Journal of Combinatorics (1997). www.combinatorics.org/Surveys/ds5/VennEJC.html
44. Shin, S.J.: The Logical Status of Diagrams. CUP (1994)
45. Sowa, J.: Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley (1984)
46. Stapleton, G., Howse, J., Taylor, J.: A decidable constraint diagram reasoning system. J. of Logic and Computation **15**(6), 975–1008 (2005)
47. Stapleton, G., Masthoff, J., Flower, J., Fish, A., Southern, J.: Automated theorem proving in Euler diagram systems. J. of Automated Reasoning **39**(4), 431–470 (2007)
48. Stapleton, G., Thompson, S., Howse, J., Taylor, J.: The expressiveness of spider diagrams. J. of Logic and Computation **14**(6), 857–880 (2004)
49. Swoboda, N., Allwein, G.: Using DAG transformations to verify Euler/Venn homogeneous and Euler/Venn FOL heterogeneous rules of inference. J. of Soft. and Syst. Model. **3**(2), 136–149 (2004)
50. Taentzer, G.: Parallel and distributed graph transformation - formal description and application to communication-based systems. Berichte aus der Informatik. Shaker (1996)
51. Thivre, J., Viaud, M.L., Verroust-Blondet, A.: Using Euler diagrams in traditional library environments. In: Proc. Euler Diagrams'2004, *ENTCS*, vol. 134, pp. 189–202 (2005)
52. Urbas, M., Jamnik, M., Stapleton, G., Flower, J.: Speedith: A diagrammatic reasoner for spider diagrams. In: Proc. Diagrams 2012, *LNCS*, vol. 7352, pp. 163–177. Springer (2012)
53. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise modeling with UML. Addison-Wesley (1999)
54. Zykov, A.: Hypergraphs. Russian Math. Surveys **29**(6), 89–156 (1974)

A Presentation of rules 1-4,7

We complete the presentation of the TUs realising the whole reasoning system, and for each of them, we prove its correctness.

Rule 1. ***[Introduction of a strand.]*** Figure 35 shows the rule for adding strands, realizing the first alternative for Rule 1, where a NAC (not shown here) prevents the formation of the forbidden graph ($F1$) in Figure 8. Figure 36 shows the rule for the second alternative, replacing a tie by a strand. In both cases, the rule receives as parameters the labels associated with the spiders to be connected, and picks non-deterministically a zone inhabited by both spiders. The resulting TU is a choice between the two rules.

IntroduceStrand(`S1,S2`) =
(`addStrand(S1,S2)` | `replaceTieWithStrand(S1,S2)`)

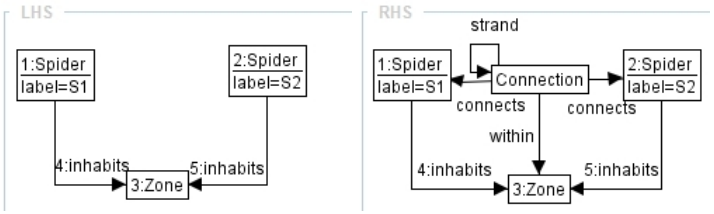


Fig. 35 Rule `addStrand(S1,S2)`.

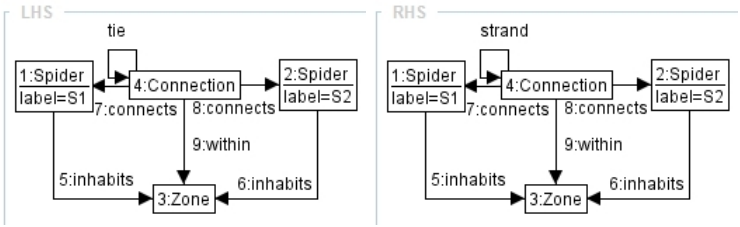


Fig. 36 Rule `replaceTieWithStrand(S1,S2)`.

Lemma 4 *The TU* IntroduceStrand *realises Rule 1 of the SD system.*

Proof The SD rule corresponds to two separate SD specifications:

$SD!addStrand(s, t \in \mathcal{S})$

$\triangle\ \exists z \in \mathcal{Z} \bullet (z \in h(s) \cap h(t)) \wedge ((\{s,t\}, z) \notin \tau \cup \upsilon)$
$\triangledown\ (\{s,t\}, z) \in \upsilon$

$SD!replaceTieWithStrand(s, t \in \mathcal{S})$

$\triangle\ \exists z \in \mathcal{Z} \bullet (z \in h(s) \cap h(t)) \wedge ((\{s,t\}, z) \in \tau)$
$\triangledown\ (\{s,t\}, z) \notin \tau \wedge (\{s,t\}, z) \in \upsilon$

The SG rules `addStrand` and `replaceTieWithStrand` clearly realise these specifications. Note that the condition $(\{s,t\}, z) \notin \tau \cup \upsilon$ for the first rule is guaranteed by its NAC. □

Rule 2. ***[Extension of habitat.]*** The spider to be extended is *chosen* by passing a parameter `S` to the TU, which will pass it on to the invoked rules. The zone in which to extend the chosen spider, and the spiders that are to have connections (of different types) with it added are chosen non-deterministically. The process is managed by a TU with control expression:

ExtendHabitat(`S`) =
`extendSpiderToZone(S)`;
asLongAsPossible `markSpiderForConnections(S)` **end**;
asLongAsPossible (`addStrandForNewExtension(S)` |
`addTieForNewExtension(S)` | `noConnectionAddded`) **end**;
`removeMarkFromZone`;

The process starts with rule `extendSpiderToZone` in Figure 37. A new *inhabits* edge is drawn from the *chosen* spider to a zone, which is marked to specialise the context for the rest of the TU.

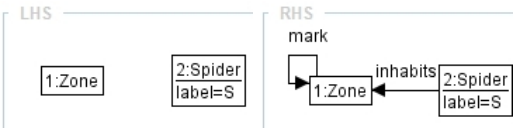


Fig. 37 Rule `extendSpiderToZone`.

The iteration on rule `markSpiderForConnections` in Figure 38 marks each spider, other than the chosen one, inhabiting the marked zone as a candidate for connection. All marks are removed in the second iteration within the TU, where for each candidate spider, a choice is made of whether to add a strand from the chosen spider to the candidate one, to add a tie or do neither. We only present the rule for adding a tie in Figure 39, the case for strand being analogous; the third alternative simply removes the mark from the candidate spider.

Finally, the mark is removed from the zone (rule `removeMarkFromZone`, not shown here). In Lemma 5, in order to track the effects of marking in the SG, we introduce a new temporary set $M \subset \mathcal{Z} \cup \mathcal{C} \cup \mathcal{S}$, and we refer to this in the specification of the rules within the TU.

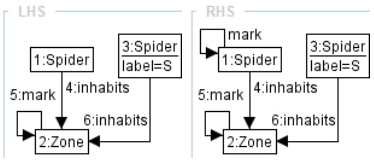


Fig. 38 Rule `markSpiderForConnections`.

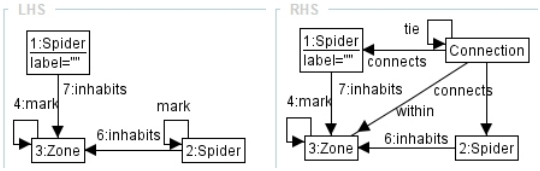


Fig. 39 Rule `addTieForNewExtension`.

Lemma 5 *The TU* ExtendHabitat *realises Rule 2 of the SD system.*

Proof The SD rule has specification:

$SD!extendHabitat(s \in \mathcal{S})$

$\triangle\ \exists z \in \mathcal{Z} \bullet z \notin h(s)$

$\triangle\ \exists t \in \mathcal{S} \bullet z \in h(t)$

$\triangledown\ (\{s,t\},z) \in \tau$

$\triangle\ \exists t \in \mathcal{S} \bullet z \in h(t)$

$\triangledown\ (\{s,t\},z) \in \upsilon$

$\triangle\ \exists t \in \mathcal{S} \bullet z \in h(t)$

$\triangledown\ (\{s,t\},z) \notin \tau \cup \upsilon$

$\triangledown\ z \in h(s)$

The rules `extendSpiderToZone`(18), `markSpiderForConnections`(19), and `addStrandForNewExtension` / `addTieForNewExtension` / `noConnectionAdded`(20) in the TU have effects as follows:

$$z \notin h(s) \cup M \vdash z \in h'(s) \cap M'. \tag{18}$$

$$z \in h(s) \cap M \wedge \exists t \in \mathcal{S}[z \in h(t) \wedge t \notin M] \vdash t \in M'. \tag{19}$$

$$\begin{aligned} z \in h(s) \cap M \wedge \exists t \in \mathcal{S}[z \in h(t) \wedge t \in M \wedge (\{s,t\},z) \notin \tau \cup \upsilon] \vdash \\ (\{s,t\},z) \in \tau' \wedge t \notin M'. \\ z \in h(s) \cap M \wedge \exists t \in \mathcal{S}[z \in h(t) \wedge t \in M \wedge (\{s,t\},z) \notin \tau \cup \upsilon] \vdash \\ (\{s,t\},z) \in \upsilon' \wedge t \notin M'. \\ z \in h(s) \cap M \wedge \exists t \in \mathcal{S}[z \in h(t) \wedge t \in M \wedge (\{s,t\},z) \notin \tau \cup \upsilon] \vdash \\ t \notin M'. \end{aligned} \tag{20}$$

To see that effect of the TU complies with the SD specification, suppose that we have $z \in \mathcal{Z} \wedge z \notin h(s)$. Then, by entailment (18), the habitat of s is extended, satisfying the outer post-condition, and the zone z is marked, ensuring that subsequent rules apply only to this chosen zone. Entailment (19) marks any other spider that also inhabits the zone z. The alternative rules have effects that correspond to the three alternative nested pre-post condition pairs in the SD specification (see entailment (20)). The removal of the mark from the spider for each rule application ensures that each marked spider triggers exactly one of these alternative rules once only, and so the process terminates. All of the marks have been removed in the post-state: the marks of the spiders by the above process, whilst the `removeMarkFromZone` rule precisely ensures that $z \notin M$ in the post-state. □

Before we consider Rule 3, we need some notation.

Definition 7 *Let $G(z), Con(s,z)$ and $Con_s(t,z)$ denote the strand-tie graph within zone z, the set of vertices in the connected component of s in $G(z)$, and the set of vertices in the connected component of t in $G(z) \setminus \{s\}$, resp.*

Rule 3. ***[Erasure of a spider.]*** The deletion of a spider s is prepared by the deletion of all its connections to other spiders, reconstructing the connectedness of the "strand-tie graph" in all the zones inhabited by s, when needed. To this end, we compute the separate connected components resulting from the removal of s, where each component has at least one spider that was directly connected to s in the original graph and which is taken as the *representative* of the component. The process is started by selecting one representative (of one connected component of the strand-tie graph in the given zone after the deletion of s) via the rule `startComputation(S)` of Figure 40, while the marking of the spiders in a connected component is achieved with rule `computeConnectedComponent` of Figure 41. The two NACs prevent the original spider s, or an already marked spider, from being marked. Connections are then created between the representative of the computed component and another component (recognised by being directly connected to s but not marked) to ensure that connectedness of the strand-tie graph is not affected by the deletion of s, through rule `createStrandToOtherComponent` of Figure 42. The representative of the reached component is also marked, so that the process (of computing the connected component and then extending it by adding a connection) can be iterated, until all the spiders

directly connected to the s are now directly or indirectly connected with one another (i.e lie in the same component).

EraseSpider(S) =
asLongAsPossible
`startComputation(S);`
asLongAsPossible
asLongAsPossible `computeConnectedComponent(S)` **end**;
asLongAsPossible `createStrandToOtherComponent(S)` **end**;
`removeZoneMark` **end**;
end;
asLongAsPossible `removeConnection(S)` **end**;
asLongAsPossible `removeSpiderMark` **end**;
`deleteSpider(S)`

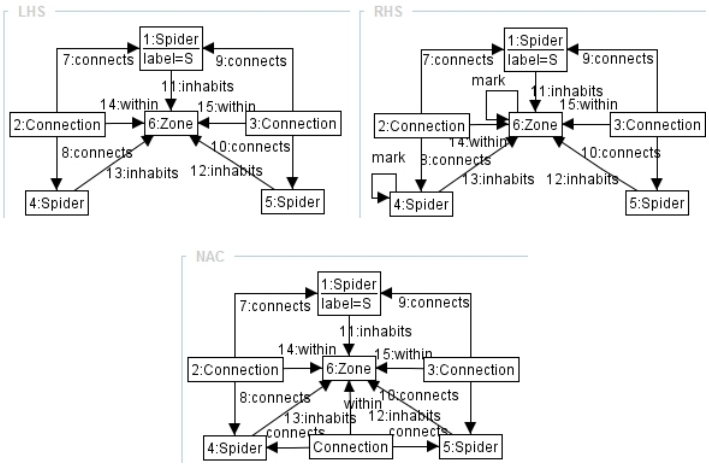


Fig. 40 Rule `startComputation`.

At the end of this iteration, all the connections associated with the spider can be removed by rule `removeConnection` in Figure 43 (left), whose SPO application also removes the *connects* edge to any other spider, and the spider is finally deleted by rule `deleteSpider` in Figure 43 (right). Note that the NAC would make this rule fail if the spider inhabited some shaded zone, making the whole TU fail. Such a check could also be performed by an additional rule at the beginning of the process.

Lemma 6 *The TU* EraseSpider *realises Rule 3 of the SD system.*

Proof The SD rule has specification:

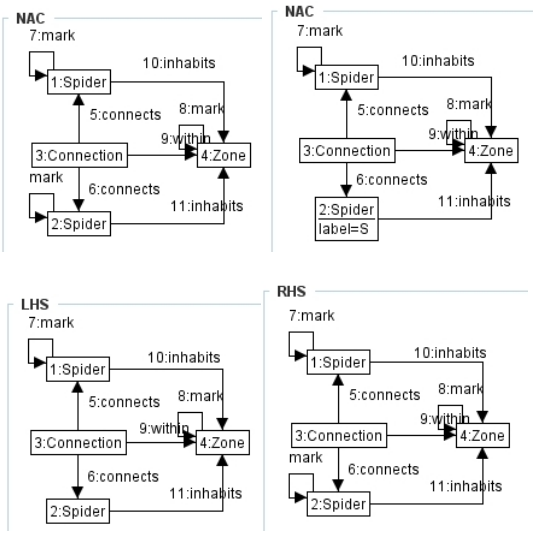


Fig. 41 Rule `computeConnectedComponent`.

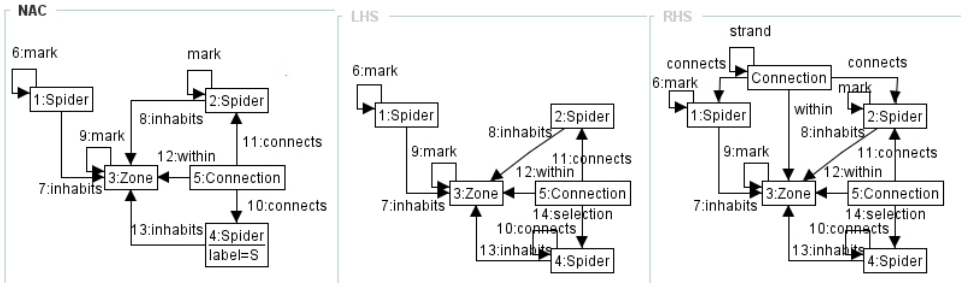


Fig. 42 Rule `createStrandToOtherComponent`.

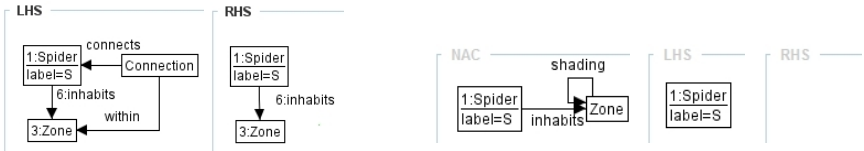


Fig. 43 Rules `removeConnection` and `deleteSpider`.

$SD!eraseSpider(s \in \mathcal{S})$

$\triangle\ h(s) \cap \mathcal{Z}^* = \emptyset$

$\triangle\ \exists z \in \mathcal{Z}, r, t \in \mathcal{S} \bullet (r, t \neq s) \wedge (r \neq t) \wedge (r, t \in Con(s, z))$

$\nabla\ r \in Con(t, z)$

$\nabla\ s \notin \mathcal{S}$

The rules `startComputation`(21), `computeConnectedComponent`(22), `createStrandToOtherComponent`(23), `removeConnection`(24), `deleteSpider`(25) in the TU have effects as follows:

$$z \in h(s) \cap h(t_1) \cap h(t_2) \wedge (\{s, t_1\}, z), (\{s, t_2\}, z) \in \tau \cup \upsilon \wedge (\{t_1, t_2\}, z) \notin \tau \cup \upsilon \vdash z, t_1 \in M'. \tag{21}$$

$$z, t_1 \in M \wedge t_2 \notin M \wedge t_2 \neq s \wedge z \in h(t_1) \cap h(t_2) \wedge (\{t_1, t_2\}, z) \in \tau \cup \upsilon \vdash t_2 \in M'. \tag{22}$$

$$z \in h(s) \cap h(t_1) \cap h(t_2) \wedge z, t_1 \in M \wedge t_2 \notin M \wedge (\{s, t_2\}, z) \in \tau \cup \upsilon \vdash (\{t_1, t_2\}, z) \in \upsilon' \wedge t_2 \in M'. \tag{23}$$

$$z \in h(s) \wedge (\{s, t\}, z) \in \tau \cup \upsilon \vdash (\{s, t\}, z) \notin \tau' \cup \upsilon'. \tag{24}$$

$$h(s) \cap \mathcal{Z}^* = \emptyset \vdash s \notin \mathcal{S}'. \tag{25}$$

We consider the effects of the nested part of the control condition of the TU, before considering the SD specification. The `startComputation` rule finds, and marks, an arbitrary (representative) spider t_1 that is connected to spider s but is not connected to every other spider that s is connected to (see entailment (21)). If there are no such spiders then the removal of spider s cannot disconnect the strand-tie graph in z since every pair of spiders connected to s within z are already connected to each other; note that in this case no match for this SG rule is found and the TU proceeds to `removeConnection`. Otherwise, iterating `computeConnectedComponent` (see entailment (22)) repeatedly marks every spider (except for s itself) which is connected to a marked spider (by a connection), ending when all of $Con_s(t_1, z)$ is marked. If there is a spider t_2 such that $t_2 \notin Con_s(t_1, z)$ (hence t_2 is not marked) but there is a connection between s and t_2 within z, then rule `createStrandToOtherComponent` (see entailment (23)) adds a strand between t_2 and t_1 within z. The iteration in the control condition marks all of the extended connected component (which is the union of the original connected components of t_1 and t_2 together with the new strand that joins them). This is repeated until there is only a single connected component. At this point the deletion of the spider s will not disconnect the strand-tie graph. The `removeZoneMark` rule just removes the marking on the zone. So the total effect of this nested part of the control condition is to add enough strands to ensure that the deletion of spider s does not disconnect any component of the strand-tie graph, satisfying the inner pre-/post-condition pair. Now, suppose we have a spider s such that $h(s) \cap \mathcal{Z}^* = \emptyset$, so the pre-condition of the *eraseSpider* SD rule is satisfied. Then, `removeConnection` (see entailment (24)) removes all connection nodes involving s, and `deleteSpider` (see entailment (25)) removes s, satisfying the outer postcondition. □

Rule 4. ***[Erasure of shading.]*** Shading can be erased from any shaded zone as shown in Figure 44 for rule `eraseShading`.

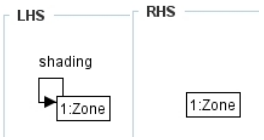


Fig. 44 Rule `eraseShading`: erasing shading from a zone.

Lemma 7 *The rule* eraseShading *realises Rule 4 of the SD system.*

Proof The rule effect clearly satisfies the SD rule specification given by:

$SD!eraseShading()$
$\triangle\ \exists z \in \mathcal{Z}^*$
$\triangledown\ z \in \mathcal{Z} \setminus \mathcal{Z}^*$

□

Rule 7. ***[Equivalence of Venn and Euler forms.]*** The final rule establishes the equivalence between SDs in Euler and Venn forms, meaning that they have underlying EDs or VDs (which have no missing zones), by showing how to transform one form into the other. We first present the TU for adding missing zones, in which missing zones can be progressively introduced as the twins of existing zones, as defined by the control condition:

AddMissingZones() =
asLongAsPossible `addMissingTwin`;
asLongAsPossible `completeInsideRelations` **end**;
`removeTwin`
end

Rule `addMissingTwin` in Figure 45 creates a new zone z_1 and marks it as twin of some existing zone z_2, outside some curve c. The zone z_1 is inside the boundary curve and c, and $twin_c(z_1, z_2)$ will hold at the end of the TU.

The GAC in Figure 46 checks that z_2 lacks a twin. The newly created z_1 must be inside all and only the curves that contain z_2, as guaranteed by rule `completeInsideRelations` in Figure 47. A NAC, not shown here, ensures that only one *inside* edge is created between a zone and a curve. The rule `removeTwin` (not shown here) is then applied to remove all *twin* edges, before proceeding with identifying and constructing the next twin.

We also consider the opposite transformation, from Venn to Euler, by which any shaded uninhabited zone can be erased (provided it is not the last zone inside any curve), realised by the TU whose control expression is:

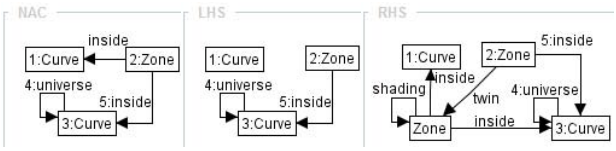


Fig. 45 Rule `addMissingTwin`.

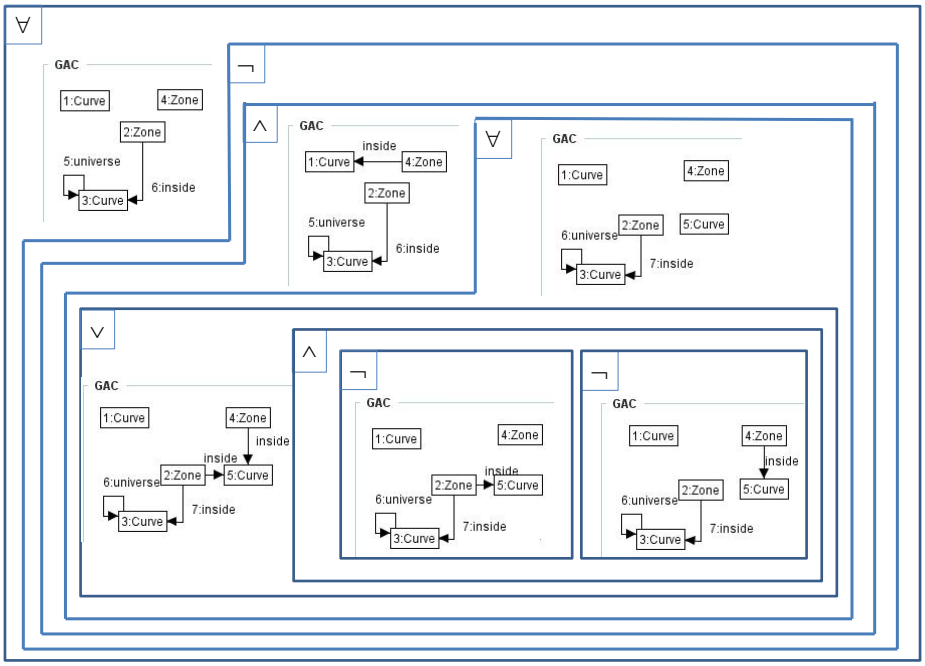


Fig. 46 A graphical representation of the GAC for identifying missing twins, to be checked on each match m for the LHS of the rule `addMissingTwin` in Figure 45. The associated formula reads: "for all matches $m_1 : g_1 \to G$ of the graph g_1 in the outer $\forall$-box extending m, it is not the case that there exists a match $m_2 : g_2 \to G$ of the graph g_2 in the outer $\wedge$-box extending m_1 such that, for each match $m_3 : g_3 \to G$ of the graph g_3 in the next $\forall$-box extending m_2, either a match extending m_3 exists for the graph in the $\vee$-box, or neither of the two graphs in the $\neg$-boxes has a match extending m_3".

EraseShadedZone() =
asLongAsPossible `markShadedZone` **end**;
asLongAsPossible (`deleteMarkedZone` | `removeMarkFromZone`) **end**;

In order to simulate a non-deterministic choice of the zones to be deleted, we first iterate on rule `markShadedZone` (see Figure 48) to mark all shaded zones satisfying the following conditions: (1) the zone is not the only one

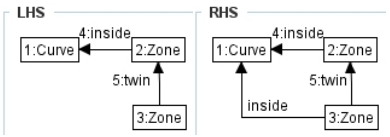


Fig. 47 Rule `completeInsideRelations`.

inside a curve (checked in the LHS of the rule and in the GAC of Figure 49, where the different ways in which a second zone can be inside a curve is presented) and (2) the zone is not inhabited by any spider (checked in the NAC). After that, an iteration is performed on the alternative between deleting a marked zone, or simply removing the mark.

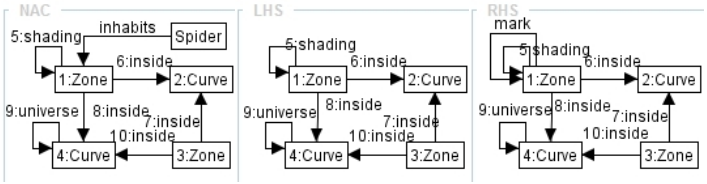


Fig. 48 Rule `markShadedZone`.

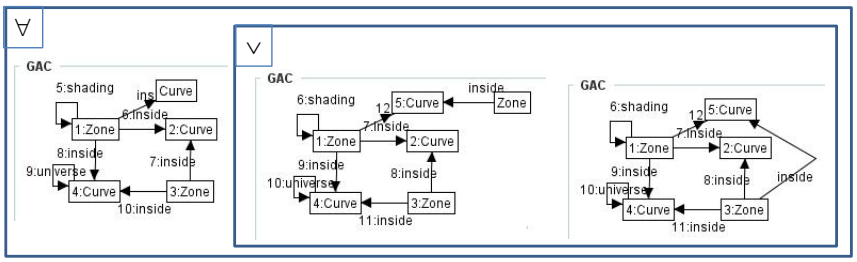


Fig. 49 A graphical representation of the GAC for checking that the zone is not the only one inside a curve, to be checked on each match m for the LHS of the `markShadedZone` rule in Figure 48. The associated formula reads: "for any match $m_1 : g_1 \to G$ of the graph g_1 in the $\forall$-box extending m, at least one of the two graphs inside the $\vee$-box has a match extending m_1."

Lemma 8 *The TUs for* AddMissingZones *and* EraseShadedZone *realise Rule 7 of the SD system.*

Proof The addition of all of the missing zones is specified as:

SD!*addMissingZones*()

$\triangle\ X \cup Y = \mathcal{C} \wedge X \cap Y = \emptyset \wedge u \in X \wedge (X, Y) \notin \mathcal{Z}$
$\triangledown\ (X, Y) \in \mathcal{Z}^*$

The rule `addMissingTwin` in the TU (where the GAC is written in square brackets) followed by the `completeInsideRelations` rule has effect:

$$\begin{aligned} & c_1 \in \mathcal{C} \wedge z_2 = (X_2 \cup \{u\}, Y_2 \cup \{c_1\}) \in \mathcal{Z} \wedge \\ \forall z_4 \in \mathcal{Z}[\neg z_4 = (X_4 \cup \{c\}, Y_4) \wedge \forall c_5 \in \mathcal{C}[z_2 = (X_5 \cup \{c_5\}, Y_5) \wedge & \\ & z_4 = (X_6 \cup \{c_5\}, Y_6) \vee (z_2 = (X_5, Y_5 \cup \{c_5\}) \wedge \\ & z_4 = (X_6, Y_6 \cup \{c_5\}))]] \vdash z_1 = (X \cup \{c_1, u\}, Y) \in \mathcal{Z}^{*\prime}. \end{aligned} \tag{26}$$

In the post-state of the SG after completing the iteration, $twins_c(z_1, z_2)$ holds. After the `addMissingTwin` rule the `Zone`-node for $z_1 = (\{u, c\}, Y)$ is not a correct twin for z_2 yet, but is related to z_2 via a twin indicator. Immediately afterwards, iterating the rule `completeInsideRelations` adds all of the inside edges from z_1 to the same set of curves as z_2, as required to ensure that it represents the required missing twin zone.

The above ensures that the addition of any missing twin zone can be achieved. It remains to show that one can just add missing twin zones to any SD in order to create the Venn form. Since any SD has an outermost zone, we can consider if there is any curve which does not have a zone which is inside only that curve (and the boundary curve). Such a zone would be a twin of the outermost zone w.r.t. that curve. The addition of any such missing zone ensures that every zone inside exactly the boundary curve and one other curve is present. Then we consider if there are any missing zones that are inside exactly two curves (plus the boundary curve), which are twins of the zones that is inside one curve (plus the boundary curve). Continuing this process ensures that all missing zones are added.

The removal of some of the shaded zones that are not inhabited by any spider (and do not contain the last zone within any curve) is specified as:

SD!*eraseShadedZones*()

$\triangle\ z = (X, Y) \in \mathcal{Z}^* \wedge X \neq \{u\} \wedge (\nexists s \in \mathcal{S} \bullet z \in h(s)) \wedge$ $(\nexists c \in \mathcal{C} \bullet c \in X \wedge (\nexists z_3 = (X_3, Y_3) \in \mathcal{Z} \bullet z_3 \neq z \wedge c \in X_3))$
$\triangle$
$\triangledown\ z \notin \mathcal{Z}$
$\triangle$
$\triangledown\ z \in \mathcal{Z}^*$

The `markShadedZone` rule is specified by entailment (27) (with the GAC indicated within the second pair of square brackets):

$$z_1 = (X_1 \cup \{u, c\}, Y_1) \in \mathcal{Z}^* \wedge z_2 = (X_2 \cup \{u, c\}, Y_2) \in \mathcal{Z} \wedge$$
$$(\neg \exists s \in \mathcal{S} \bullet z_1 \in h(s)) \wedge (\forall c_k \in \mathcal{C} \setminus \{u, c\} \bullet (z_1 = (X_1 \cup \{u, c, c_k\}, Y_1)) \wedge$$
$$((\exists z_3 = (X_3 \cup \{c_k\}, Y_3)) \vee (\exists z_2 = (X_2 \cup \{u, c, c_k\}, Y_2)))) \vdash z_1 \in M'. \tag{27}$$

This marks a set of shaded zones. During the following iteration, at each step one of `deleteMarkedZone` or `removeMarkFromZone` is selected. The effect of the two rules is described in entailments (28) and (29), respectively.

$$z \in \mathcal{Z}^* \cap M \vdash z \notin \mathcal{Z}' \wedge z \notin M'. \tag{28}$$

$$z \in \mathcal{Z}^* \cap M \vdash z \in \mathcal{Z}' \wedge z \notin M'. \tag{29}$$

After the iteration is performed, a (possibly empty) subset of $\mathcal{Z}^*$ has been removed and no zone is marked. □