VISUAL LANGUAGES AND LOGIC

VLL 07

Coeur d'Aléne, Idaho, USA

September 23rd, 2007

Editors:

PHILIP COX, ANDREW FISH AND JOHN HOWSE

Contents

Preface iii

Programme Committee iv

DAVE BARKER-PLUMMER AND NIK SWOBODA
A Sequent Based Logic for Coincidence Grids 1

BENEDEK NAGY AND SÁNDOR VÁLYI
Visual reasoning by generalized interval-values and interval temporal logic .. 13

AIDEN DELANEY AND GEM STAPLETON
Spider Diagrams of Order 27

ROBIN CLARK
Fast Zone Discrimination 41

FRITHJOF DAU AND PETER EKLUND
A Peirce Style Calculus for ALC 55

HARALD STÖRRLE
A PROLOG-based Approach to Representing and Querying Software Engineering Models 71

SACHA BERGER, FRANÇOIS BRY, TIM FURCHE AND CHRISTOPH WIESER
Visual Languages: A Matter of Style 85

CORIN GURR
Visualising a Logic of Dependability Arguments 97

Preface

This volume contains the proceedings of the First International Workshop on Visual Languages and Logic (VLL 07), held in Coeur d'Aléne, Idaho, USA, on the 23rd September 2007, as a satellite event of the 2007 IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2007).

Our goal in proposing the VLL Workshop to the VL/HCC organisers was to bring together researchers to explore the current state of research at the intersection of visual languages and logic, including topics such as: graphical notations for logics (either classical or non-classical, such as first or higher order logic, temporal logic, description logic, independence friendly logic, spatial logic); diagrammatic reasoning; theorem proving; formalisation (syntax, semantics, reasoning rules); expressiveness of visual logics; visual logic programming languages; visual specification languages; applications; and tool support for visual logics.

The eight papers presented here were each reviewed by three or four programme committee members, and provide an insight into some of the interesting combinations of logic and visualisation currently being investigated.

As anyone who has organised such a meeting knows, success depends on many people. We wish to thank the members of the Programme Committee, who, despite being given a very short time to complete their tasks, provided prompt and helpful feedback.

Thanks are also due to the VL/HCC 2007 organisers for providing the opportunity to run VLL 07, and for their logistic support, and to the Swedish Institute of Computer Science for its sponsorship. Finally, we wish to thank the VLL 07 presenters, without whom there would be no workshop.

These proceedings will be published as volume 274 in the CEUR series, published electronically and available online at http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/.

Philip Cox[1], Andrew Fish [2] and John Howse [2]
23rd September 2007

(1) Dalhousie University, Canada
(2) University of Brighton, UK

Programme Committee

- Gerry Allwein, Naval Research Laboratory, USA;
- Omid Banyasad, IBM, Canada;
- Dave Barker-Plummer, Stanford University, USA;
- Paolo Bottoni, Universita di Roma, La Sapienza, Italy;
- Frithjof Dau, University of Wollongong, Australia;
- Mateja Jamnik, University of Cambridge, UK;
- Alexander Knapp, Ludwig-Maximilians Universität, Munich, Germany;
- Bernd Meyer, Monash University, Australia;
- Nathaniel Miller, University of Northern Colorado, USA;
- Mark Minas, Universität der Bundeswehr, Munich, Germany;
- Ian Pratt-Hartman, University of Manchester, UK;
- Andy Schürr, Technische Universität Darmstadt, Germany;
- Gem Stapleton, University of Brighton, UK;
- Nik Swoboda, Universidad Politécnica de Madrid, Spain;
- Simon Thompson, University of Kent, UK.

A Sequent Based Logic for Coincidence Grids

Dave Barker-Plummer
Stanford University
Stanford, CA, 94305-4101, USA
dbp@csli.stanford.edu

Nik Swoboda
Universidad Politécnica de Madrid
Boadilla del Monte, Madrid, 28660, Spain
nswoboda@clip.dia.fi.upm.es

Abstract

Information is often represented in tabular format in everyday documents such as balance sheets, sales figures, and so on. Tables represent an interesting point in the spectrum of representation systems between pictures and sentences, since some aspect of tables are sentential or conventional in nature, while others are graphical. In this paper we describe the logic of a particular formalized tabular representation system, that of coincidence grids. Although less common than everyday tables, this system is recommended for use in the search for solution of so-called "Logic Puzzles". Such puzzles provide a specific reasoning task in service of which the tabular representation is used.

1 Introduction

Representations appear to range along a spectrum from the highly conventional sentential representations to pictorial representations which are strongly isomorphic to the objects which they represent. The diagrams used in the Hyperproof program are "pictorial" in this sense, being pictures of a checkerboard on which blocks of various sizes and shapes are placed [3]. The diagrams introduced by Euler and Venn [4, 8] and studied by Shin in [6] and Hammer in [5] have some features of pictorial representations, but lack others. For example, in Euler diagrams, closed curves are used to represent sets, and the points within such curves represent the members of the represented sets. This is not a direct pictorial representation of a set in the way that Hyperproof's blocks are, or could be, pictures of real objects.

The main body of this paper is concerned with a discussion of the logic of a particular tabular representation, one that we call coincidence grids. This representation is recommended for use in the solution of certain "logic puzzles" found some puzzle books. We have chosen to study this somewhat uncommon representation because the representation is used in service of particular reasoning problems, and these problems have clear cut structures. This representation uses graphical constraints in only a very simple way compared to representations like the Venn and Euler systems. The only graphical constraint which is exploited by the use of tabular representations is that

each cell of the table can contain exactly one value, and therefore if a value is already present in a cell then no other value can fill that role.

1.1 Outline

The methodology which we adopt is exactly that used by logicians investigating proof and model theories of the more familiar sentential logics, for example first order logic. In Section 3 we specify the syntax of the representation, and then, in Section 4 we present a model theory for the representation, that is a mapping from the representation to the world which allows us to assert that certain instances of the representation are accurate descriptions of the (real or imaginary) world. Next, in Section 5, we describe the inference rules which may be applied to instances of the representation to produce new instances. Finally, in Sections 6 and 7, we give proofs of soundness and completeness for the logic described.

2 Coincidence Grids

Coincidence grids are recommended in "logic puzzle" books as an aid to solving certain kinds of logic puzzles. Here is an example problem:

> Sylvia and four other workers in Midsville were unemployed for a short time last year when they decided to change their occupations (one was a telephone operator) and undergo retraining for new jobs. The five are now happily re-employed (one is a mechanic). From the premises below, determine each worker's first name, last name (one's is Swanson), former job, and present job.
>
> 1. The five workers are Tom, the former welder, the present arcade manager, Ms. Cortez, and the present fitness instructor (who is not Mr. Bertram).
> 2. Ralph used to be a foreman.
> 3. Marie who is neither Cortez nor Monroe, used to be a secretary.
> 4. Mr. Hampton is now a mail carrier.
> 5. The programmer, who is not Erica, used to repair TVs.

Figure 1 shows an example coincidence grid used in the solution of this puzzle. Puzzles of this type involve the attributes possessed by individuals. Each individual, typically a person, is known to have a number of attributes: first name, current occupation, etc. The set of available values for these attributes is stated in the puzzle and these values are shared among the individuals exhaustively and exclusively. A solution to the puzzle is a statement of exactly which single and unique value for all of the attributes each individual has.

The main interest in this paper is in formalizing and characterizing the reasoning that may be performed using the coincidence grid representation system. In contrast, the main interest in logic puzzles is in the extraction of the correct information for display in the initial diagram. This is an essentially heterogeneous reasoning problem, of the form discussed in [1, 2, 7] but we do not discuss this feature of logic puzzles in this paper. If one were to imagine the sentential assumptions expressed in a formal logic, perhaps as formulae whose atomic subformulae are constrained to be equalities, then the puzzles would be quite trivial to solve, indicating that the real trick in these puzzles is an appropriate understanding of the subtleties of the semantics of natural language.

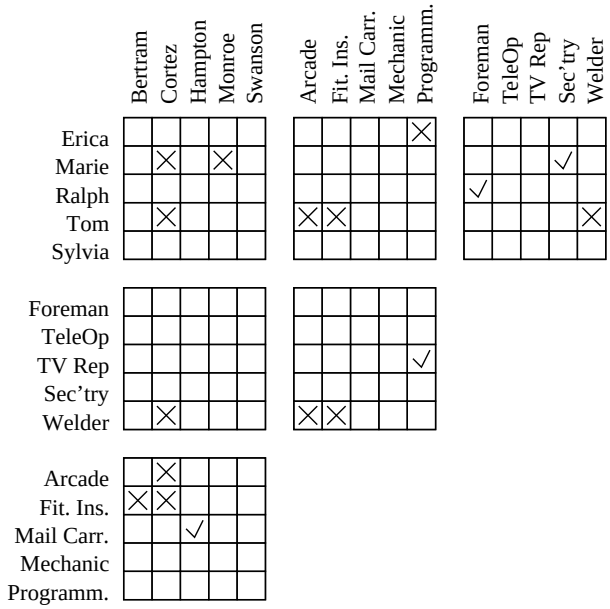


Figure 1: A coincidence grid diagram

There are two variable quantities in the coincidence grid, the number of attributes with which the problem is concerned, and the number of values each attribute may take on. Information concerning these problems comes in the form of assertions about the coincidence or non-coincidence of pairs of values for attributes. For example, the sentence "Ralph used to be the foreman" asserts that the object with first name "Ralph" also has the former occupation "foreman". The diagram of Figure 1 consists of a number of individual grids, each of which have as their axes two of the attributes mentioned in the problem. For example the rightmost grid on the top row of the diagram has as its axes the "first name" and "former occupation" attributes. The diagram has a unique square grid for each pair of distinct attributes.

If we adopt the convention that a cell marked with a ✓ which is in a particular row and column indicates that the object with the property labeling the row is the same as the object with the property labeling the column, and the same cell marked with a $\times$ means that those same properties are known not to hold of the same object, then we can use the representation to indicate concisely certain assertions about the problem. For example, the ✓ in the leftmost column of the grid representing the product of the "first name" and "former occupation" attributes, represents the assertion that "Ralph used to be a foreman" (hypothesis 2 of the example problem). While the $\times$ in that same grid represents the assertion that "Tom is not the former welder".

3 Syntax

We begin by defining the basic syntactic building blocks which will be used to construct coincidence grids:

- Grids - for all natural numbers n we will have a countably infinite number of $n \times n$ grids each consisting of n^2 *cells*. Each grid is taken to represent informa-

tion relating two attributes, each row or column represents information about an individual value, and each cell is taken to represent whether there is some object which has as values both those represented by that cell's row and column.

- Labels - a countably infinite number of labels (collected into the set $\mathcal{L}$) used to give names to the rows and columns of each grid.

- Marks - Cells of grids can be marked with either ✓ or $\times$. ✓ will be used in a cell when a single object is taken to have the pair of values of the cell's row and column labels, and $\times$ will be used when the object does not have that value pair.

When referring to the rows and columns of a grid we will rely on the common understanding of these notions. For convenience we will make reference to cells of a grid using pairs of rows and columns, for this we will use a square bracket notation, e.g., $[r, c]$. When using this notation the order of r and c is irrelevant, i.e., $[r, c] = [c, r]$.

Definition 3.1 (Labeled Grid)
*A **labeled grid** l of size n is $\langle g_l, rows_l, cols_l \rangle$ where g_l is a $n \times n$ grid and $rows_l$, $cols_l$ are one-to-one functions from the rows (resp. columns) of g_l to disjoint subsets of $\mathcal{L}$. We will refer to the ranges of $rows_l$ and $cols_l$ as **label sets**.*

For convenience we define $labels_l = rows_l \cup cols_l$[1]. Using the function $labels_l$ we can derive the partial function $cellFor_l$ from pairs of cell labels to cells (with cells represented by the intersection of a column and a row), $cellFor_l(i, j) = [a, b]$, when $labels_l(a) = i$ and $labels_l(b) = j$.[2] Finally we will sometimes refer to the cells of a labeled grid using the pairs of their column and row labels l, m, e.g., (l, m) where as before the order is irrelevant, i.e., $(l, m) = (m, l)$.

Definition 3.2 (Grid Layout)
*A **grid layout** is a compatible collection of labeled grids. A collection of labeled grids, $l_1, \ldots l_k$ all of size m, is said to be compatible when:*

- *No two labeled grids in the layout have the same set of labels (for each $i \neq j$, $labels_{l_i} \neq labels_{l_j}$). Note that this condition also excludes the inclusion of two labeled grids where the columns and rows are swapped.*

- *Row and column labels travel in packs, i.e., it isn't possible for the same label to appear in more than one distinct label set (for each i, j, $rows_{l_i}$ and $cols_{l_i}$ are each either equal to or disjoint from each of $rows_{l_j}$ and $cols_{l_j}$).*

- *Every pair of label sets from some labeled grid is represented by some labeled grid in the layout (for each distinct pair $rows_{l_i}$, $cols_{l_j}$ there is a grid l with $labels_g = rowsl_i \cup cols_{l_j}$)).*

We call a grid layout with n distinct row label collections, each of m labels, an n, m-grid layout.

When drawing a grid layout, we observe the convention that all grids sharing the same row label collections are drawn in series from right to left and with their row

[1] Here and where convenient we will view functions as a sets of ordered pairs, and subject them to set operations.

[2] Here and throughout the remainder of the paper subscripts will be omitted when they can be unambiguously inferred from the context.

labels in the same order, and that all grids sharing the same set of column labels are drawn top to bottom with their columns in the same order. Thus the labels for the rows and columns of the grid layout can be placed along the top and left edges of the grid layout labeling rows and columns which span multiple grids.

The grid layout defines the tabular structure in which information may be represented. Information is represented by placing values into this structure. This is modeled using a marking function.

Definition 3.3 (Marking function)
*A **marking function**, M, for the grid layout G is a (possibly partial) function from pairs of labels of the cells of G to the set $\{\checkmark, \times\}$. Given a cell in a row labeled r and column labeled s, $M(r,s) = \checkmark$ and $M(r,s) = \times$ are taken to mean that the referenced cell is marked with the corresponding symbol. $M(r,s)$ is undefined when a cell is unmarked. We will say that a marking function is **total** when it assigns either $\checkmark$ or $\times$ to all pairs of labels of cells in the grid layout.*

Definition 3.4 (Coincidence grid)
*A **coincidence grid**, (G, M), is a grid layout G along with a marking function M for that grid layout.*

4 Semantics

Coincidence grids are used to reason about information regarding the values for attributes of a set of objects. Coincidence Structures are mathematical objects which model this kind of information and thereby are used to give meaning to coincidence grids.

Definition 4.1 (Coincidence Structure)
*An n, m**-coincidence structure** is a tuple $\langle Partition, denotedBy \rangle$ where $Partition$ is a collection of n disjoint finite sets each with m elements and $denotedBy$ is a one-to-one function from members of sets in $Partition$ to labels in $\mathcal{L}$. We call the union of the sets in $Partition$ the* values *of the structure.*

If A is an coincidence structure, the relation $coincides_A$ is defined so that $coincides_A(a, b)$ is true just when a and b are labels in $\mathcal{L}$ and there are members j, k of the same set in $Partition_A$ such that $denotedBy_A(j) = a$ and $denotedBy_A(k) = b$. It is trivial to show that $coincides_A$ is an equivalence relation.

Definition 4.2 ($\models$)
Given a n, m-coincidence structure A and the coincidence grid (G, M) with G a n, m-grid layout, we write $A \models (G, M)$, and say that A is a model of (G, M) iff

- *for each labeled grid g in G there are not two row labels l, l' such that $coincides(l, l')$ nor two column labels m, m' such that $coincides(m, m')$.*
- *for each cell in g in a row labeled l and a column labeled m marked with $\checkmark$, $coincides(l, m)$ is true.*
- *for each cell in g in a row labeled i and a column labeled j marked with $\times$, $coincides(l, m)$ is false.*

Definition 4.3 ($\mathcal{C} \models \mathcal{C}'$)
*A coincidence grid $\mathcal{C}'$ is a **consequence** of a coincidence grid $\mathcal{C}$, $\mathcal{C} \models \mathcal{C}'$, if every model of $\mathcal{C}$ is a model of $\mathcal{C}'$.*

5 Proof System

The inference rules of the proof theory are given in sequent style below. When we reason with coincidence grids, the grid layout is fixed: inference proceeds by modifying the marking function, by adding or removing values at individual cells.

5.1 Erasure

The rule of ERASURE allows us to remove zero or more marks from a diagram.

$$\frac{}{(G, M) \rightsquigarrow (G, N)}$$

Proviso: $N \subseteq M$

Figure 2: Rule: ERASURE

The ERASURE rule does not require N to be a proper subset of M. When N and M are identical, we draw attention to this fact by referring to the rule as REITERATION.

5.2 Cases

Definition 5.1 (Extension of M at a cell)
Let M be a marking function which is undefined at (l, m).

- $M^{\checkmark}_{(l,m)} = M \cup \{((l, m), \checkmark)\}$, *i.e.* $M^{\checkmark}_{(l,m)}$ *is just like* M *except that* $M^{\checkmark}_{(l,m)}$ *assigns* $\checkmark$ *to* (l, m).
- $M^{\times}_{(l,m)} = M \cup \{((l, m), \times)\}$, *i.e.* $M^{\times}_{(l,m)}$ *is just like* M *except that* $M^{\times}_{(l,m)}$ *assigns* $\times$ *to* (l, m).

The CASES rule allows us to examine the exclusive cases generated by extending a marking function at a single cell in both possible ways.

$$\frac{(G, M^{\checkmark}_{(l,m)}) \rightsquigarrow (G, N) \qquad (G, M^{\times}_{(l,m)}) \rightsquigarrow (G, N)}{(G, M) \rightsquigarrow (G, N)}$$

Figure 3: Rule: CASES

5.3 Contradiction

We now need a crucial definition which defines the pathways that allow information to flow between grids. Intuitively, a cell in grid relates two values (a, b) to one another (perhaps by containing a check). There is a cell in another grid relating the one of these values to a third value (a, c) say. Yet a third cell relates (b, c), and the values on these

three cells are dependent on one another (in particular, it isn't consistent for exactly two of them to contain a check and the other a cross.) We identify these cell collections as *triads*.

Definition 5.2 (Triad)
*Given a coincidence grid $\mathcal{C}$, a **triad** is three cells of $\mathcal{C}$ related by a chain of attribute pairs. Three cells $[x_1, y_1], [x_2, y_2], [x_3, y_3]$ are related by a chain of attribute pairs when there are labels t_1, t_2, t_3 in three labeled grids a, b, c in $\mathcal{C}$, such that $cellFor_a(t_1, t_2) = [x_1, y_1]$, $cellFor_b(t_2, t_3) = [x_2, y_2]$, and $cellFor_c(t_1, t_3) = [x_3, y_3]$.*

In Figure 1, an example of a triad would be the cells (TV Rep, Programm.), (Programm., Erica), (TV Rep, Erica). This example also shows the importance of triads, i.e., since we know that the former TV repair-person is currently employed as a programmer, and we know that Erica isn't currently a programmer we can conclude that Erica wasn't formerly employed as a TV repair-person. With the notion of triad in hand, we can now define a contradictory marking function for a diagram, which we will later show means that the diagram cannot represent a solution to the problem.

Definition 5.3 (Contradictory Marking Function)
*A marking function M is **contradictory** if any of the following conditions hold:*

1. *M assigns two checks in the same row or column of a grid ($M(l_0, l_1) = \checkmark = M(l_0, l_2)$ for any l_0 from one label set and with l_1 and l_2 distinct values from a second label set).*
2. *M assigns one row or column of a grid to be completely filled by crosses ($M(l_0, l) = \times$ for any l_0 and for all l drawn from a second label set.)*
3. *M assigns a triad two checks and a cross ($M(l_0, l_1) = \checkmark = M(l_0, l_2)$ and $M(l_1, l_2) = \times$).*

$$\frac{}{(G, M) \rightsquigarrow (G, N)}$$

Proviso: M is contradictory

Figure 4: Rule: CONTRADICTION

Definition 5.4 ($\vdash$)
For any coincidence grids (G, M) and (G, M'), we say that (G, M') can be proved from (G, M), written $(G, M) \vdash (G, M')$, iff there is a tree of applications of the above rules of inference whose root contains $(G, M) \rightsquigarrow (G, M')$, and whose leaves are all closed (derived from no premises using ERASURE *or* CONTRADICTION*).*

A simple example proof is shown in Figure 5. Before leaving this discussion of the proof theory, we prove the following important result, which we will use later:

Proposition 5.1 *For any sequent $(G, M) \rightsquigarrow (G, N)$, we can build a proof tree whose leaves contain all of the sequents $(G, M_i) \rightsquigarrow (G, N)$, where M_i is a total extension of M.*
Proof *The proof is by induction on the number of places that M is undefined.*

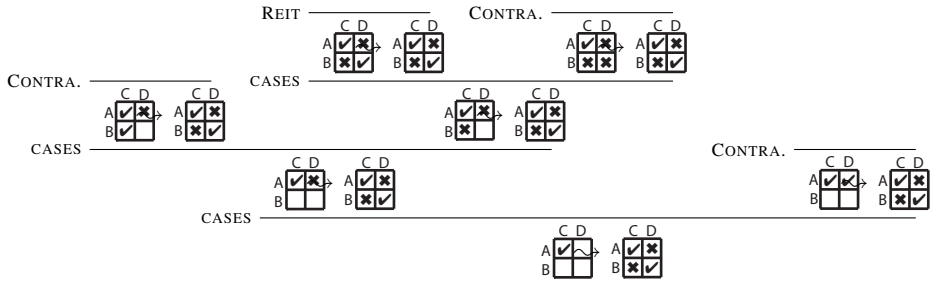


Figure 5: An Example Proof

Basis: *If M is total, then we can build a proof with $(G, M) \rightsquigarrow (G, N)$ as the only leaf of a tree consisting of one application of the* REITERATION *rule.*

Step: *Assume that the result follows for all marking functions undefined at $i < k$ cells, we will show that the result follows for any marking function undefined at k cells.*

A single application of CASES *to the given sequent yields the following tree:*

$$\text{CASES}\ \frac{(G, M^{\checkmark}_{(l,m)}) \rightsquigarrow (G, N) \qquad (G, M^{\times}_{(l,m)}) \rightsquigarrow (G, N)}{(G, M) \rightsquigarrow (G, N)}$$

By the induction hypothesis, we can build a proof tree with the required property above each of the new sequents:

$$\text{CASES}\ \frac{\dfrac{\Pi}{(G, M^{\checkmark}_{(l,m)}) \rightsquigarrow (G, N)} \qquad \dfrac{\Sigma}{(G, M^{\times}_{(l,m)}) \rightsquigarrow (G, N)}}{(G, M) \rightsquigarrow (G, N)}$$

The proof Π has at it leaves all of the sequents which are total extensions of $M^{\checkmark}_{(l,m)}$, and Σ's leaves are the total extensions of $M^{\times}_{(l,m)}$, so together the leaves have all total extensions of (G, M) on their left hand sides (and (G, N) on the right).

6 Soundness

Theorem 6.1 (Soundness) *For any coincidence grids (G, M) and (G, M'), if $(G, M) \vdash (G, M')$ then $(G, M) \models (G, M')$*

Proof *The proof is by induction on the height of the proof tree.*

Basis *If the height of the tree is 1, then the proof must involve one application of* CONTRADICTION *or* ERASURE.

- **ERASURE:** *Suppose for the sake of contradiction that $(G, M) \not\models (G, M')$. There exists some model, A such that $A \models (G, M)$ and $A \not\models (G, M')$. Since both coincidence grid contain the same labeled grid G we know that $A \not\models (G, M')$ can not be due an incompatibility between label sets in G and the $coincides$ relation. Thus there is some cell (l, m) such that it is marked with a check in M' and $coincides(l, m)$ is false, or it is marked with a cross in M' and $coincides(l, m)$ is true. In either case, this mark was present in M, and A therefore fails to be a model of (G, M). Contradiction.*

- **Contradiction:** *It suffices to show that any coincidence grid to which* CONTRADICTION *can be applied has no models.*

  Suppose for sake of contradiction that (G, M) is an coincidence grid of size n to which CONTRADICTION *applies, and that $A \models (G, M)$. There are three cases depending on which of the clauses of* CONTRADICTION *applied.*

  1. *Some grid of (G, M) contains a row labeled l with two check marks, in the columns labeled m and m' say. This means that both $coincides(l, m)$ and $coincides(l, m')$ are true, and hence that $coincides(m, m')$ is also true, which means that $A \not\models (G, M)$, contradiction. The case of two checks in the same column is analogous.*
  2. *Some grid of (G, M) contains a row $[l, m_1], \ldots, [l, m_n]$ with all crosses. This means that no row label m_i is mapped to the same member of A as l is. But this is impossible since there are n members of A, n of the m_i, and each m_i is mapped to a different member of A. Contradiction. The case of a grid with a column with all crosses is analogous.*
  3. *Some triad in (G, M) contains two checks and a cross, i.e., $M(l_0, l_1) = \checkmark = M(l_0, l_2)$ and $M(l_1, l_2) = \times$. Since $M(l_0, l_1) = \checkmark = M(l_0, l_2)$, $coincide(l_0, l_1)$ and $coincide(l_0, l_2)$ are derived from A. Since $coincide$ is transitive, it follows that $coincide(l_1, l_2)$. But since $A \models (G, M)$ and $M(l_1, l_2) = \times$, this cannot be true. Contradiction.*

Step *Suppose that the height of the proof tree is $k > 1$ and that the result holds for all proofs of length less than k.*

The inference at the root must be an application of CASES *on some cell with labels l and m. We must show that $(G, M) \models (G, M')$. Suppose not, i.e. that there is some model A of (G, M) which is not a model of (G, M'). In every model $coincides(l, m)$ is either true or false, and so either A is a model of $(G, M^{\checkmark}_{(l,m)})$ or of $(G, M^{\times}_{(l,m)})$. But we know that $(G, M^{\checkmark}_{(l,m)}) \models (G, M')$ and that $(G, M^{\times}_{(l,m)}) \models (G, M')$ from the induction hypothesis, which is a contradiction.*

7 Completeness

In this section we will demonstrate the completeness of the proof system, i.e., that it is sufficiently powerful to allow that any logical consequence of a coincidence grid can be proved to be so. To show this result, we will present a generic strategy (Algorithm 7.1) for building a proof with any coincidence grid $\mathcal{C}$ as the premise and any coincidence grid $\mathcal{C}'$ as the conclusion. Furthermore we will show that if this strategy fails then that $\mathcal{C}'$ can not be a logical consequence of $\mathcal{C}$.

The strategy proceeds as follows. First, we build a proof tree with $\mathcal{C} \rightsquigarrow \mathcal{C}'$ as the root and a leaf $\mathcal{C}_i \rightsquigarrow \mathcal{C}'$ for each total extension of $\mathcal{C}$. If there is any leaf which can not be closed, established using either the ERASURE or CONTRADICTION rules with no premise then the strategy fails.

Algorithm 7.1 (Canonical Proof Strategy)
We begin with any two coincidence grids (G, M) (the premise) and (G, M') (the conclusion).
Step #1: *Start the proof with the sequent $(G, M) \rightsquigarrow (G, M')$ as the root.*

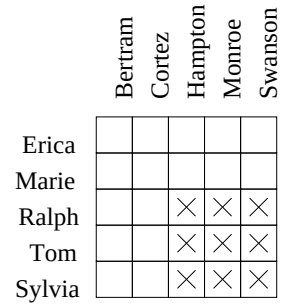


Figure 6: A coincidence grid with no models to which CONTRADICTION can not be applied

Step #2: *While there exist unmarked cells in the left hand side of the sequent of some leaf of the proof: select at random any unmarked cell* (l, m) *in* (G, M'') *of each leaf* $(G, M'') \rightsquigarrow (G, M')$ *and apply the* CASES *rule to add the branches* $(G, M''^{\checkmark}_{(l,m)}) \rightsquigarrow (G, M')$ *and* $(G, M''^{\times}_{(l,m)}) \rightsquigarrow (G, M')$ *to the proof above that leaf.*
Step #3: *Close each leaf with an application of the* ERASURE *or the* CONTRADICTION *rule with no premise. If this is not possible for any leaf then fail, otherwise the proof is complete.*

To prove that this strategy is correct (that any logical consequence of a diagram can be proved in this manner) we first consider the case that $\mathcal{C}$ has no models, and show that if a diagram has no models then the contradiction rule can be applied to every total extension of $\mathcal{C}$ (Proposition 7.2). Then we need to consider the case where the proof has a leaf $\mathcal{C}_i \rightsquigarrow \mathcal{C}'$ which can not be established using the using either the ERASURE or CONTRADICTION rules. First we show that any total non-contradictory coincidence grid is true in some model (Proposition 7.1). Then we know that $\mathcal{C}_i$ must have a model, and that $\mathcal{C}_i$ and $\mathcal{C}'$ differ on some cell. Using this diagram we build a model A such that $A \models \mathcal{C}$ but that $A \not\models \mathcal{C}'$ (Proposition 7.4), which means that it can't be the case that $\mathcal{C} \models \mathcal{C}'$.

We begin by observing that there are some coincidence grids which have no models, but to which CONTRADICTION can not be applied. An example of one such coincidence grid can be found in Figure 6. We need to show that every total extension of such a marking function does permit the application of CONTRADICTION.

Proposition 7.1 *All coincidence grid* (G, M) *with* G *a total marking function which is not contradictory are true in a model.*
Proof *We construct a model of* A *of* (G, M)*. For each label* l *in* G*, determine* $\{m \mid M(l, m) = \checkmark\} \cup \{l\}$*. There is exactly one* m *along each dimension that has this property, since* CONTRADICTION *does not apply to* (G, M)*. Let* $Properties$ *be the collection of these sets. (In other words, we use the labels themselves as the values of* $\langle Properties, denotedBy\rangle$*.) As the function* $denotedBy$ *we use the identity function on the labels of* G*.* $\langle Properties, denotedBy\rangle$ *is a model of* (G, M)*.*

Proposition 7.2 *If* (G, M) *has no models, then* CONTRADICTION *can be applied to every total extension of* (G, M)*.*
Proof *We prove the contrapositive. Let* T *be a total extension of* M *to which* CON-

TRADICTION *does not apply. Using Proposition 7.1, we know that* (G, T) *has a model, and since* T *is an extension of* M *we know that this model is also a model of* (G, M).

Proposition 7.3 *If* (G, M) *has no models, then* $(G, M) \vdash (G, M')$ *for any marking function* M'.
Proof *By Proposition 5.1 there is a proof tree* Π *whose root contains* $(G, M) \rightsquigarrow (G, M'))$ *and whose leaves contain the sequents* $(G, M_i) \rightsquigarrow (G, M')$ *for all total extensions,* M_i *of* M. *By Proposition 7.2, the contradiction rule can be applied to each leaf of this tree, and so* $(G, M) \vdash (G, M')$.

Proposition 7.4 *Given coincidence grids* (G, M) *which has a model and* (G, M'), *such that* $M(l, m) = \checkmark$ *and* $M'(l, m) = \times$ *for some* l, m, *then no model* A *such that* $A \models (G, M)$, *can be a model of* (G, M').
Proof *Since we know that in* (G, M), $M(l, m) = \checkmark$ *we also have that* $coincides_A(l, m)$ *can be derived from all models* A *in which* (G, M) *is true. However in any model* A' *which makes* (G, M') *true,* $coincides_{A'}(l, m)$ *can not hold (since* $M'(l, m) = \times$*), so there can be no model* A *such that* $A \models (G, M)$ *and* $A \models (G, M')$.

Lemma 7.1 *For all coincidence grid* (G, M) *and* (G, M'), *Algorithm 7.1 finds a proof* $(G, M) \vdash (G, M')$ *iff* $(G, M) \models (G, M')$.
Proof *First we consider the possibility that* (G, M) *has no models. In this case all coincidence grid are logical consequences of* (G, M) *so the algorithm should generate a valid proof for any coincidence grid* (G, M') *(see Proposition 7.3). Using Proposition 7.2 we know that the* CONTRADICTION *rule can be applied to each leaf generated by the algorithm and thus that a valid proof is generated.*

Now we consider the case that (G, M) *has a model. If the algorithm generates a proof then from soundness we know that* $(G, M) \models (G, M')$. *If the algorithm fails then we need to show that* $(G, M) \not\models (G, M')$. *Assuming that the algorithm fails we know that there was some leaf* $(G, M_i) \rightsquigarrow (G, M')$ *generated by the strategy which could not be closed, established by the* ERASURE *or* CONTRADICTION *rules without a premise. From the fact that the* CONTRADICTION *rule can not be applied to that sequent and Proposition 7.1 we know that* (G, M_i) *has some model which we will call* A. *Furthermore from the fact that the* ERASURE *rule can not be applied and that* M_i *is total, we know that* M_i *and* M' *disagree on the content of some cell. Thus from Proposition 7.4 we know that no model of* (G, M_i) *can be a model of* (G, M'), *i.e.,* $A \not\models (G, M')$. *Since* (G, M_i) *is an extension of* (G, M) *we know that* $A \models (G, M)$ *and finally that* $(G, M) \not\models (G, M')$.

Theorem 7.1 (Completeness)
For any two coincidence grids (G, M) *and* (G, M'), *if* $(G, M) \models (G, M')$ *then* $(G, M) \vdash (G, M')$.
Proof *The proof of this theorem is a direct result of Lemma 7.1.*

Corollary 7.1 *For any two coincidence grid* $\mathcal{C}, \mathcal{C}'$ *the question of whether* $\mathcal{C}'$ *is a logical consequence of* $\mathcal{C}$ *is decidable.*
Proof *Since there are a finite number of unmarked cells in any coincidence grid we know that Algorithm 7.1 will terminate (though possibly in exponential time), the rest is a direct result of Lemma 7.1.*

8 Conclusion

In this paper we have formalized a logic of coincidence grids, by defining the syntax proof theory and semantics for the representation. We have shown that the proof theory is both sound and complete for the semantics.

The logic that we have defined permits a search for proofs based on a "generate and test" method. The CASES rule allows us to mark a previously unmarked cell and then to determine the consequences of each possible mark. While having the twin virtues of soundness and completeness, this is not a natural logic for proof search using these representations. There are more intuitive rules which allow inference between diagrams, for example a rule which allows the addition of a $\times$ to any unmarked cell in a row or column that already contains a $\checkmark$. This rule, and others like it, are easily definable using the inference rules presented here however we have not yet found a way to define a collection of rules which allow us to dispense with CASES entirely. This is due to the fact that we have yet to find a natural contradiction rule that can be applied to recognize all diagrams that have no models. Finding that stronger version of contradiction and a set of natural inference rules which would allow us to dispense with CASES is the subject of future work.

References

[1] Dave Barker-Plummer and John Etchemendy. Visual decision making: A computational architecture for heterogeneous reasoning. In Borris Kovalerchuk and James Schwing, editors, *Visual and Spatial Analysis: Advances in Data Mining, Reasoning and Problem Solving*, pages 79–109. Springer, 2004.

[2] Dave Barker-Plummer and John Etchemendy. A computational architecture for heterogeneous reasoning. *Journal of Theoretical and Experimental Artificial Intelligence*, to appear.

[3] Jon Barwise and John Etchemendy. *Hyperproof*. CSLI Press, 1994. (ISBN: 1-881526-11-9).

[4] Leonhard Euler. *Lettres à une Princesse d'Allemagne sur Divers Sujets de Physique et de Philosophie*. Imprimerie de l'Academie Impériale des Sciences, 1768–1772. Letters 102–108.

[5] Eric Hammer. *Logic and Visual Information*. Number 3 in Studies in Logic, Language and Information. CSLI Publications, 1995.

[6] Sun-Joo Shin. A Situation-Theoretic Account of Valid Reasoning with Venn Diagrams. In J. Barwise, J.M. Gawron, G. Plotkin, and S. Tutiya, editors, *Situation Theory and its Applications*, number 26 in CSLI Lecture Notes, pages 581–605. CSLI Press, 1991.

[7] Nik Swoboda and Gerard Allwein. Modeling heterogeneous systems. In Mary Hegarty, Bernd Meyer, and N. Hari Narayanan, editors, *Diagrammatic Representation and Inference*, number 2317 in Lecture Notes in Artificial Intelligence, pages 131–145. Springer-Verlag, Berlin, 2002.

[8] John Venn. *Symbolic Logic*. Burt Franklin, New York, 1971.

Visual reasoning by generalized interval-values and interval temporal logic

Benedek Nagy[1] and Sándor Vályi[2]
1: Department of Computer Science
Faculty of Informatics
University of Debrecen
Debrecen, Hungary
nbenedek@inf.unideb.hu
2: Department of Health Informatics
Faculty of Health
University of Debrecen
Nyíregyháza, Hungary
valyis@de-efk.hu

Abstract

Interval-valued computation is an unconventional computing paradigm. It is an idealization of classical 16-, 32-, 64- etc. bit based computations. It represents data as specific subsets of the unit interval – in this sense this paradigm is classified into the continuous space machine paradigm near to optical computing. In this paper we show the visual reasoning power of interval-valued computations, namely, we demonstrate that the decision process of quantified propositional formulae is fully representable in a natural visual form. Further, we give a temporal-logical interpretation of interval-valued computations.

Keywords: new computing paradigms, visual reasoning, interval temporal logic

1 Introduction

In the last fifteen years a new direction of computing has emerged which develops ideas for computing devices motivated by nature. It includes DNA computing, quantum computing and relativistic computers, among others.

In [11] another new computing paradigm was introduced, the so-called interval-valued computation system. In this paradigm, data is represented by specific subsets of the unit interval, namely, by finite unions of disjoint subintervals. This data representation corresponds to the notion of generalized intervals ([2], [7]). In these papers some logics of temporal relations between such generalized intervals (that we call *interval-values* in this paper) are analyzed. In [11] some other operators were proposed to construct an interval-valued computing system and also the SAT problem was solved by a linear interval-valued computation. In [12] and [13] it was proved that a restricted class of interval-valued computations is adequate for PSPACE, that is, the class of languages decidable by this class of interval-valued computations coincides with PSPACE.

Reasoning by diagrams and intervals is an important area of visual representations of mathematical and logical reasoning. For example, Venn- and Euler-diagrams are well known, such as graphical versions of interval temporal logic ([4], [8]). An old method for visualizing Boolean algebraic calculations is the method of Venn diagrams. It is suitable to formulæ built from two or three propositional variables. There are good ideas to generalize Venn diagrams to a higher number of variables ([1], [3], [5], [6], [10], and [14]). Venn-diagrams are suitable to visually represent propositional logical laws. Of course, our interval-values are also able to represent propositional reasoning in a nice and natural visual form, because the interval-values form a Boolean algebra in which every finite Boolean algebra is visually representable. Moreover this visual representation is also suitable to help visually to follow the decision process of the validity of quantified propositional formulae. This problem is PSPACE-complete. This complexity class includes such typical problems that solution of two-player games like chess or go. In this paper we demonstrate the visual expressibility of the decision process of validity of quantified propositional formulae. We also formalize an interval temporal logic equipped by some modalities concerning the operators on interval-values. A decidability and a complexity result will also be given.

2 The idea of interval-valued computations

In [11] Nagy proposed a new discrete time / continuous space computational model, the so-called interval-valued computing. A precise description of the model can be found in [13], that we will recall and use. It involves another type of idealization than Turing machines – the density of the memory can be raised unlimitedly instead of its length. It is a natural model that can formulate computations of computers with higher and higher bit number in a byte in a unified framework.

As long as the paradigm keeps using only finite unions of intervals, the system fits within the bounds of classical Neumann-Church-Turing type computations.

The computation works on specific subsets of the interval $[0, 1)$, more specifically, on finite unions of [)-type subintervals. In a nutshell, interval-valued computations start with $\left[0, \frac{1}{2}\right)$ and continue with a finite sequence of operator applications. It works sequentially in a deterministic manner.

The allowed operations are motivated by the operations of the traditional computers on bit sequences: Boolean operations, shift operations and an extra operator, the product. The role of the introduced product is connecting interval-values on different 'resolution levels'. Essentially, it has the same function like magnification operators in optical computing ([15]) which is another continuous space computing paradigm where data is represented by 2-dimensional complex-valued images.

In the interval-valued computing system, an important restriction is eliminated, i.e. there is no permanent limit on the number of bits in a data unit (byte); we have to suppose only that the number is always finite. Of course, in the case of a given computation an upper bound (the bit height of the computation sequence) always exists, and it gives the maximum number of bits the system needs for that computation process. Hence our model still fits into the framework of the Church-Turing paradigm, but it faces different complexity bounds than the classical Turing model. Although the computation in this model is sequential, the inner parallelism is extended. One can consider the system without restriction on the size of the information coded in an information unit (interval-value). It allows to increase the size of the alphabet unlimitedly in a computation. In this article we employ this inner parallelism to extend the visual expressiveness

of calculations with interval-values. Complex manipulations on the interval-bytes can be shown, acting uniformly to the whole stored data – the interval-value. This makes possible, for instance, the visual representation of the decision process of quantified propositional formulae.

3 Interval-values and operators

As we mentioned, interval-values are finite unions of disjoint left-closed, right-open subintervals of the unit interval $[0, 1)$.

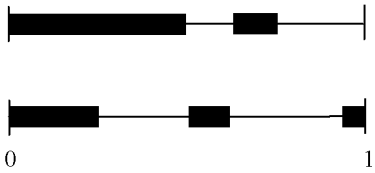


Figure 1: Examples of visual presentations of interval-values

Formally these values are defined in the following way.

Definition 1 *The set* $\mathbb{V}$ *of* interval-values *coincides with the set of finite unions of* $[)$*-type subintervals of* $[0, 1)$*. The set* $\mathbb{V}_0$ *of* specific interval-values *coincides with* $\left\{ \bigcup_{i=1}^{k} \left[\frac{l_i}{2^m}, \frac{1+l_i}{2^m}\right) \ : m \in \mathbb{N}, k \leq 2^m, 0 \leq l_1 < \ldots < l_k < 2^m \right\}$.

We note that the set of finite unions includes the empty set $(k = 0)$, that is, $\emptyset$ is also an allowed interval-value.

Similarly to traditional computers working on bytes, we allow bitwise Boolean operations. If we consider interval-values as subsets of [0,1) then the corresponding operations coincide with the set-theoretical operations of complementation ($\overline{A}$), union ($A \cup B$) and intersection ($A \cap B$). $\mathbb{V}$ forms an infinite Boolean set algebra with these operations. $\mathbb{V}_0$ is an infinite subalgebra of the last algebra. Instead of set theoretical operators we also can use the appropriate Boolean logical operators (negation, disjunction, conjunction). We note that other usual Boolean operators, as xor ($A \oplus B$) or implication ($A \rightarrow B$) are definable in the usual way.

Assisting formulation of the remaining operations, a function $Flength : \mathbb{V} \rightarrow \mathbb{R}$ is going to be defined. Intuitively, it determines the length of the first (starting) "component" of the input interval-value, that is, the first (from left) maximal subinterval of the unit interval included in the given interval-value.

Definition 2 *Let* A *be an interval-value. Let the function* $Flength : \mathbb{V} \rightarrow \mathbb{R}$ *be defined as follows. If there exist* $a, b \in [0, 1]$ *satisfying* $[a, b) \subseteq A$*,* $[0, a) \cap A = \emptyset$ *and* $[a, b') \not\subseteq A$ *for all* $b' \in (b, 1]$*, then* $Flength(A) = b - a$*, otherwise* $Flength(A) = 0$*.*

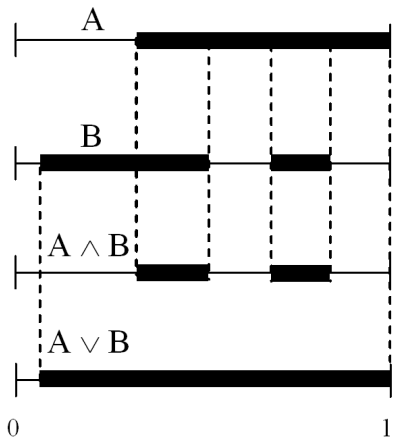


Figure 2: Examples for $\cap$ and $\cup$

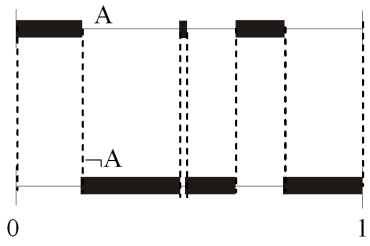


Figure 3: Example for complement

$Flength$ helps us to define the binary shift operators on $\mathbb{V}$. The *left-shift* operator will shift the first interval-value to the left by the first-length of the second operand and remove the part which is shifted out of the interval $[0,1)$. As opposed to this, the *right-shift* operator is defined in a circular way, i.e. the parts shifted above 1 will appear at the lower end of $[0,1)$. In this definition we write interval-values in their "characteristic function" notation instead of subset notation.

Definition 3 *The binary operators $Lshift$ and $Rshift$ on $\mathbb{V}$ are defined in the following way. If $x \in [0,1]$ and $A, B \in \mathbb{V}$ then*

$$Lshift(A,B)(x) = \begin{cases} A(x + Flength(B)), & \text{if } 0 \leq x + Flength(B) \leq 1, \\ 0 \text{ in other cases.} \end{cases}$$

$$Rshift(A,B)(x) = \begin{cases} A(frac(x - Flength(B))), & \text{if } x < 1, \\ 0 \text{ if } x = 1. \end{cases}$$

Here the function frac gives the fractional part of a real number, i.e., frac$(x) = x - \lfloor x \rfloor$, where $\lfloor x \rfloor$ is the greatest integer which is not greater than x.

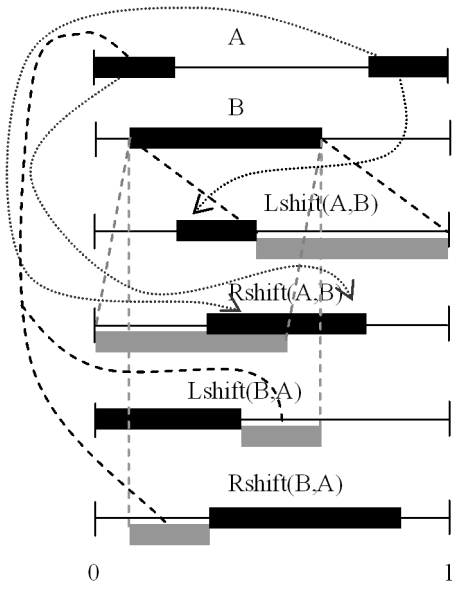


Figure 4: Examples of shift operators with interval-values

In Figure 4 some examples can be seen for both operations $Rshift$ and $Lshift$. The second operands are shown in grey but they are not the real parts of the resulting interval-values. Notice that using both shift operators in a combined way one can delete any desired parts/components of an interval-value.
Now we define the so-called *fractalian product* on interval-values.

Definition 4 *Let A and B be interval-values and $x \in [0,1)$. Then the fractalian product $A * B$ includes x if and only if $B(x) = 1$ and $A\left(\frac{x - \underline{B}_x}{\overline{B}_x - \underline{B}_x}\right) = 1$, where $\underline{B}_x$ denotes*

the lower end-point of the B-component including x and $\overline{B}_x$ denotes the upper end-point of this component, that is, $[\underline{B}_x, \overline{B}_x)$ is the maximal subinterval of B containing x.

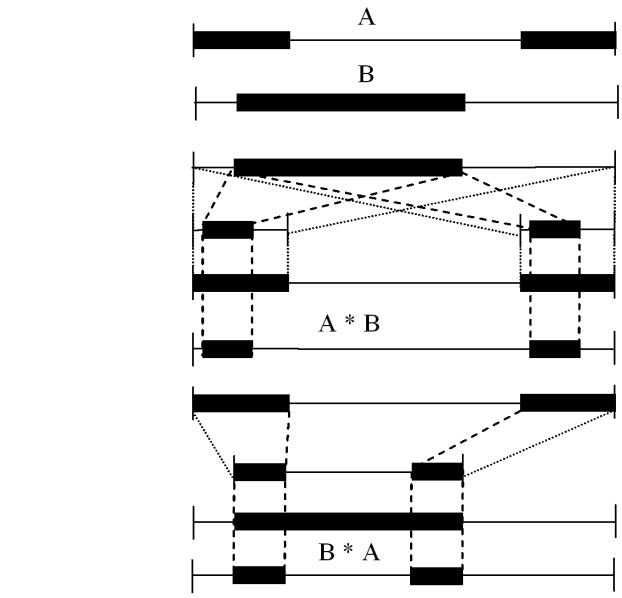


Figure 5: Examples for product of interval-values

We can explain this in a more descriptive manner. If A contains exactly k interval components with ends $a_{i,1}, a_{i,2}$ $(1 \leq i \leq k)$ and B contains exactly l components with ends $b_{i,1}, b_{i,2}$ $(1 \leq i \leq l)$, then we determine the value of $C = A * B$ as follows: we set the number of components of C to be $k \cdot l$. For this process we can use double indices for the components of C. The starting- and end points of the ij-th component are $a_{i1} + b_{j1}(a_{i2} - a_{i1})$ and $a_{i1} + b_{j2}(a_{i2} - a_{i1})$, respectively.

The idea and the role of this operation is similar to that of unlimited shrinking of 2-dimensional images in optical computations ([15]). It will be used to connect interval-values of different resolution. As we can observe in Figure 5, as well, the fractalian product of two interval-values is the result of shrinking the first operand to each component of the second one.

4 A representation of n independent truth values

In this section we give a natural interval-valued computational representation of all variations of n independent truth values which is only a visual rephrase of the well-known truth tables and will be useful not only in checking whether a given propositional formula is a logical law or not but also in the interval-valued computations deciding whether a given quantified propositional formula is true or not.

For lack of space, we do not define formally the *interval-valued computations*, consult [12] or [13] for the formal details. Our focus is on the visual expressivity of the model. For the aims of the present paper it is enough to know that it is a sequence of interval-values where each new member of the sequence results from an operator application of one or two precedents in the same sequence and which is starting with $\left[0, \frac{1}{2}\right)$. In this manner, *deciding a language L by an interval-valued computation* means constructing an algorithm that for any input problem instance responds an interval-valued computation sequence with the following property: the result of the interval-valued computation sequence created by the algorithm to an input word w is equal to the unit interval $[0, 1)$ if and only if w is in L.

We give a computation which constructs a quite natural interval-valued representation of n independent truth values. Let K_1 be $\left[0, \frac{1}{2}\right)$. For all non-negative integers k, we define $K_{3k+2} = K_1 * K_{3k+1}$, $K_{3k+3} = RShift(K_{3k+2}, K_{3k+1})$ and $K_{3k+4} = K_{3k+2} \cup K_{3k+3}$.

Fact 1 *By an induction on k one can establish that*

$$K_{3k+1} = \bigcup_{l=0}^{2^{k-1}-1} \left[\frac{2l}{2^k}, \frac{2l+1}{2^k}\right).$$

By the previous fact this computation sequence produces suitable interval-values since it satisfies the following.

Fact 2 *For any $(x_1, \ldots, x_n) \in \{0, 1\}^n$ there exists $r \in [0, 1)$ satisfying that $(x_1, \ldots, x_n) = (r \in K_1, r \in K_4, \ldots, r \in K_{3n+1})$.*

All of our interval-valued computations (at least the ones deciding validity of quantified propositional formulae) will start with the construction of $K_1, \ldots, K_{3n+1}$, if n is the number of propositional variables of the input formula. The first 4 interval-values in Figure 6 and 8 are K_1, K_4, K_7, K_{11}, they represent 4 independent truth values of x_1, x_2, x_3, x_4. This method is an alternative of Venn/Euler diagrams to have all possible combinations of the truth values of the Boolean variables in the same diagram. The novel idea is that we assign 1 dimensional objects (interval-values) for the variables without requiring their connectivity.

Of course, using these interval-values representing all possible variations of the truth/falsity of the propositional variables $x_1, \ldots, x_n$, one can easily decide the validity of propositional formulae, by executing the Boolean operations on the interval-values on the desired order. In this way any propositional formula $\varphi(x_1, \ldots, x_n)$ gets its interval-truth-value by an appropriate interval-valued computation $C(\varphi)$. Not specifying $C(\varphi)$ more formally, we can observe that for any propositional formula ϕ built from propositional variables $x_1, \ldots, x_n$ and for any $r \in [0, 1)$ the following holds: $r \in C(\varphi) \Leftrightarrow \varphi$ is satisfied by the truth valuation $(x_1 : (r \in K_1), x_2 : (r \in K_4), \ldots, x_n : (r \in K_{3n+1}))$. The fifth lines of Figure 6 and 8 represent $C(\varphi)$ for the two given formulae, respectively.

5 Visual solution of a PSPACE-complete problem

We employ the visual reasoning power of interval-valued computations for a more complex task. We show that the sequence of interval-values produced by the computation represents visually the full information needed to understand the solution of

the given case of the PSPACE-complete problem QSAT, i.e. the problem whether any given quantified propositional formula is true. This problem is decidable by a linear interval-valued computation.

We specify visually the needed computation. The computation starts with the determination of the interval-values of the independent variables (lines 1–4 on Figures 6 and 8). We concentrate on that how these interval-values visually encode the information needed to follow the decision process for validity of quantified propositional formulae.

A quantified propositional formula –without loss of generality – is of form $\forall t_1 \exists t_2 \dots Q_n \varphi$ where Q_i is $\forall$ for odd i and $\exists$ for even i. It is called true or valid if and only if $\forall t_1 \in \{0,1\} \exists t_2 \in \{0,1\} \dots Q_n t_n \in \{0,1\} \varphi(x_1 : t_1, \dots, x_n : t_n)$ holds.

By using only Boolean operators the interval-value of the quantifier-free formula φ can be computed (the result can be seen on line 5 in Figures 6 and 8). Then by using shift and logical operations in an appropriate way, one can continue the computation in a way such that the interval-valued decision algorithm constructs interval-values $C_0(\varphi)(= C(\varphi)),\ C_1(\varphi), \dots, C_n(\varphi)$ with the following properties.

- $C_i(\varphi) = \{r \in [0,1) :$
  $Q_{n-i+1} t_{n-i+1} \dots Q_n t_n$
  $\varphi(x_1 : (r \in K_1), \dots, x_{n-i} : (r \in K_{3(n-i)-1}), x_{n-i+1} : t_{n-i+1}, \dots, x_n : t_n)\}$,
- $C_{i+1}(\varphi)$ can be constructed from $C_i(\varphi)$ and the interval-value corresponding to x_{n+1-i}, that is, from $K_{3(n+1-i)-1}$.

Figure 7 shows the way of computation at existential quantifier. By disjunction and shift operators the corresponding neighbor parts of the components are also filled. The corresponding neighbor parts of the interval-values are the following interval pairs: $\left[\frac{2l}{2^k}, \frac{2l+1}{2^k}\right)$ and $\left[\frac{2l+1}{2^k}, \frac{2l+2}{2^k}\right)$ where k is the index of the quantified variable we are dealing with in the actual step. They are not separated by vertical lines on Figures 6, 7 and 8. A part and its corresponding neighbor differ (i.e. exactly one of them is contained by the interval-value) if and only if the value of the formula depends on the value of the actual variable using the fixed values of the other variables that are represented by the actual part of the interval-value.

The steps to determine $C_{i+1}(\varphi)$ needs alternating $\forall$- and $\exists$-transformations. A $\forall$-step means checking the interval AND its corresponding neighbor in $C_i(\varphi)$, while an $\exists$-step amounts to checking the interval OR its corresponding neighbor. $\forall$-step visually means simply a check if the appropriate neighbor of the examined subinterval is also in $C_i(\varphi)$ while an $\exists$-step checks if the appropriate neighbor of the examined subinterval OR the subinterval itself is in $C_i(\varphi)$. An example of $\exists$-transformation is presented on Figure 7, the $\forall$-transformations are going in a similar manner. Since the steps compute the parts of the interval-value of the various corresponding parts in a parallel way, an $\exists$-step and a $\forall$-step needs a constant number of operation application on interval-values (see Figure 7, where the computation obtaining line 6 of Figure 6 is shown using line 5 and the value of the variable x_4). Since the number of these steps exactly the same as the number of the variables, the computation can be performed in a linear number

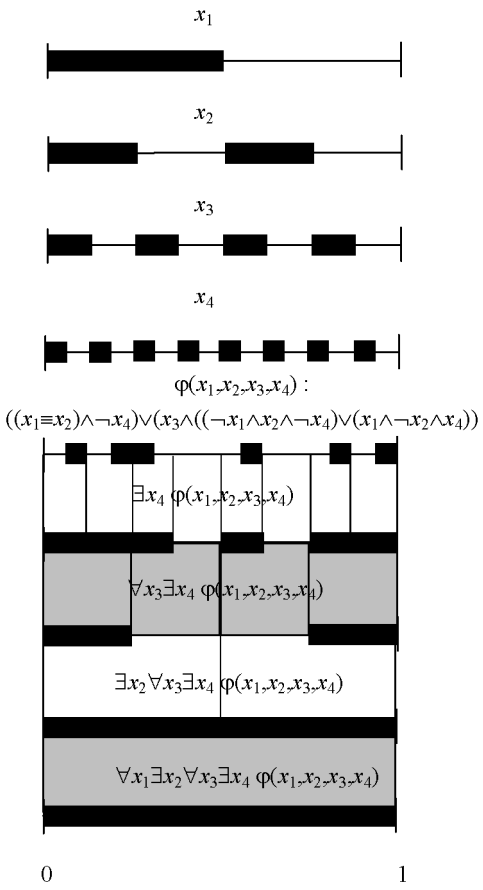


Figure 6: This quantified formula is true

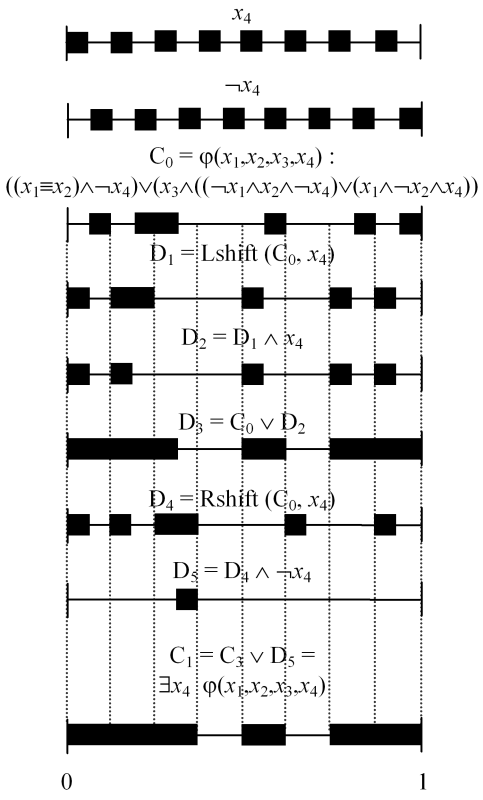


Figure 7: Visual computation at existential quantifier

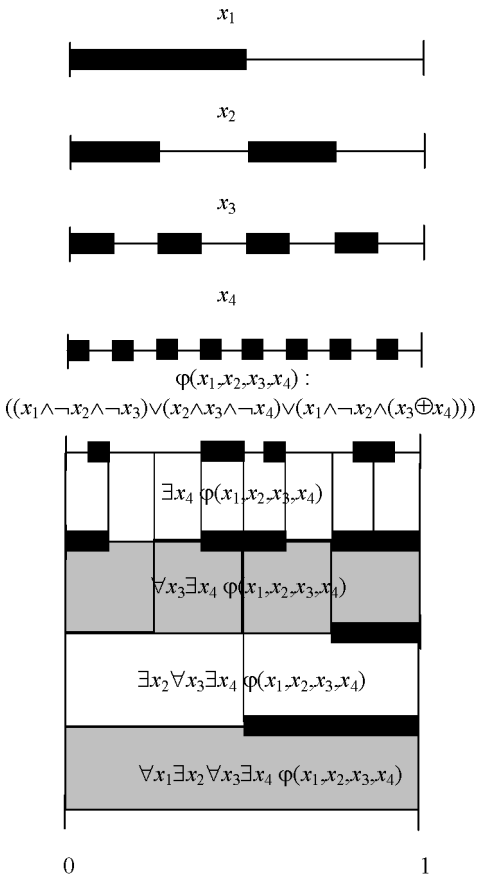


Figure 8: This quantified formula is not true

of operation, i.e. a linear size of algorithm on the length of the computation sequence (number of computed interval-values).

In Figure 6 and 8 one can follow two interval-valued computations deciding whether a given quantified propositional formula is true. The lines 1–4 show the 4 independent truth values, the 5th line shows the result of the evaluation of the Boolean operators and lines 6–9 include the result of adding one quantifier per line to the formulae. Finally, the QSAT formula is true if and only if the resulted interval-value (line 9) is [0,1). (If it is not true, the empty interval is obtained.)

6 Interval-valued computations and interval temporal logic

Temporal logic also has strong connection to visual computing. In [4], a visual specification language of propositional temporal conditions is given which constitutes a subset of propositional temporal logic, more specifically, interval temporal logic. In [8], an interval temporal logic for repeating temporal events is introduced. Thinking [0,1) as a time flow we can investigate its temporal logic. If we consider only classical temporal operators, then its temporal logic trivially coincides with the temporal logic of $(\mathbb{R}^{+0}, <)$ where $\mathbb{R}^{+0}$ is the set of nonnegative reals. However, it is an interesting question, what happens if we add the non-logical operators of interval-values to the temporal logic over [0,1) as binary modal operators.

Definition 5 *The members of the following set of formulae are interval-valued modal-temporal formulae. It is the minimal set of strings satisfying the following:*

- $a, b, \ldots$ *are (atomic) formulae,*
- $FirstHalf$ *is a formulae,*
- *if* φ, θ *are formulae, then* $(\varphi \wedge \theta)$, $(\varphi \vee \theta)$ *and* $\neg\varphi$ *are formulae, too,*
- *if* φ, θ *are formulae,* $\Box\!\!\rightarrow \varphi$ *and* $\leftarrow\!\!\Box\varphi$ *are formulae, too,*
- *if* φ, θ *are formulae then* $R(\varphi, \theta)$, $L(\varphi, \theta)$ *and* $P(\varphi, \theta)$ *are formulae, too. (*R, L *and* P *are binary operators, they coincide with the shift and the product operators.)*

Definition 6 *An interval-valuation* v *is a function assigning to each member of* $\{a, b, \ldots\}$ *an interval-value. Then for any interval-valued modal-temporal formula* $\|\varphi\|_v$ *is an interval-value of the interval-valued modal-temporal formula* φ*. The definition of this notion is the expected one. We just write three clauses of this definition.*

- $\|FirstHalf\|_v = \left[0, \frac{1}{2}\right)$,
- $\|\Box\!\!\rightarrow \varphi\|_v$ *is* $\{t \in [0,1) \ : \ (t,1) \subseteq \|\varphi\|_v\}$,
- $\|P(\varphi, \theta)\|_v = \|\varphi\|_v * \|\theta\|_v$.

The shift operators have intuitive meaning in this temporal logic: an event can start only earlier/later by the starting component of the value of a second event. The product operator can be explained as a modal operator in the following way. $P(\varphi, FirstHalf)$ expresses that φ holds at the first half of the actual evaluating interval, or generally,

$P(\varphi, \theta)$ expresses that φ holds at that parts of the actual evaluation interval what belong to the down-scaled "copy" of $\|\theta\|_v$.

A modal-temporal formula is said to be modal-temporal logical law if with every valuation v its interval-value is [0,1).

Problem 1 *How to axiomatize this kind of modal-temporal logic? Is it decidable? If yes, what is its complexity?*

We have a partial answer to this question.

Claim 1 *The problem if a modal-temporal formula built up only from $FirstHalf$ but without other propositional variables is decidable by exponential time. If the usage of the product operator is restricted such that it always takes a product with $FirstHalf$, then the arising problem is solvable in polynomial space. Moreover there is a PSPACE-complete problem is among them (as it was presented).*

7 Conclusion

We have demonstrated the visual reasoning power of a recent unconventional computing system. Its expressiveness depends on data representation by interval-values which makes it possible by its topological properties.

It is worth thinking over what further problems can be naturally represented by generalized intervals. Possible candidates are problems about occurring events in temporal logic with a notion of compositionality. The product operator would provide transfer between different compositional levels; embeddability of macro- and micro scales can be conceptualized. In this way, also visual analysis and visual representation of re-occurring, periodic hierarchical events – e.g. in biostatistics and health insurance – would be available.

Further generalization is possible to regions in higher dimensional spaces, mainly to $\mathbb{R}^2$. In this way one should work out the connections of interval-valued computing to so-called optical computing where objects of computing are 2-dimensional images ([15]) through their visual applications.

Acknowledgements

Comments of the reviewers are gratefully acknowledged. This work has been supported by the grant of the Hungarian National Foundation for Scientific Research OTKA T049409, by a grant of the Hungarian Ministry of Education and by the Öveges programme of the Agency for Research Fund Management and Research Exploitation (KPI) and National Office for Research and Technology .

References

[1] Anderson, D., E., and F. L. Cleaver, *Venn-type diagrams for arguments of n terms*, J. Symb. Logic **30** (1965), 113–118.

[2] Balbiani, P., J.-F. Condotta, L. Farinas del Cerro, and A. Osmani, *Reasoning about generalized intervals*, in: F. Giunchiglia (ed), Artificial Intelligence:

Methodology, Systems and Applications, Lecture Notes in Artificial Intelligence **1480** (1998), 50–61, Springer.

[3] Chilakamarri, K., B., and R. E. Pippert, *Venn diagrams and planar graphs*, Geometriae Dedicata **62** (1996), 73–91.

[4] Dillon, L., K., G. Kutty, L. E. Moser, P. M. Melliar-Smith and Y. S. Ramakrishna, *A graphical interval logic for specifying concurrent systems*, ACM Transactions on Software Engineering and Methodology (TOSEM) **3** (1994), 131–165.

[5] Edwars, A., W., F., "Cogwheels of the Mind: The Story of Venn Diagrams," John Hopkins University Press, 2004.

[6] Henderson, D., W., *Venn diagrams for more than four classes*, Amer. Math. Monthly **70** (1963), 424–426.

[7] Ligozat, G., *On generalized interval calculi*, AAAI-91 Proc. of the 9th National Conf. on Artif. Intelligence **1** (1991), 234–240.

[8] Morris, R., A., and L. Khatib, *Quantitative Structural Temporal Constraints on Repeating Events*, IEEE Proceedings of the Fifth International Workshop on Temporal Representation and Reasoning (1998), 74–80.

[9] Nagy, B., and G. Allwein, *Diagrams and Non-monotonicity in Puzzles*, in: Diagrammatic Representation and Inference, Proc. of Diagrams'2004, 3rd Int. Conf. on Theory and Application of Diagrams, Cambridge, England, (eds.:A. Blackwell, K. Marriott, A. Shimojima) Lect. Notes in Comp. Sci. LNAI **2980** (2004), 82–96.

[10] Nagy, B., *Reasoning by intervals*, Proc. 4th Int. Conf. on Theory and Applications of Diagrams, Stanford(CA), USA, Lect. Notes in Comp. Sci. LNAI **4045** (2006), 145–147.

[11] Nagy, B., *An Interval-valued Computing Device*, in: Computability in Europe 2005: New Computational Paradigms, (eds. S. B. Cooper, B. Löwe, L. Torenvliet), ILLC Publications X-2005-01, Amsterdam, 166–177.

[12] Nagy, B., and S. Vályi, *Solving a PSPACE-complete problem by a linear interval-valued computation*, in: Proc. of Conf. Computability in Europe 2006: Logical Approaches to Computational Barriers, (eds. A. Beckmann, U. Berger, B. Loewe), Uni. of Swansea Report no. CSR-7-2006, Swansea, 216–225. http://www.cs.swansea.ac.uk/reports/yr2006/CSR7-2006.pdf

[13] Nagy, B., and S. Vályi, *Interval-valued computations and their connections with PSPACE*, accepted for publication in Theoretical Computer Science.

[14] Ruskey, F., and M. Weston, *A survey of Venn diagrams*, Electronic Journal of Combinatorics **12** 2005.

[15] Woods, D., and T. Naughton, *An optical model of computation*, Theoretical Computer Science **334** (2005), 227-258.

Spider Diagrams of Order

Aidan Delaney* and Gem Stapleton†
Visual Modelling Group,
University of Brighton,
Brighton, United Kingdom BN2 4GJ

Abstract

Spider diagrams are a visual logic capable of makeing statements about relationships between sets and their cardinalities. Various meta-level results for spider diagrams have been established, including their soundness, completeness and expressiveness. Recent work has established various relationships between spider diagrams and regular languages, which highlighted various classes of languages that spider diagrams could not define. In particular, this work illustrated the inability of spider diagrams to place an order on certain letters in words. To overcome this limitation, in this paper we introduce *spider diagrams of order*, incorporating an order relation and present a formalisation of the syntax and semantics. Subsequently, we define the language of such a diagram and establish that the class of such languages includes that of the piecewise testable languages.

1 Introduction

Diagrams are often used to convey information and aid communication in a variety of areas, including software engineering, mathematics and every day life. Recently, the perception of the role of diagrams in logic has been overturned, with advances showing that diagrams can be given precise syntax and semantics with, subsequently, formal reasoning systems being built on them; for example [5, 6, 8, 14, 17]. As a result, the utility of diagrams is seen as broader, and some considerable effort is now being placed on exploring visual languages in the context of logic.

One such logic is the language of spider diagrams (see, for example [8, 16]). With regard to applications of spider diagrams, they have been used to assist with the task of identifying component failures in safety critical hardware designs [1] and (implicitly) in a variety of other areas, such as [2, 9, 18]. It has been established that spider diagrams have the expressiveness of monadic first order logic with equality by providing translations between these two languages that preserves semantics [16]. In this paper, we consider the expressiveness of spider diagrams in comparison with regular languages, building on results presented in [3] where a limitation is highlighted. In particular, regular languages often constrain the orders that letters may appear in a word of that language, but spider diagrams are unable to do this. To overcome this expressiveness limitation, we extend the spider diagram language to include facilities for ordering elements. Extending spider diagrams to include an order relation will allow them to be

*a.j.delaney@brighton.ac.uk
†g.e.stapleton@brighton.ac.uk

used in more application areas. For example, one may choose to use spider diagrams over finite state machines when defining languages; see [3] for further discussions on this relationship.

One of our goals in increasing the expressiveness of spider diagrams is to provide a specification tool for trace semantics and synchronisation expressions. Trace semantics and synchronisation expressions have existing formal language characterisation in [4] and [13] respectively. Our first step is to extend spider diagrams and examine the ramifications with respect to formal language theory. A longer term goal of this body of work is to examine whether diagrammatic logics make a more succinct 'programming language' for problems with solutions in regular language space. The results in [3] on the descriptional complexity of spider diagrams and finite state automata support this succinctness conjecture.

In more general terms, the study of the relationships between logics and formal languages has led to a range of important results related to decidability, the circuit synthesis problem and has provided new perspectives to the construction of non-terminating programs, discussed in [19]. In this vein, it may well prove fruitful to further our understanding of the relationship between spider diagrams and regular languages. For example, fragments of the spider diagrams language might correspond to classes of regular languages that are not naturally characterised in any other way. Consequently, this may provide a deeper understanding of the relationships between classes of regular languages themselves.

In section 2, we briefly overview the existing spider diagram notation. Section 3 introduces various concepts from formal language theory that are necessary for this paper and discusses the relationship between spider diagrams and regular languages. *Spider diagrams of order* are introduced and formalised in section 4. Finally, in section 5, we prove that a fragment of the language of spider diagrams of order gives rise to the well known class of piecewise testable languages also called level 1 of the Straubing-Thérin hierarchy.

2 Spider Diagrams

This section will provide a brief overview of the spider diagram syntax presented in [8]. In figure 1, the spider diagram d_1 contains two labelled *contours*, A and B. Contours are simple closed curves. The diagram also contains three minimal regions, called *zones*. There is one zone inside A, another inside B and the other zone is outside both A and B. Each zone can be described by a two-way partition of the contour label set. The zone inside the contour A can be described as inside A but outside B. A *region* is a set of zones. The two zones outside B contain a *spider*; spiders are trees whose vertices, called *feet*, are placed in zones (in this case, the spider has two feet). Spider diagrams can also contain *shading* placed in zones, as in d_2 (which contains two spiders and four zones of which one is shaded). The horizontal line connecting d_1 and d_2 in figure 1 denotes disjunction between diagrams; thus, the figure contains $d_1 \vee d_2$. Similarly, juxtaposition of two diagrams d_1 and d_2 with no connecting line denotes their conjunction, $d_1 \wedge d_2$.

Our attention now turns to the semantics. Spider diagrams make statements about sets (represented by contours) and their cardinalities (by using spiders and shading). In figure 1, d_1 expresses that A and B are disjoint, because there are no points interior to both of the contours. Spiders assert the existence of elements, so d_1 specifies that there is (at least) one elements in either A or the universe outside both A and B. The spiders

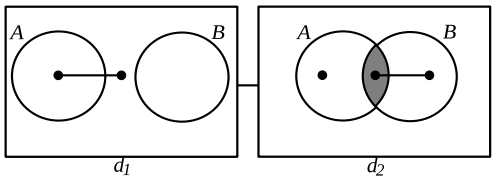


Figure 1: Two spider diagrams.

in d_2 assert that there are at least two elements, one of which is in $A - B$ and the other is in $A \cap B$ or $B - A$. Shading is used to place upper bounds on set cardinality: in the set represented by a shaded region, all of the elements are represented by spiders. For example, d_2 expresses that the set $A \cap B$ contains at most one element.

3 The Straubing-Thérin Hierarchy

In our previous work [3] we have studied the relationship between spider diagrams and star-free regular languages. We established that sets of words from a subset of star-free regular languages can be thought of as corresponding to *models* for spider diagrams (a model will be formally defined later). The Straubing-Thérin hierarchy serves as a fine-grained tool for describing various subsets of star-free regular languages. This hierarchy is infinite but it is an open question as to whether the hierarchy is proper above so-called 'level 2'.

Level 0 of the Straubing-Thérin hierarchy is the set of languages $\{\Sigma^*, \emptyset\}$ where Σ is an alphabet. Level $1/2$ is the well known *shuffle ideal* set, which is the polynomial closure of Level 0. Level 1 is defined as the boolean closure of $1/2$. This hierarchy has been extended by Pin to consider varieties of languages [10]: in general for any positive integer $n > 0$

level $n + \frac{1}{2}$ is the polynomial closure of level n, and

level $n + 1$ is the boolean closure of level $n + \frac{1}{2}$.

The boolean closure of a set of languages $\mathscr{L} \subseteq \Sigma^*$ is $B(\mathscr{L})$ which is formed by taking the union, intersection and complement of languages. The polynomial closure of a set of languages $\mathscr{L} \subseteq \Sigma^*$, $Pol(\mathscr{L})$, is the finite union of languages of the form $L_0 a_1 L_1 \ldots a_n L_n$ where $L_0, L_1, \ldots, L_n \in \mathscr{L}$ and $a_1, \ldots, a_n \in \Sigma$.

For our purposes, it is sufficient to state that a formal language is a set of words defined over an alphabet, Σ. The boolean operations $\cup, \cap$ and $\subset$ and the unary complement $\neg$ operator maintain their well understood semantics over sets of words. The additional boolean operation called the shuffle product, denoted $\sqcup$, will allow us to utilise characterisations of the Straubing-Thérin hierarchy more suitable for our definitions and theorems.

Definition 3.1. *The **shuffle product** of two languages L_1, L_2 denoted $L_1 \sqcup L_2$ informally takes all words from L_1 and intersperses letters from each word in L_2. More formally, the words in $L_1 \sqcup L_2$ are precisely those of the form $w_0 w_1 \ldots w_n$ where there exists a partition $I \cup J$ of $\{1, 2, \ldots, n\}$ with*

1. $I = \{p_1, p_2, \ldots, p_i\}, p_1 < p_2 < \ldots < p_i$,

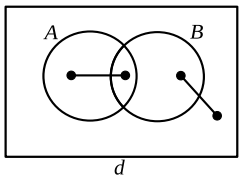


Figure 2: A spider diagram.

2. $J = \{q_1, q_2, \ldots, q_j\}, q_1 < q_2 < \ldots < q_j$ *(thus* $i + j = n$*), and*

3. $w_{p_1} \ldots w_{p_i} \in L_1$ *and* $w_{q_1} \ldots w_{q_j} \in L_2$.

As an example, the shuffle product of the sets of words $A = \{xy\}$ and $B = \{yz, y\}$ is the set of words $\{xyyz, xyzy, yxzy, yzxy, yxyz, xyy, yxy\}$. Languages of catenation level $1/2$, that is the shuffle ideals, are of the form $k \sqcup \Sigma^*$ where k is a finite set of words.

In this paper, we are concerned with the relationship between spider diagrams and regular languages. We have already established various relationships between spider diagrams and catenation hierarchy levels $1/2$ and $3/2$. In particular, we proved that spider diagrams give rise to languages that are closed under permutation of words and, thus, cannot constrain a language to contain words, w, in which certain letters must occur before others.

As an example, the diagram d in figure 2 represents a star-free language of words over the four-letter alphabet $\Sigma = \{AB, \overline{A}B, A\overline{B}, \overline{AB}\}$; here the alphabet has been obtained by considering the contours in the diagram, i.e. A and B, and the four possible combinations of being inside or outside the contours, with $\overline{A}$ denoting 'being outside A'. The diagram asserts, by way of the spiders, that there is an element in $A - B$ or $A \cap B$ (because of the spider placed inside A) and an element not in A (by the placement of the other spider). In terms of regular languages, we can take this diagram as asserting that all words contain letters corresponding to these possibilities given rise to by the spiders. The spider inside A, therefore, tells us that the words must contain either the letter $A\overline{B}$ or AB. The other spider tells us that words must contain either $\overline{A}B$ or the letter $\overline{AB}$. We further refine the notation to include square brackets, $[A\overline{B}]$ to aid readability of words. Words in the language of the diagram, denoted $\mathscr{L}(d)$, are precisely those that contain at least one letter from the first spider and one letter from the other spider, in either order. Using the characterisation of shuffle-ideal languages given above we may construct a set of words, k, such that $\mathscr{L}(d) = k \sqcup \Sigma^*$. Such a k is given by

$$\begin{aligned} k \;=\; & \{[A\overline{B}][\overline{A}B], [\overline{A}B][A\overline{B}], [A\overline{B}][\overline{AB}], \\ & [\overline{AB}][A\overline{B}], [AB][\overline{A}B], [\overline{A}B][AB], [AB][\overline{AB}], [\overline{AB}][AB]\} \end{aligned}$$

and we observe that k is closed under permutation. The language $\mathscr{L}(d) = k \sqcup \Sigma^*$ maintains the closure under permutation property of the set k.

Spider diagrams are a monadic first order logic with equality (MOFLe) [16]. We are interested in the relationship between logics and subsets of regular languages. The main body of literature discussing this relationship [7, 10, 11, 12, 19] assumes the existence of an order relation $<$ adjunct to the standard monadic first order operators of $\neg, \vee, \wedge, \iff$, the quantifiers $\exists$ and $\forall$ and predicates of the form $P_a(x)$ which

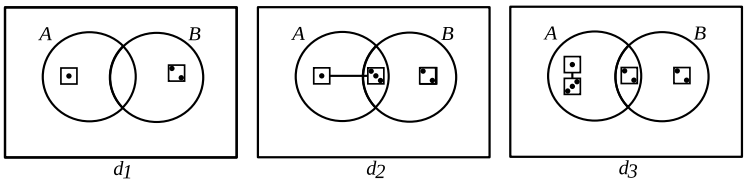


Figure 3: Generalising spider diagrams.

states that the letter a is at positive position x in a word w. Intuitively, if we do not have an order relation, $<$, as in MOFLe then any language corresponding to a formula will be closed under permutation. In other words, languages of the form, for example, $\Sigma^* A \Sigma^* B$ (A comes before B in every word) do not correspond to languages arising from formulae in MOFLe. Consequently spider diagrams do not contain facilities for ordering elements and it is this main body of literature, referenced above, that provides a motivation for generalising spider diagrams to include facilities for ordering elements.

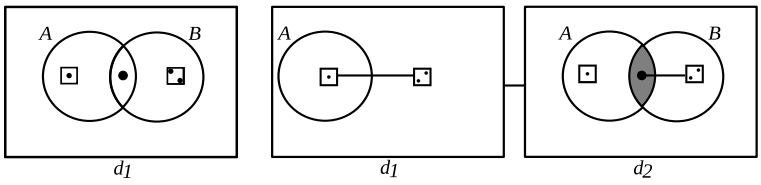
4 Generalising Spider Diagrams to Include an Order Relation

As just stated, spider diagrams are limited in their expressive power and cannot enforce any kind of order on elements represented by the spiders. Here, we generalise the spider diagram syntax and extend their semantics appropriately to overcome this expressiveness limitation. For example, the *spider diagram of order* labelled d_1 in figure 3 contains spiders whose feet are labelled with dots. The number of dots is used to place an order on the elements represented by the spiders (alternative syntax would simply label the feet with natural numbers as opposed to dots; such a change of syntax would have no impact on the work that follows and is merely a different means of visualisation). Thus, this diagram d_1 is interpreted as saying that there is an element, x, in $A - B$ and another element, y, in $B - A$ such that $x < y$. The semantics of the spider diagram of order labelled d_2 are a little more subtle. This diagram expresses that there is an element, x, in A and an element, y, in $B - A$ such that, if $x \in A - B$ then $x < y$, otherwise $y < x$. Here we see that the labels can be used to place an order on the elements represented by the spiders in the context of which sets those elements are located.

A further modification is to allow spiders to have more than one foot placed in each zone, an idea first raised in [15] but not in the context of ordering elements. For example, in figure 3, d_3 expresses that there is an element, x, in $A-B$, another element, y, in $B - A$ and a third element, z, in $A \cap B$. The element x satisfies either $x < y$ and $x < z$ or $y < x$ and $z < x$. The elements y and z are both represented by spiders that have the same label (i.e. 2) and we interpret this as expressing y and z can be in either order: $y < z$ or $z < y$.

Sometimes we might want to express an order on certain elements (as in the examples we have just seen) but not on other elements; in the previous example, we did not specify an order on y and z. Suppose we want express the following:

1. there exist three distinct elements, x_1, x_2 and x_3,



(a) A unitary diagram. (b) A compound diagram.

Figure 4: Spider Diagrams of Order.

2. x_1 is in the set $A - B$,
3. x_2 is in the set $B - A$,
4. x_3 is in the set $A \cap B$, and
5. $x_1 < x_2$.

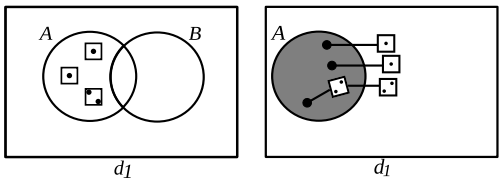
To allow this statement to be expresses succinctly by spider diagrams, we allow the use of non-labelled feet as in the original notation. A diagram of order making this statement can be seen in figure 4(a), where the spider placed in $A \cap B$ has no label, thus indicating we do not mind whether $x_1 < x_3$ or $x_3 < x_1$, for example.

With regard to shading, it is interpreted in the same way as the original notation: in a shaded region, all of the elements are represented by spiders. As with the spider diagram language in [8], we allow diagrams to be taken in disjunction and conjunction, forming *compound diagrams*. In addition, the compound diagram $\neg d$ is also allowed. We have just provided various examples of unitary spider diagrams of order. The *compound diagram* in figure 4(b) represents the disjunction of two unitary diagrams $d_1 \vee d_2$. The horizontal line joining d_1 to d_2 denotes disjunction; the juxtaposition of the unitary diagrams would represent conjunction. For the remainder of this section, we provide a formalisation of the syntax and semantics of spider diagrams of order.

4.1 Syntax

Each spider diagram of order consists of contours (closed, plane, labelled curves), spiders which are trees whose nodes (called *feet*) are a character such as $\bullet, \boxdot, \boxdot, \boxdot, \ldots$ and shading. For example, the diagram d_1 in figure 4(a) contains two contours labelled A and B and three spiders, one with a foot labelled $\boxdot$, another with a foot labelled $\bullet$, and the other with a foot labelled $\boxdot$. In general, any given spider may contain both *ordered feet* (those of the form $\boxdot$) and *unordered feet* (those of the form $\bullet$).

Formally, the syntax is defined at an abstract level, extending that given in [16]. The contour labels in spider diagrams are selected from a finite set $\mathcal{L}$. A **zone** is defined to be a pair, (in, out), of finite disjoint subsets of $\mathcal{L}$. The set in contains the labels of the contours that the zone is inside whereas out contains the labels of the contours that the zone is outside. The set of all zones is denoted $\mathcal{Z}$. To describe the spiders in a diagram, it is sufficient to say how many spiders there are with any given foot arrangement. For example, in figure 5(a), there are two spiders inside A with a single foot labelled 1 and another spider also inside A but with single foot labelled 2. Thus, our abstract definition of a spider diagram will specify the labels used, the zones, identify which



(a) Simple example. (b) More complex example.

Figure 5: Illustrating the syntax.

zones are shaded and use a set of spider identifiers to describe the spiders. We have adopted this approach because it directly extends the abstract syntax presented in [16].

To begin our formalisation, we start by defining spider feet, which may be ordered, denoted with an integer index, or unordered, denoted with a $\bullet$ character and subsequently we define spiders. When we formalise the semantics, it is useful to have access to the region in which a spider is placed, called its *habitat*.

Definition 4.1. *A **spider foot** is an element of the set $(\mathbb{Z}^+ \cup \{\bullet\}) \times \mathcal{Z}$ and the set of all feet is denoted $\mathcal{F}$. A **spider**, s, is a set of feet together with a number: $s \in \mathbb{Z}^+ \times (\mathbb{P}\mathcal{F} - \{\emptyset\})$ and the set of all spiders is denoted $\mathcal{S}$. The **habitat** of a spider $s = (n, p)$ is the region $habitat(s) = \{z : \exists k\,(k, z) \in p\}$.*

Spiders are numbered because unitary diagrams can contain many spiders with the same foot set; essentially, we view a unitary diagram as containing a bag of spiders.

Definition 4.2. *A **unitary spider diagram of order** is a quadruple $d = \langle L, Z, ShZ, SI\rangle$ where*

$L = L(d) \subseteq \mathcal{L}$ *is a set of contour labels,*

$Z = Z(d) \subseteq \{(a, L - a) : a \subseteq L\}$ *is a set of zones,*

$ShZ = ShZ(d) \subseteq Z(d)$ *is a set of shaded zones,*

$SI = SI(d) \subsetneq \mathcal{S}$ *is a finite set of **spider identifiers** such that for all $(n_1, p_1), (n_2, p_2) \in SI(d)$,*

$$(p_1 = p_2 \implies n_1 = n_2) \wedge habitat(n_1, p_1) \subseteq Z(d).$$

The symbol $\perp$ is also a unitary spider diagram. We define

$$L(\perp) = Z(\perp) = ShZ(\perp) = SI(\perp) = \emptyset.$$

*If d_1 and d_2 are spider diagrams then $(d_1 \wedge d_2), (d_1 \vee d_2)$ and $\neg d_1$ are **compound spider diagrams of order**.*

We observe that the set of spider identifiers is just a set of spiders, but with at most one spider with any given foot arrangement present. If, for example, the spider identifier set contains the pair $(2, \{(\bullet, z)\})$ then this would tell us that d contains two spiders in zone z whose feet are of the form $\bullet$. As a more concrete example, the spider diagram of order in figure 5(b) has abstract syntax:

1. labels $L(d) = \{A\}$

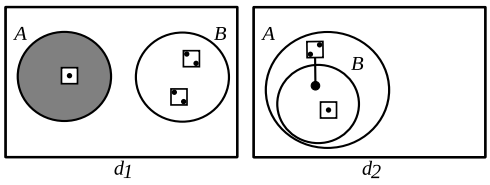


Figure 6: Illustrating the semantics.

2. zones $Z(d) = \{z_1 = (\{A\}, \emptyset), z_2 = (\emptyset, \{A\})\}$
3. shaded zones $ShZ(d) = \{z_1\}$
4. spider identifiers $SI(d) = \{(2, \{(\bullet, z_1), (1, z_2)\}), (1, \{(\bullet, z_1), (2, z_1), (2, z_2)\})\}$.

It is useful to identity the set of spiders present in a diagram, which is implicit in the spider identifier set and to be able to arbitrarily select feet of spiders. For example, when defining the semantics, each spider, s, represents and element and the feet place a disjunction of constraints on that element; thus to identify whether and *interpretation* (see below) is a model for a unitary diagram there needs to be a choice of foot for which s satisfies the constraint imposed.

Definition 4.3. *The set of **spiders** in unitary diagram d is defined to be*

$$S(d) = \{(i, p) : \exists (n, p) \in SI(d)\, 1 \leq i \leq n\}.$$

*Let $FootSelect\colon S(d) \to \mathcal{F}$ be a function. If, for all $(n, p) \in S(d)$, $FootSelect(s) \in p$ then $FootSelect$ is called a **foot selection function** for d.*

It is further useful to identify which zones could be present in a unitary diagram, given the label set, but are not present; semantically, *missing* zones provide information.

Definition 4.4. *Given a unitary diagram, d, a zone (a, b) is said to be **missing** if it is in the set $\{(a, L - a) : a \subseteq L\} - Z(d)$ with the set of such zones denoted $MZ(d)$. If d has no missing zones then d is in **Venn form** [8].*

4.2 Semantics

Regions in spider diagram represent sets and the spatial arrangement of the contours places constraints on the relationships between those sets. For example, in figure 6, the diagram d_1 contains two contours, A and B, that do not overlap, indicating that the sets they represent are disjoint. Spiders make existential statements about elements. In particular, each spider denotes the existence of a particular element in the set represented by the spider's habitat, with distinct spiders denoting distinct elements. From d_1 we can deduce that there is an element, x, in A and two elements, y and z, in B. The numbers on the spiders feet provide information on the ordering of elements in the universe. With regard to d_1, we deduce that $x < y$ and $x < z$. The shading tells us that all of the elements are represented by spiders, so d_1 asserts that A contains a single element, namely x. The diagram d_2 expresses $B \subseteq A$ (because B is placed inside A) and there is an element, x, in A and another element, y, in B such that if $y \in A - B$

then $x < y$. The foot labels have particular significance as they are used to place a restriction on the order of elements.

To formalise the semantics, we need to first interpret the contour labels as sets and interpret the order relation $<$. Thus, we extend the definition of an interpretation given in [8] to spider diagrams of order.

Definition 4.5. *An **interpretation** is a triple $(U, \Psi, <)$ where U is a universal set and $\Psi\colon \mathcal{L} \to \mathbb{P}U$ is a function that assigns a subset of U to each contour label and $<$ is an irreflexive, antisymmetric and transitive relation on U. The function Ψ can be extended to interpret zones and sets of regions as follows:*

1. *each zone, $(a, b) \in \mathcal{Z}$, represents the set $\bigcap_{l \in a} \Psi(l) \cap \bigcap_{l \in b} \overline{\Psi(l)}$ and*
2. *each region, $r \in \mathbb{P}\mathcal{Z}$, represents the set which is the union of the sets represented by r's constituent zones.*

For brevity, we will continue to write $\Psi\colon \mathcal{L} \to \mathbb{P}U$ but assume that the domain of Ψ includes the zones and regions also. Given an interpretation we wish to know whether it is a model for a diagram; in other words, when the information provided by the interpretation agrees with the intended meaning of the diagram. Informally, an interpretation is a *model* for unitary diagram d ($\neq\bot$) whenever

1. all of the zones which are missing represent the empty set,
2. all of the regions represent sets whose cardinality is at least the number of spiders placed entirely within that region and
3. all of the entirely shaded regions represent sets whose cardinality is at most the number of spiders with a foot in that region.
4. the elements represented by the spiders obey the ordering imposed on them by the spiders' feet.

We now make this notion precise.

Definition 4.6. *Let $I = (U, \Psi, <)$ be an interpretation and let d ($\neq\bot$) be a unitary spider diagram of order. Then I is a **model** for d if and only if the following conditions hold.*

1. ***The missing zones condition*** $\bigcup_{z \in MZ(d)} \Psi(z) = \emptyset$.
2. ***The function extension condition*** *There exists an extension of Ψ to spiders, $\Psi : \mathcal{L} \cup S(d) \to \mathbb{P}U$ which ensures the following further conditions hold.*
   (a) ***The habitats condition*** *All spiders represent elements (strictly, singleton sets) in the sets represented by their habitats:*
   $$\forall s \in S(d)\ \Psi(s) \subseteq \Psi(habitat(s)) \wedge |\Psi(s)| = 1.$$
   (b) ***The distinct spiders condition*** *Distinct spiders denote distinct elements:*
   $$\forall s_1, s_2 \in S(d) : \Psi(s_1) = \Psi(s_2) \implies s_1 = s_2.$$

(c) ***The shading condition*** *Shaded regions represent sets containing elements denoted by spiders:*

$$\Psi(ShZ(d)) \subseteq \bigcup_{s \in S(d)} \Psi(s).$$

(d) ***The order condition*** *The ordering information provided by the spiders agrees with that provided by $<$: there exists a foot selection function, $FootSelect\colon S(d) \to \mathcal{F}$, for d such that*

- *for all $s \in S(d)$, $FootSelect(s) = (n, z)$ implies $\Psi(s) \subseteq \Psi(z)$*
- *for all $s_1, s_2 \in S(d)$ with $FootSelect(s_1) = (n_1, z_1)$ and $FootSelect(s_2) = (n_2, z_2)$, if $n_1 \neq n_2$ then either*
  - *i. $n_1 < n_2$ and $x < y$ where $\Psi(s_1) = \{x\}$ and $\Psi(s_2) = \{y\}$ or*
  - *ii. $n_2 < n_1$ and $y < x$ or*
  - *iii. $n_1 = \bullet$ or*
  - *iv. $n_2 = \bullet$.*

*If $\Psi\colon \mathcal{L} \cup S(d) \to \mathbb{P}U$ ensures that the above conditions are satisfied then Ψ is a **valid** extension to spiders for d. A foot selection function, $FootSelect\colon S(d) \to \mathcal{F}$, that ensures the above conditions are satisfied is also called **valid**. If $d = \bot$ then the interpretation is not a model for d.*

For compound diagrams, the definition of a model extends in the obvious inductive way.

Theorem 4.1. *Let d be a unitary spider diagram. Then d has a model.*

Proof. (Sketch) Take $U = S(d)$ and any foot selection function $FootSelect\colon S(d) \to \mathcal{F}$ for d. Using $FootSelect$, define $\Psi\colon \mathcal{L} \to \mathbb{P}U$ ensuring that $\Psi(l)$ contains precisely the set of spiders whose selected foot lies in a zone contained by l, whenever l is in $L(d)$. Further, use $FootSelect$ to define $<$ in the obvious way: for spiders s_1 and s_2, $s_1 < s_2$ if and only if both of the feet selected for s_1 and s_2 have integer labels, n_1 and n_2 respectively, and $n_1 < n_2$. □

5 Regular Languages and Spider Diagrams of Order

The finite models of spider diagrams of order give rise to words. We take our alphabet Σ to be a finite set of zones, $(a, b) \in \mathcal{Z}$, such that $a \cup b = \mathcal{L}$ and further assume that all unitary diagrams have a label set $\mathcal{L}$ (all unitary diagrams are semantically equivalent to another unitary diagram with label set $\mathcal{L}$, therefore expressiveness is not affected). Each spider s in a unitary diagram d is said to *give rise to* a letter in word w by selecting a foot of s using some fixed $FootSelect$ function for d. An unordered foot $(\bullet, z_i) = FootSelect(s)$ specifies that the letter z_i appears in w. An ordered foot of the form $(\boxdot, z_i) = FootSelect(s_1)$ specifies that z_i appears at a position in w before the letter z_j of an ordered foot of the form $(\boxdot, z_j) = FootSelect(s_2)$, and so-forth.

For example, the diagram d_1 in figure 4(a) has a model $U = \{x, y, z\}$ where A represents the set $\{x, y\}$ and B represents the set $\{y, z\}$ and $< = \{(x, z)\}$. Taking $Z(d) = \Sigma$, but using the notational convention established in section 3 rather than the formal syntax, then the following words

$$[A\overline{B}][AB][\overline{A}B], [A\overline{B}][\overline{A}B][AB]$$

arise from this model. By contrast, $\overline{A}AB\overline{A}\overline{B}A$ does not arise from this model for d_1 as there must be an occurrence of the letter $\overline{A}B$ at some index in w after the letter $A\overline{B}$ as $(x, z) \in <$. In other words if $A\overline{B}$ were at index 1 (the first letter) then $\overline{A}B$ must occur at index 2 or 3. Conversely, given a word, we want to establish whether it arises from a model for some given diagram. We say such words *conform* to the diagram.

Definition 5.1. *Let $w = w_1w_2...w_n$ be a word in $\Sigma = Z(d)$ and d ($\neq\perp$) be a unitary diagram. The bag (or multiset) of letters of which w consists is denoted bag(w). The word w **conforms**, to d if and only if there exists a function, $f : S(d) \to (\mathbb{Z}^+ \cup \{\bullet\}) \times bag(w)$ satisfying*

1. *$f(s)$ is a foot of s,*
2. *f is injective with regard to $bag(w)$: for all $(n_1, w_i), (n_2, w_i) \in im(f)$, $n_1 = n_2$,*
3. *f is bijective with regard to $bag(w)$ when the image is restricted to the maximal sub-bag of w whose elements are shaded zones in d, denoted $shadebag(w)$: for all $w_i \in shadebag(w)$ there exists $s \in S(d)$ such that $f(s) = (n, w_i)$ for some n.*
4. *for all $s_1, s_2 \in S(d)$, if $f(s_1) = (n_1, w_i)$ and $f(s_2) = (n_2, w_j)$ are distinct feet and $n_1 < n_2$ then the $i < j$*

For $d = \perp$, no words in Σ^* conform to d.

So, w conforms to unitary diagram $d(\neq\perp)$ provided, for each spider, s, in d,

1. each spider in d gives rise to a letter in w by way of selecting a foot,
2. a low ranked foot gives rise to a letter appearing at a lower index of w than a letter corresponding to a higher ranked foot,
3. for each shaded zone, z, the number of occurrences of z in w is precisely the number of spiders whose selected foot is z.

Definition 5.2. *The **language of a unitary spider diagram of order**, denoted $\mathscr{L}(d)$, is the set of words which conform to the diagram d.*

Definition 5.3. *When considering the language of a compound spider diagram of order:*

- $\mathscr{L}(d_1 \wedge d_2) = \mathscr{L}(d_1) \cap \mathscr{L}(d_2)$,
- $\mathscr{L}(d_1 \vee d_2) = \mathscr{L}(d_1) \cup \mathscr{L}(d_2)$,
- $\mathscr{L}(\neg d_1) = \Sigma^* - \mathscr{L}(d_1)$.

We can describe the language of d_1 in figure 4(a) in the form $k \sqcup \Sigma^*$ where k is the finite set of words generated only by spiders i.e.

$$k = \{[A\overline{B}][AB][\overline{A}B], [AB][A\overline{B}][\overline{A}B], [A\overline{B}][\overline{A}B][AB]\}$$

As d_1 contains no shading or missing zones there are no letters prevented from being in words of $\mathscr{L}(d_1)$, thus $\mathscr{L}(d_1) = k \sqcup \Sigma^*$.

Theorem 5.1. *The set of languages for spider diagrams of order whose unitary parts contain no shaded zones is the boolean closure of shuffle-ideal languages, that is level 1 of the Straubing-Thérin hierarchy.*

Proof. (Sketch) Let l be a level 1 language. Then l is a boolean combination of shuffle ideals. We show that each shuffle ideal is the language of a spider diagram. The result that l can be represented then follows by the inductive construction of l and the spider diagram language. An arbitrary shuffle-ideal language can be seen as the finite disjunction:

$$k \sqcup \Sigma^* = (k_1 \sqcup \Sigma^*) \cup (k_2 \sqcup \Sigma^*) \cup \ldots \cup (k_n \sqcup \Sigma^*)$$

where $k_1, k_2, \ldots, k_n$ is a partition of k where each $|k_i| = 1$ and $\Sigma \subseteq \mathcal{Z}$. We may then draw a spider diagram of order for each k_i with zone set Σ and one single-foot spider for each letter in k_i placed in the appropriate zone and having the appropriate rank. The language of the disjunction of these diagrams is $\mathscr{L}(k \sqcup \Sigma^*)$. Conversely, we must show each such spider diagram has a level 1 language, which is left to the reader. □

6 Conclusion

The main contributions of this paper are two-fold. First, we introduced spider diagrams of order, increasing the expressiveness of the spider diagram language. We then defined the language of a spider diagram of order and established the set of such languages includes all of level 1 of the Straubing-Thérin hierarchy. This builds upon the relationships previously established in [3]. The exact relationship between spider diagrams of order and the Straubing-Thérin hierarchy remains to be determined. We conjecture that there are languages in the class $3/2$ that are not the language of any spider diagram of order. In the future we wish to endow spider diagrams with the full power of monadic first order logic with equivalence of order. We further plan to establish whether spider diagrams are a natural specification technique for use in trace analysis.

Acknowledgement: Gem Stapleton is supported by a Leverhulme Trust Early Career Fellowship and by UK EPSRC grant EP/E011160/1 for the Visualization with Euler Diagrams project.

References

[1] R. Clark. Failure mode modular de-composition using spider diagrams. In *Proc. of Euler Diagrams 2004*, volume 134 of *Electronic Notes in Theoretical Computer Science*, pages 19–31, 2005.

[2] R. DeChiara, U. Erra, and V. Scarano. A system for virtual directories using Euler diagrams. In *Proc. of Euler Diagrams 04*, volume 134 of *Electronic Notes in Theoretical Computer Science*, pages 33–53, 2005.

[3] Aidan Delaney and Gem Stapleton. On the descriptional complexity of a diagrammatic notation (to appear). In *Visual Languages and Computing*, 2007.

[4] Volker Diekert and Yves Métivier. *Partial Communication and Traces*, pages 457–531. Springer-Verlag New York, Inc., 1997.

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

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

[7] Pierre-Cyrille Héam. On shuffle ideals. *Theoretical Informatics and Applications*, 36:359–384, 2002.

[8] J. Howse, G. Stapleton, and J. Taylor. Spider diagrams. *LMS J. of Computation and Mathematics*, 8:145–194, 2005.

[9] J. Lovdahl. *Towards a Visual Editing Environment for the Languages of the Semantic Web*. PhD thesis, Linkoping University, 2002.

[10] Jean-Eric Pin. *Syntactic semigroups*, pages 679–746. Springer-Verlag New York, Inc., 1997.

[11] Jean-Eric Pin and Pascal Weil. Polynomial closure and unambiguous product. *Theoretical Computer Science*, 30:1–39, 1997.

[12] Alexander Rabinovich. Star free expressions over the reals. *Theor. Comput. Sci.*, 233(1–2):233–245, 2000.

[13] Kai Salomaa and Sheng Yu. Synchronization expressions and lanugages. *J. of Universal Computer Science*, 5(9), 1999.

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

[15] G. Stapleton and J. Howse. Enhancing the expressiveness of spider diagram systems. In *Proc. of Distributed Multimedia Systems, Intl. Workshop on Visual Languages and Computings*, pages 129–138. Knowledge Systems Institute, 2006.

[16] G. Stapleton, S. Thompson, J. Howse, and J. Taylor. The expressiveness of spider diagrams. *J. of Logic and Computation*, 14(6):857–880, 2004.

[17] N. Swoboda and G. Allwein. Heterogeneous reasoning with Euler/Venn diagrams containing named constants and FOL. In *Proc. of Euler Diagrams 2004*, volume 134 of *ENTCS*. Elsevier Science, 2005.

[18] J. Thièvre, M. Viaud, and A. Verroust-Blondet. Using euler diagrams in traditional library environments. In *Euler Diagrams 2004*, volume 134 of *ENTCS*, pages 189–202. ENTCS, 2005.

[19] Wolfgang Thomas. *Languages, automata, and logic*, pages 389–455. Springer-Verlag New York, Inc., 1997.

Fast Zone Discrimination

Robin Clark*
R&D Department
Energy Technology Control
25 North Street, Lewes, BN7 2PE, Great Britain

Abstract

This paper describes an algorithm (Fast Zone Discrimination - FZD) for analysing Concrete Euler diagrams and listing all present zones.

One application area of Euler/Spider diagrams, is modelling failure modes in electronic circuits. For this a procedure is required to check Spider diagrams for logical consistency by ensuring that all present zones in a model have been considered. Unused zones could be viewed as un-handled failure conditions. In order to know which zones have not been used, a complete list of present zones is required for each concrete diagram drawn.

To determine if zones are present, a concrete diagram must be examined using programmatic area operations. These area operations are costly of computer time and it is desirable to eliminate all that are unnecessary.

The algorithm initially builds two sets of relationships from a concrete diagram and then uses these to target searches for zones that may be present. Using the two sets of relationships eliminates checking for a large number of missing zones; thus processing diagrams quickly and efficiently.

Keywords: Euler, Fast Zone Discrimination, present, available region, java area, algorithm.

1 Introduction

Definition 1.1 *An Euler diagram is a finite set of of simple closed curves in the plane.*

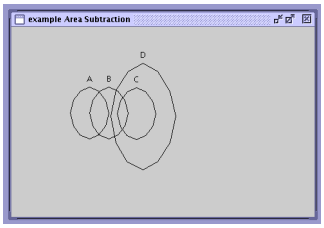
In earlier work[1] spider diagrams [2] have been used to represent the failure modes of components and modules within safety critical electronic systems. By using logical reduction and hierarchies of abstraction, mathematical modelling of complete safety critical systems is possible.

Spider diagrams are based on Euler diagrams. This paper looks specifically at determining which zones are present in an Euler diagram.

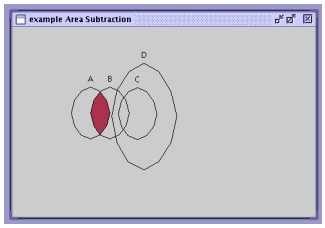
A zone is defined as a region in the plane formed by the intersection of curves in the set I (the 'included', or inside set) and a set of curves E, representing the curves excluded from the set. For instance in figure1(a) there is a zone described by $I = \{B, C, D\}$ with $E = \{A\}$.

One way of looking for present zones would be to look for every possible combination and then to use the excluded zones to check for obscuration. Checking all possible combinations is henceforth referred to as the 'Brute force method'.

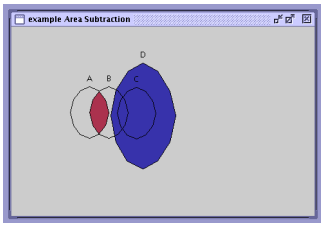
*r.clark@energytechnologycontrol.com



(a) Euler Diagram



(b) Intersection A B



(c) area to subtract

Figure 1: Simple Euler Diagram

The brute force method is simple, and would be practical for small numbers of contours. However as constraint/spider diagrams become used in practice larger numbers of contours and very large diagrams will become common. The Brute force method for finding present zones is of the complexity order $NC.2^{NC}$ (where NC is the number of contours in the Euler diagram). Because of this, a more efficient algorithm has been sought.

In order to obtain information from the concrete diagram, a Java class called an Area[3] is used. This provides Area operations such as *exclusive or* and *subtract*. To measure the effectiveness of a zone searching algorithm, the number of Area operations is considered an appropriate metric.

Section 2 defines determining 'present' zones, and provides Euler diagram examples showing checking for the existence and obscuration of zones. Section 3 defines relations between contours in an Euler diagram, and introduces the terms 'pure intersection', 'pure intersection chain' and 'enclosure'. The mathematical properties of these relationships are then discussed and defined. Section 4 shows how these relationships can be used to draw graphs representing Euler diagrams, and discusses a practical algorithm implemented from spanning these graphs. Section 5 compares the performance of the algorithm with the 'brute force' method.

2 Determining Missing and Present Zones

A 'present' zone is simply one on which one can place an object (or spider 'foot'). For instance there may be an area of intersection on a diagram that is obscured by other contours. That intersection is impossible to use and therefore not considered 'present' in the diagram.

Actually determining whether a zone is present or not in a simple diagram is easy by inspection. An example follows describing two zones and how they are proved 'present' or otherwise, using 'Area'[3] operations.

2.1 Zone Present Example

Let us examine an intersection and determine whether or not it is 'present' in the diagram. In figure 1(a) the intersection $A \cap B$ can be observed. To define this intersection we can say that it has a set of included contours $I_n = \{A, B\}$ and a set of excluded Contours $E_n = \{C, D\}$ (where n is the nth zone under investigation).

By looking at the area formed (figure 1(b))we can see that the intersection exists. The area of all other contours in the diagram must now be subtracted from the intersection to check for obscuration (see figure 1(c)). After subtraction of the areas formed by $C \cup D$ there is an area remaining of the intersection $A \cap B$.

The Zone $A \cap B$ is therefore present in this diagram.

Definition 2.1 (Obscuration) *A zone Z formed from the intersection of contours $c_1 ... c_j$ is said to be obscured by a collection of contours $o_1 ... o_k$ when*

$$\bigcap_{i \in 1...j} Interior(c_i) \subset \bigcup_{i \in 1...k} Interior(o_i)$$

2.2 Zone Missing Example

Consider the potential zone $C \cap B$. This would have an included set $I_N = \{B, C\}$ and an excluded set $E_N = \{A, D\}$. It can be seen that subtracting the interior formed by $A \cup D$ from the interor formed by $B \cap C$ as an area operation; leaves nothing. The zone $B \cap C$ is therefore considered missing in this diagram.

2.3 The General Case : Proving a Zone is Present

The total number of contours in the diagram, will be referred to as NC.

For some diagram elements the contours will not interact and therefore searching for zones can be applied within subsets of contours. These subsets are defined later in the paper. The variable, Interacting contours count, ICC_n represents the number of contours within these subsets. It will always be less than or equal to the number of contours in the diagram.

$$ICC_n \leq NC \tag{1}$$

A zone can be defined by two disjoint contour sets, included I_n and excluded E_n, where n is the zone under investigation.

First of all we determine the concrete area formed by the intersection set I_n. This involves taking the intersections of all the interiors of the sets in I_n, as area operations.

$$AreaIntersection_n = \bigcap_{x \in I_n} Interior(x) \tag{2}$$

The result of these intersection area operations could be a NULL area, indicating that there is no intersection. If the intersection does exist we then need to ensure that it is not covered up by any of the contour interiors formed by the set E_n.

In order to check for obscuration,we must find the area formed by all other sets that could cover it. This is done by taking the union of all the excluded contours, as an area operation.

$$AreaExclusion_n = \bigcup_{x \in E_n} Interior(x) \tag{3}$$

The $AreaIntersection_n$ may now have the $AreaExclusion_n$ subtracted from it, as an area operation. This is an area subtraction and area is only subtracted where it overlaps. If the result has non zero area, then the zone is considered present.

$$RemainingIntersection_n = AreaIntersection_n - AreaExclusion_n \quad (4)$$

Finally the sets I_n and E_n must contain all contours for the the Euler diagram under investigation.

$$InteractingContours = I_n \cup E_n \quad (5)$$

For reference a table of all possible zones, showing which are present in the four contour diagram (figure 1(a)), is shown below.

Inside	*outside*	
Included set I	*Excluded set E*	*Present*
{ }	{ D C B A }	Y
{ A }	{ D C B }	Y
{ B }	{ D C A }	Y
{ B A}	{ D C }	Y
{ C }	{ D B A }	N
{ C A }	{ D B }	N
{ C B }	{ D A }	N
{ C B A }	{ D }	N
{ D }	{ C B A }	Y
{ D A }	{ C B }	N
{ D B }	{ C A }	Y
{ D B A}	{ C }	N
{ D C }	{ B A }	Y
{ D C A }	{ B }	N
{ D C B }	{ A }	Y
{ D C B A }	{ }	N

3 Relationships that can be obtained from an Euler Diagram

Intersection areas in the concrete diagram can be formed in two ways. By enclosing another contour, or by overlapping a part of another contour. These two types of intersection are clearly mutually exclusive.

The intersections that do not involve enclosure have been termed 'pure intersections'.

The algorithm developed in this paper applies two initial searches to the diagram. The first looks for enclosure relationships and the second for pure intersections. These searches are applied to the cross products of all contours in the diagram.

Definition 3.1 (Enclosure) *we say that contour A encloses contour B if* $Interior(A) \supset Interior(B)$

Enclosure for a given pair of contours is expressed in the boolean equation 6.

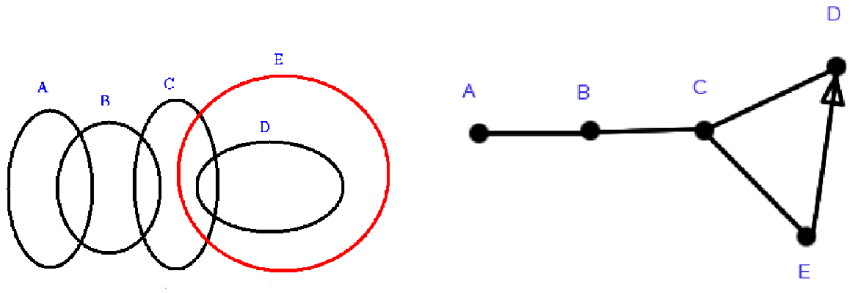
$$enc(a, b) = (Interior(a) \supset Interior(b)) \quad (6)$$

Definition 3.2 (Pure Intersection) *we say that there is a pure intersection of* a, b*, where there is an intersection* $a \cap b$ *but* a *does not enclose* b*, and* b *does not enclose* a*.*

Pure intersection for a given pair of contours is expressed in the boolean equation 7 using the definition of enclosure above.

$$pi(a, b) = ((Interior(a) \cap Interior(b) \neq \emptyset)) \wedge \neg enc(a, b) \wedge \neg enc(b, a) \wedge a \neq b \quad (7)$$

Because the definition of pure intersection expressly forbids enclosure, we have



(a) Euler Diagram (b) Graph of diagram

Figure 2: Pure Intersection Chain with Enclosure

Lemma 3.1 (Mutually exclusive) *Pure intersection and Enclosure are mutually exclusive.*

3.1 Relationship Properties

By applying the equations 6 and 7 to the cross product of all contours in the concrete diagram, we have two lists of relationships, the pure intersections and the enclosures.

From the diagram in figure 2(a), we obtain the following relationships.

3.1.1 Pure intersections relations

Pure Intersection relationships for the diagram in figure 2(a) are :-

$$A \xrightarrow{pi} B, B \xrightarrow{pi} A, B \xrightarrow{pi} C, C \xrightarrow{pi} B, D \xrightarrow{pi} C, C \xrightarrow{pi} D, E \xrightarrow{pi} C, C \xrightarrow{pi} E$$

3.1.2 Enclosure Relations

Relationships for the diagram in figure 2(a) include one enclosure within a pure intersection chain.

$$E \xrightarrow{enc} D$$

Examining these relations, we can classify them.

3.2 Transitive

For pure intersection relationships, it does not follow that (looking at figure 2(a)) because A purely intersects with B, and B with C that A purely intersects with C.

$$A \xrightarrow{pi} B \wedge B \xrightarrow{pi} C \not\Rightarrow A \xrightarrow{pi} C$$

.

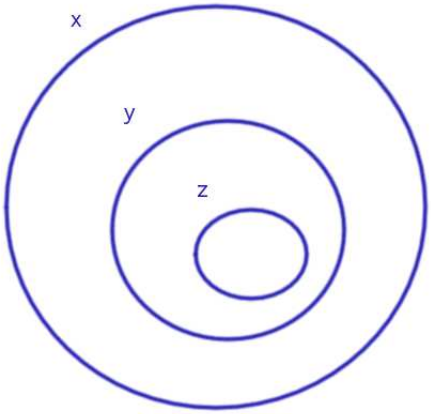


Figure 3: Enclosure within Enclosure : A transitive relationship

Pure Intersection is therefore not a transitive relationship.

Figure 3 demonstrates the transitive nature of enclosure relationships. For the contours in the diagram $\{x, y, z\}$ it can be seen that because x encloses y, and y encloses z, x encloses z. Enclosure is based on a relationship of proper subsets of areas, and is therefore a transitive relationship.

$$x \xrightarrow{enc} y \ \wedge \ y \xrightarrow{enc} z \ \Rightarrow \ x \xrightarrow{enc} z \tag{8}$$

3.3 Reflexive

A reflexive relationship is one where an element can be related to itself [4]. Clearly enclosure and pure intersection are anti-reflexive (i.e. no pure intersection or enclosure can be reflexive) because they require interactions between contours.

3.4 Symmetric

A symmetric relation is one such that $a \rightarrow b \Rightarrow b \rightarrow a$ [4]. Clearly enclosure cannot be symmetric. Because pure intersection is defined by sharing an intersection (see eqn 7, i.e.without enclosure), pure intersection relations are always symmetric. The relationships defined for the pure intersections above (3.1.1), can now be expressed thus.

$$A \overset{pi}{\leftrightarrow} B, B \overset{pi}{\leftrightarrow} C, D \overset{pi}{\leftrightarrow} C, C \overset{pi}{\leftrightarrow} E$$

3.5 Pure Intersection Relationship Properties

Pure intersection relationships are

- Not Reflexive
- Symmetric

Note that by following pure intersection relationships, sets of contours connected by pure intersections can be determined. These are referred to as ‘pure intersection chains’.

Contours in the pure intersection chain may obscure other members in the same chain (see figure 2(a)). Formally a ‘pure intersection chain’ is defined thus

Definition 3.3 (Pure Intersection Chain) *Let d be an Euler diagram : a pure intersection chain is a maximal set of contours C in d such that for any pair of contours in C there exists a sequence of contours such that*

$$c_i \xrightarrow{pi} c_n \; for \; i = 1, ..., n-1$$

3.6 Enclosure Relationship Properties

Enclosure relationships are

- Not Reflexive
- Not Symmetric
- Transitive

4 Methods for finding Present Zones

Firstly we can consider the diagram in terms of pure intersection relationships. The effects of enclosure and obscuration are dealt with later.

Lemma 4.1 (pure intersection cases) *When analysing an Euler diagram from pure intersection relations, there are only three possible cases that can be presented. Lone contours, lone purely intersected contours and pure intersection chains.*

4.1 3 cases for zone identification

The three cases are described in greater detail below.

4.1.1 Lone Contours

Definition 4.1 (Lone Contour) *A Lone Contour is a countour not intersected by any other.*

A lone contour will always represent one present zone, which may include enclosing contours (if any). The contours in figure 3, are all lone contours, and analysing these determines the following present zones.

Lemma 4.2 (Lone Contour Single Zone) *A lone contour will always produce one present zone in an Euler diagram.*

lone contour under investigation	*Included Set* I_n	*Excluded Set* E_n
$\{z\}$	$\{xyz\}$	$\{\}$
$\{y\}$	$\{xz\}$	$\{y\}$
$\{x\}$	$\{x\}$	$\{yz\}$

4.1.2 Lone Pure Intersection Pairs

Definition 4.2 (Lone Pure Intersection Pair) *A Lone Pure Intersection is a pair of intersecting contours, not belonging to a pure intersection chain.*

The pure intersections in figure 6(a) are all lone pure intersections.

Lone pure intersections may be enclosed by other contours (see eqn 8). Three present zones will always be found upon analysing a lone pure intersection. Enclosing contours (if any) are added to the intersection sets I_n.

Lemma 4.3 (Lone Pure Intersection 3 zones) *A lone pure intersection will always produce three present zones in an Euler diagram.*

4.1.3 Pure Intersection Chains

Pure intersections can contain zones formed by multiple intersections, they can have contours within the chain that obscure zones, and they may contain enclosure within the chain.

In describing the process for finding the present zones within a pure intersection chain it helps to graph them, with the contours becoming vertices, enclosures forming directed edges and pure intersections forming undirected edges.

Circuits of undirected edges indicate the possibility of multiple intersections within the chain. The graphs and their meanings are dealt with in the next section.

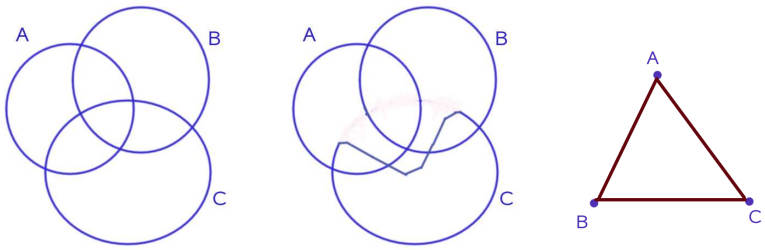
Pure intersection chains within a diagram can be viewed as separate groups within the Euler diagram. That is to say, they may be analysed in isolation with the enclosure rules being applied afterward. This means that a diagram can be broken down into smaller more manageable chunks and this significantly reduces the number of area operations to perform (see eqn 1). The reasons for this are described in the section 4.4. (i.e. area operations need only be performed within pure intersection chains). Thus the number of contours in the diagram NC, comprises of smaller sets of contours (of size ICC_n) that can be analysed separately. See eqn 1.

Contours may enclose pure intersection chains. Where this is the case all enclosing contours (see eqn 8) are added to the intersection sets I_n of all zones discovered within the pure intersection chain.

4.2 Graphing Pure Intersection and Enclosure Relationships

By representing contours as vertices, enclosure relationships as directed edges (because they are transitive) and pure intersections as undirected edges (because they are symmetric), the diagram in figure 2(a) can be represented by the graph in figure 2(b).

4.3 Graphs with Circuits



(a) Triple zone (b) No Triple Zone (c) Graph

Figure 4: Graph with Circuit

Ignoring enclosure for the time being, consider the Euler diagrams in figures 4(a) and 4(b) .

These both produce the same graph representation see figure 4(c).

Note that the existence of a circuit of undirected edges (i.e. pure intersections) indicates the possibility of a multiple intersection. If the area operations in equations 2, 3, 4 and 5 are satisfied the multiple area is 'present'.

Note the corollary of this, *if no circuit exists then no multiple intersection can*. The principle strength of the algorithm hinges on this. By targeting the searching to only zones that 'may' be present, all those that cannot are by-passed.

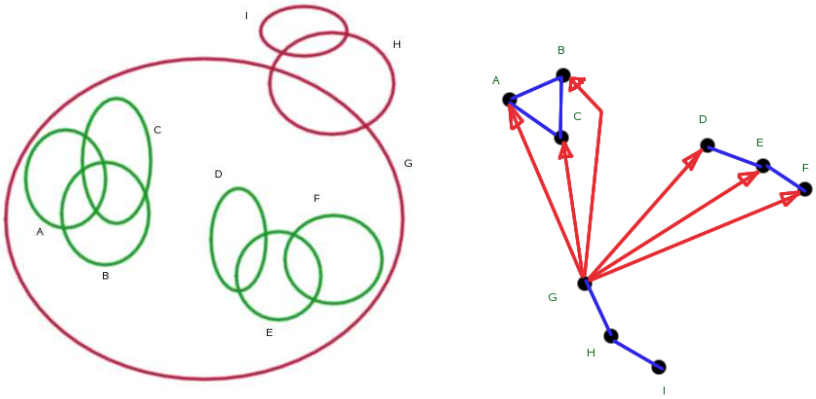
Note that the circuits within the pure intersection chain can be investigated, and then the enclosures can be added afterward. This is because 'pure intersection' and 'enclosure' are mutually exclusive. Also the transitive relationship established for enclosure, means that we simply have to add enclosure intersections to the included set I_N (see eqn 8).

Lemma 4.4 (Venn N circuit) *For a Venn N zone to exist there must be a circuit in the corresponding graph of pure intersections with N vertices, where each vertex corresponds to a contour in the Venn N intersection.*

Lemma 4.5 (Venn N pure intersection chain) *For a circuit of pure intersections to exist, they must be all members of the same pure intersection chain.*

4.4 Non connected graphs

Consider the Euler diagram in figure 5(a).



(a) Separate Pure Intersection Chain

(b) Graph of Separate pure int.chains

Figure 5: Non-Connected Graphs

There are three separate pure intersection chains in this diagram. They are $\{A, B, C\}$, $\{D, E, F\}$ and $\{G, H, I\}$. Note that present zones found from the pure intersection chains $\{A, B, C\}$ and $\{D, E, F\}$ all include an intersection with contour G.

By finding present zones within the pure intersection chains, and adding any enclosing contours (see eqn 8) to the intersection set I_N the present zones germane to the pure intersection chain are discovered.

A more complicated scenario is when enclosure occurs within a pure intersection chain, see figure 2(a). In analysing the Intersection $C \cap D$ it will not be recognised as an present zone, because the enclosure relationship $E \xrightarrow{enc} D$ will be applied and the intersection becomes $C \cap D \cap E$: after passing obscuration testing (area subtraction of $A \cup B$), the zone with $I_N = \{C, D, E\}$ and $E_N = \{A, B\}$ will be registered as a 'present'

4.5 Algorithm Design

The aim of this algorithm is to avoid the burdensome 2^{NC} complexity order of checking for all possible zones in an Euler diagram with NC contours.

By breaking the diagram into a number of smaller sets of contours, which can be checked in isolation, the number of checks is significantly reduced.

The smallest possible sets that can be analysed in isolation are the 'lone contour', the 'lone pure intersection' and the 'pure intersection chain'. A 'lone contour' will always produce one present zone (see lemma 4.2). A 'lone pure intersection' will always produce 3 (see lemma 4.3).

A 'pure intersection chain' can potentially produce 2^{ICC_n} present zones (where ICC_n is the number of contours in the chain).

One could check for all 2^{ICC_n} possible zones within the 'pure intersection chain'. However, the 'pure intersection chain' is handled more effeciently than this by only applying checks to circuits of pure intersections within the chain (see lemma 4.4).

By applying the enclosure relations to the present zones discovered in each of the three cases, all present zones in the diagram are discovered.

The correctness of the algorithm rests on the lemmas 3.1, 4.1, 4.2, 4.3 and 4.4.

4.6 Algorithm Pseudo Code

In high level pseudo code, the algorithm works thus:

```
BEGIN
 Determine all Enclosure relationships.
 Determine all Pure intersections
 Search through pure intersection relationships and obtain pure
 intersection chains.

 For all contours in the diagram where contour is not a member
    of a pure intersection chain; add all enclosing contours;
    register as an present zone;

 For all pure intersections in the diagram where pure
    intersection is not a member of a pure intersection chain;
    add all enclosing contours to intersections register as an
    present zone;

 For all pure intersection chains
   for each pair intersection within chain add any enclosing
   contours and determine if the intersection is present using
   area operations; if present then; register as a present zone;
```

```
   obtain all circuits within the pure intersection chain;
   for all circuits within the pure intersection chain;
      add any enclosing contours and check obscuration using
      area operations;
      if the zone indicated is present; then register as an
      present zone;
END
```

4.7 Checking for all Possible zones - Brute force/Binary Count

The number of potential zones in an Euler diagram containing NC contours is 2^{NC}.

In a diagram with NC contours, each zone under investigation will comprise of the set I_n and the set E_n. The number of area operations to get the intersection area, and the number to get the obscuration check area, will always add up to NC, using the 'brute force' method.

For instance were a diagram to contain 32 contours, to brute force check for the existence of all contours would take all possible combinations of 32 objects. This corresponds to a binary count and thus 2^{32} possible zones to check for. A diagram with 32 contours would contain a potential of over 4 billion zones. Multiply that by the 32 area operations (with varying proportions of intersection (I_n) and obscuration (E_n) tests - but always adding up to 32) required and we reach an astronomical number (32.2^{32}).

In general then the brute force zone search, with intersection area operations, and obscuration testing (on average $\frac{NC}{2} + \frac{NC}{2}$), takes

$$NC.2^{NC} \tag{9}$$

4.8 Number of Area Operations using Pure intersection and Enclosure Relationships

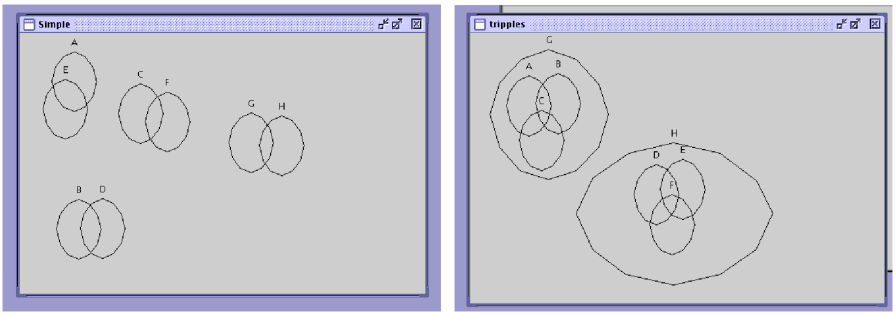
The number of area operations applied by this algorithm depends upon the complexity of the diagram, and the sizes of the pure intersection chains. Were the worst case to apply, i.e.$Venn_{NC}$ this algorithm is less efficient by $2.NC^2$ i.e. for a Venn N diagram, the number of area compares for the FZD algorithm is

$$2.NC^2 + NC.2^{NC} \tag{10}$$

Most diagrams written by human beings will be far less complicated than $VennN$. In the domain of safety critical circuit/system analysis, the diagrams will be comprised typically of a number of separate pure intersection chains, and the searches need only be applied within them. The ICC_N value for interacting contours will be equal to the number of contours in each pure intersection chain. Also because the graphs are traversed, *most contour combinations will be determined impossible by the fact that no circuit exists in the undirected edges*.

5 Case Studies

To compare the algorithms performance against the 'brute force' method, I have taken two diagrams of a typical complexity level that is drawn from the failure mode[1]



(a) Simple Pure Intersection Pairs (b) Two Enclosed Venn 3

Figure 6: Performance Comparison

applications.

The number of area operations necessary to find all zones has been analysed and a numerical formula based on NC, has been derived using techniques from [5]. This formula can now be plotted to compare the performance of the algorithm for the two test patterns. The examples shown in figures 6(a) and 6(b), use 8 contours per diagram. Using the Brute force method these would require $NC.2^{NC}$ or 2048 area compares to determine all visible zones. By duplicating the structures, values can be calculated for general NC number contour diagrams of the same family. The algorithm parses the relations built in the first two passes to eliminate unnecessary searches. These relationships are held in Java data structures in RAM and are therefore considered to have minimal impact on processing time. For this reason, only the java Area[3] operations are considered in comparing the performance of the algorithm against the 'brute force' method.

5.1 Simple pairs of contours

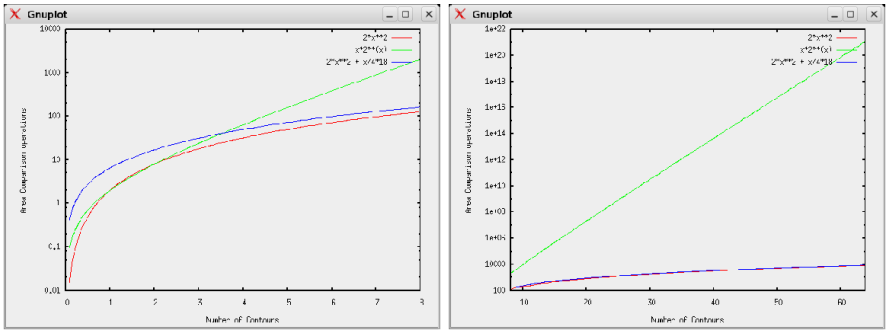
The simple diagram, shown in figure 6(a), consists of four overlapping pairs of contours. To determine the enclosure and pure intersection relations, two cross products of contour area searches are required. Thus 2 NC^2, i.e. 128 searches. Zones derived from lone contours and lone pure intersections do not need to be checked for existence or obscuration. The total number of area compares/operations is therefore $2.64 = 128$. Were one to add more lone pure intersections to this diagram, the diagram would become larger, but would have the same pattern. Five lone pure intersections would take $2.128 = 256$ area operations to find all present zones.

As a general case, for extrapolating larger diagrams of the same pattern, where N is the number of contours

$$AreaOperationsRequired = 2.NC^2 \tag{11}$$

5.2 Two Venn 3 totally Enclosed Once : a more Complex Diagram

The second diagram, see figure 6(b), contains two Venn 3 configurations each enclosed by a contour. Breaking this down, we have two single zones (from the contours G and H). Examining the two Venn3 structures, these require an existence check for the triple intersection (3 area operations). As the number of contours to check for obscuration



(a) 0 to 8 contours (b) 8 to 64 contours

Figure 7: Performance Comparison

against it is 0, they do not require obscuration testing. Within each Venn3 each of the double intersection zones must be checked for obscuration. Thus 2 area operations to construct the shape of the zone, and 1 area operation to test for obscuration, thus 3 per pair. The three single zones in the pure intersection require 1 area operation to construct the shape of the contour, and two to test for obscuration. Thus 3 per pair. This diagram therefore requires $128 + 2.(9 + 9) = 146$ area compares. As a general equation for the number of the number of area operations required can be calculated, thus:

$$AreaOperationsRequired = 2.NC^2 + \frac{NC}{4}.(18) \tag{12}$$

5.3 Extrapolating for N Contour Diagrams

Duplicating the structures in the diagrams in figures 6(b) and 6(a), and using the general case equations (11 and 12), a plot of area searches required against diagram complexity can be drawn.

These graphs were produced in Gnuplot[6] (which uses a Fortran [7] like syntax for formulas), with the following equations:

$GnuplotSyntax$	$LineColour$	$Arithmetic$
$x * 2 * *x$	$Green$	$NC.2^{NC}$
$2 * x * *2 + x/4 * 18$	$Blue$	$2.NC^2 + \frac{NC}{4}.18$
$2 * x * *2$	Red	$2.NC^2$

These graphs clearly shows that the FZD method efficiency increases with the number of contours in a diagram.

6 Conclusion

6.1 Practical Implementation

An algorithm has been implemented in Java, which finds present zones in an efficient and quick way, in a spider diagram editor application. It has been checked against a ‘brute force’ algorithm, by inspection, with Venn4, Venn5 and a variety of test diagrams.

6.2 Future Enhancements

Because Surface Areas are calculated as a side effect of the Java[3] area class, some well formed-ness[8] criteria can be checked for.

Further efficiency may be possible by analysing the structure of the graphs produced from the pure intersection chains, and determining rules to further reduce the number of Java area operations to prove a zone present.

References

[1] R. Clark. Failure mode modular de-composition using spider diagrams. In *Proceedings of Euler Diagrams 2004*, volume 134 of *Electronic Notes in Theoretical Computer Science*, pages 19–31, 2005.

[2] J. Howse, G. Stapleton, and J. Taylor. Spider diagrams. *LMS Journal of Computation and Mathematics*, 8:145–194, 2005.

[3] Sun Micro Systems. Java area operations. Available from http://java.sun.com/j2se/1.3/docs/api/java/awt/geom/Area.html, 2000.

[4] David Tall Ian Stewart. *The Foundations of Mathematics : ISBN 0-19-853165-6*. Oxford University Press, 1977.

[5] Gregory J.E. Rawlins. *Compared to What ? An introduction to the analysis of algorithms ISBN 0-7167-8243-x*. Computer Science Press, 1991.

[6] Various Open source Project. Gnuplot 4 home page. Available from http://www.gnuplot.info/, 2005.

[7] A. Balfour D.H. Marwick. *Programming in Standard Fortran 77 ISBN 0-435-77486-7*. Heinemann Educational Books, 1979.

[8] Gem Stapleton Peter Rodgers, John Howse. Properties of euler diagrams. http://www.cmis.bton.ac.uk/research/vmg/papers/, 2007.

A Peirce-style calculus for $\mathcal{ALC}$

Frithjof Dau and Peter Eklund *
Faculty of Informatics
University of Wollongong
Wollongong, Australia

Abstract

Description logics (DLs) are a well-understood family of knowledge representation (KR) languages. The notation of DLs has the flavour of a variable-free first order predicate logic. In this paper, a diagrammatic representation of the DL $\mathcal{ALC}$, based on Peirce's existential graphs, is presented, and a set of transformation rules on these graphs is provided. It is proven that these rules form a sound and complete diagrammatic calculus for $\mathcal{ALC}$.

Keywords: Existential graphs, relation graphs, description logics, diagrammatic reasoning, calculus, soundness, completeness

1 Introduction

Description logics (DLs) are a well-understood family of knowledge representation (KR) languages tailored to express knowledge about concepts and concept hierarchies that have gained widespread use. The basic building blocks of DLs are concepts, roles and sometimes individuals, which can be composed by language constructs such as intersection, union, value or number restrictions and more to build more complex well-formed formulas that themselves represent more complex concepts and roles. For example, if MAN, FEMALE, MALE, RICH, HAPPY are concepts and if HASCHILD is a role, we can define

$$\begin{aligned} \text{HM} \quad \equiv \quad & \text{MAN} \sqcap \exists \text{HASCHILD}.\text{FEMALE} \sqcap \exists \text{HASCHILD}.\text{MALE} \\ & \sqcap \forall \text{HASCHILD}.(\text{RICH} \sqcup \text{HAPPY}) \end{aligned}$$

which defines the concept of men who have both male and female children, and where all children are rich or happy (HM abbreviates HAPPYMAN).

The formal notation of DLs has the flavour of a variable-free first order predicate logic (FOL). In fact, DLs correspond to decidable fragments of FOL. Like FOL, DLs have a well-defined, formal syntax and Tarski-style semantics, and they provide sound and complete inference facilities. The variable-free notation of DLs makes them easier to comprehend than the common FOL formulas that include variables. Nevertheless, without training, the symbolic notation of FOL can be hard to learn and comprehend.

It has been argued that diagrams are useful for KR systems [9, 12, 13], a fact that has been acknowledged by the DLs authors (see introduction of [1]). Therefore one

*Email:dau,peklund@uow.edu.au

alternative to DLs symbolic notation is the development of a diagrammatic representation. A first attempt can be found in [9], where a graph-based representation for the textual DL CLASSIC is elaborated. In [3], a specific DL is mapped to the diagrammatic system of conceptual graphs [17]. In [2], a UML-based representation for a DL is provided. In these approaches, the focus is on a graphical *representation* of DL, however, as emphasized in many works on DL, *reasoning* is a distinguishing feature of DL. Correspondences between graphical representation of the DL and the DL reasoning system are therefore important inclusions in any graphical representation but to date they have remain largely unelaborated.

This paper presents a diagrammatic representation of the DL $\mathcal{ALC}$ in the style of Peirce's existential graphs (EGs) [18, 15, 16, 6] (the reasons for choosing Peirce's graphs as a diagrammatic framework for DL are presented [8]). An adequate diagrammatic calculus for $\mathcal{ALC}$, based on Peirce's calculus for his graphs, is provided.

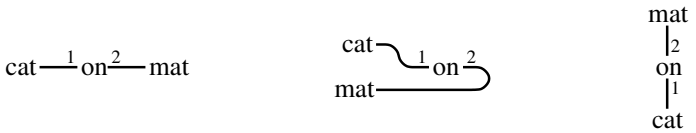
Reasoning with DLs is usually carried out by means of tableau algorithms. The calculus of this paper differs significantly from this approach in two respects. First, the rules of the calculus are *deep-inference* rules, as they modify deep nested subformulas, whereas tableau algorithms (similar to other common calculi) only modify formulas at their top-level (some interesting aspects of Peirce's rules in terms of proof-theory are investigated in [7]). More importantly for this workshop, the rules can be best understood to modify the diagrammatic Peirce-style representations of $\mathcal{ALC}$, i.e., the calculus is a *diagrammatic* calculus.

The paper is structured as follows. We assume that the reader has some familiarity with the system of EGs. In Section 1.1, only a very short introduction is provided. Thorough introductions can be found in [18, 15, 16, 6]. In Section 2 the syntax and semantics of the DL $\mathcal{ALC}$ as we use it in this paper is introduced. In Section 3, the diagrammatic calculus for $\mathcal{ALC}$ is presented, and its soundness and completeness is proven. The final section provides a discussion of this research.

1.1 Existential Graphs

Existential graphs [10] are a diagrammatic logic invented by C.S. Peirce (1839-1914) in the last two decades of his life. Existential graphs are divided into three parts called Alpha, Beta and Gamma. The three parts build on one another, Beta builds upon Alpha, and Gamma builds on both Alpha and Beta. Alpha corresponds to propositional logic, Beta corresponds to FOL (to be precise: first order logic with predicates and equality, but without functions or constants). Gamma encompasses features of higher order logic, including modal logic, self-reference and more. In contrast to Alpha and Beta, Gamma was never finished by Peirce, and even now, only fragments of Gamma (mainly the modal logic part) are elaborated to contemporary mathematical standards. In this section, only Beta is introduced.

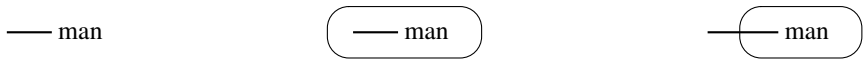
The EGs of Beta consist of predicate names of arbitrary arities, of heavily drawn lines which are both used to express existential quantification and identity, and of closed, doublepoint-free curves which are called CUTS (or sometimes SEPS) and used to negate the enclosed subgraph. We start with a very simple graph expressing 'a cat is on a mat'. Below, three different diagrams of the graph are depicted.



Each diagram contains two heavily drawn lines. In this case, they do not cross cuts or do not have branching points (see below for an explanation of cuts and branching points). This simplest form of a heavily drawn line is called LINE OF IDENTITY. The two lines of identity denote two (not necessarily different) objects. The first line of identity is attached to the unary predicate 'cat', hence the first object denotes a cat. Analogously, the second line of identity denotes a mat. Both lines are attached to the dyadic predicate 'on', i.e. the first object (the cat) stands in the relation 'on' to the second object (the mat). The meaning of the graph is therefore 'there is a cat and a mat such that the cat is on the mat', or in short: A cat is on a mat.

The three different diagrams are different representations of the same EG. In order to distinguish graphs from their diagrams, Peirce coined the term *graph* and *graph replica*, i.e., the above diagrams are different graph replicas of the same EG. Similar distinctions are made between *types* and *tokens*, known from philosophy, or *abstract* and *concrete* syntax, used widely in Computer Science. As discussed in [11, 4], for a formally precise elaboration of any logic by means of diagrams, this distinction is vital. This approach is adopted in this paper as well. In Section 2, a fragment of Peirce's graphs is used as a diagrammatic system for the DL $\mathcal{ALC}$. In this section, the syntax of this fragment is defined on a abstract level which prescinds from the topological properties of the diagrammatic representations.

In the next graphs, the cuts which are used to express negation are introduced.



The meaning of the first graph is 'there is a man'. The second graph is built from the first graph by drawing a cut around it, i.e. the first graph is denied. Hence the meaning of the second graph is 'it is not true that there is a man', i.e. 'there is no man'. In the third graph, the heavily drawn line (which is not a line of identity, as it crosses a cut) begins on the sheet of assertion. Hence, the existence of the object is asserted, not denied. For this reason the meaning of the third graph is 'there is something which is not a man'.

Peirce writes in [10], in 4.116 (we adopt the usual convention to refer to his collected papers), a "line of identity is [. . .] a heavy line with two ends and without other topical singularity (such as a point of branching or a node), not in contact with any other sign except at its extremities." So lines of identity do not have any branching points, nor are they are allowed to *cross* cuts. However, by connecting them at their endpoints, we can obtain networks of lines of identity, which are termed LIGATURES. Peirce allows only two or three lines of identity to be connected. If three lines of identity are connected, the point where they meet is called a BRANCHING POINT. Moreover, it is possible to connect lines of identity connect directly on a cut. Due to this possibility, ligatures are permitted which cross a cut.

Let us now consider the three EGs of Fig. 1, where ligatures are used.

In the first graph, the ligature consists of three lines of identity, which meet in a branching point, in the second graph, the ligature consists of two lines of identity meeting on the cut, and the ligature in the third graph is composed of seven lines of identity. Nonetheless, in all these graphs, a ligature can, similar to a line of identity, be understood to denote a single object. The meaning of the graphs of Figure 1 'there exists a male, human african', 'there exists a man who will not die', and 'it is not true that there is a pet cat such that it is not true that it is not lonely and owned by somebody', i.e., 'every pet cat is owned by someone and is not lonely'.

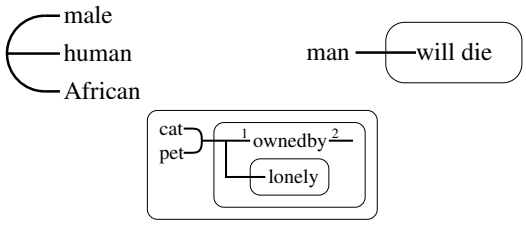


Figure 1: Four Peirce graphs with ligatures which do not traverse cuts

Nonetheless, other examples show that this interpretation of ligatures is not so simple in every case: namely a ligature may stand for more than one object. Let us consider the three EGs of Figure 2. These graphs have the meanings 'there are at least two suns', 'there are (not necessarily distinct) objects which are blue, red, large and small, respectively', and 'the blue and large or the red and small object are distinct', and 'there are objects o_1, o_2, o_3 with the properties S, P, and T resp, and these objects are not all identical' (i.e., $o_1 = o_2 = o_3$ does not hold). In every graphs, there is not a single ligature that can be understood to denote a single object.

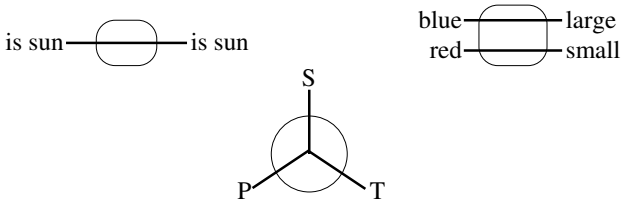


Figure 2: Three Peirce graphs with ligatures which traverse cuts

In every graph in Figure 2 a part of a ligature traverses a cut (i.e., there is a cut c and a heavily drawn line l which is part of the ligature such that both endpoints of l are placed on c and the remainder of l is enclosed by c). Such a device denotes non-identity of the endpoints of l. A complete discussion of ligatures in existential graphs goes beyond the scope of this paper, see [5] for a more detailed discussion. It will turn out that in the EGs we have to deal with in this paper, no ligature traverses a cut, so we will not run into problems caused by the kind of ligatures of we see in Figure 2.

We now have all the necessary elements to express existential quantification, predicates of arbitrary arities, conjunction and negation. As such we see that the Beta part of EGs corresponds to FOL (without object names and without function names). Moreover, Peirce equipped EGs with a set of five sound and complete inference rules. We do not address these rules in this section. For the $\mathcal{ALC}$-fragment of EGs, they will be introduced in Section 3.

2 The Description Logic $\mathcal{ALC}$

The vocabulary $(\mathcal{A}, \mathcal{R})$ of a DL consists of a set $\mathcal{A}$ of (ATOMIC) CONCEPTS, which denote sets of individuals, and a set $\mathcal{R}$ (ATOMIC) ROLES, which denote binary relationships between individuals. Moreover, we usually consider vocabularies that include the universal concept $\top$. From these atomic items, more complex concepts and roles are

built with constructs such as intersection, union, value and number restrictions, etc. For example, if C, C_1, C_2 are concepts, then so are $C_1 \sqcap C_2$, $\neg C$, $\forall R.C$, and $\exists R.C$ (these constructors are called conjunction, negation, value restriction, and exists restriction).

In this paper, we focus on the description logic $\mathcal{ALC}$, which is the smallest propositionally closed description logic. For our purpose, we consider $\mathcal{ALC}$ to be composed of conjunction, negation and existential restriction. In contrast to the usual approach, in this paper the concepts of $\mathcal{ALC}$ are introduced as labelled trees. This is more convenient for defining the rules of the calculus, and the trees are already close to Peirce's notion of graphs.

An INTERPRETATION is a pair $(\Delta^{\mathcal{I}}, \mathcal{I})$, consisting of an nonempty DOMAIN OF THE INTERPRETATION $\Delta^{\mathcal{I}}$ and INTERPRETATION FUNCTION $\mathcal{I}$ which assigns to every $A \in \mathcal{A}$ a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to role $R \in \mathcal{R}$ a relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. We require $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$.

Trees can be formalized either as rooted and acyclic graphs, or as special posets. We adopt the second approach, i.e., a tree is a poset $(T, \geq)$, where $s \geq t$ can be understood as 's is an ancestor of t'. A LABELLED TREE is a structure $\mathbf{T} := (T, \leq, \nu)$, where $(T, \leq)$ is a tree and $\nu : T \to L$ is a mapping from the set of nodes to some set L of labels. The greatest element of T is the ROOT of the tree. As usual, each node v gives rise to a SUBTREE $\mathbf{T}_v$ (formally, $\mathbf{T}_v = (T_v, \geq \big|_{T_v \times T_v}, \nu|_{T_v})$ with $T_v := \{w \in T \mid v \geq w\}$). We write $\mathbf{T}' \subseteq \mathbf{T}$, if $\mathbf{T}'$ is a subtree of $\mathbf{T}$. Isomorphic labelled trees are implicitly identified.

Next we introduce some operations to inductively construct labelled trees. These operations will be used to define the syntax and semantics of $\mathcal{ALC}$ based on labelled trees. We assume to have a set L of labels.

Chain: Let $l_1, \ldots, l_n \in L$. With $l_1\, l_2\, \ldots l_n$ we denote the labelled tree $\mathbf{T} := (T, \geq, \nu)$ with $T := \{v_1, \ldots, v_n\}$, $v_1 > v_2 > \ldots > v_n$ and $\nu(v_1) = l_1, \ldots, \nu(v_n) = l_n$. That is, $l_1\, l_2\, \ldots l_n$ denotes a CHAIN, where the nodes are labelled with l_1, l_2, $\ldots$, l_n, respectively. We extent this notation by allowing the last element to be a labelled tree: If $l_1\, l_2\, \ldots l_n \in L$ and if $\mathbf{T}'$ is a labelled tree, then $l_1\, l_2\, \ldots l_n \mathbf{T}'$ denotes the labelled tree $\mathbf{T} := (T, \geq, \nu)$ with $T := T' \cup \{v_1, \ldots, v_n\}$, $v_1 > v_2 > \ldots > v_n$ and $v_i > v$ for each $i = 1, \ldots, n$ and $v \in T'$, and $\nu := \nu' \cup \{(v_1, l_1), \ldots, (v_n, l_n)\}$. That is, $\mathbf{T}$ is obtained by placing the chain $l_1\, l_2\, \ldots l_n$ above $\mathbf{T}'$.

Subsitution: Let $\mathbf{T}_1, \mathbf{T}_2$ be labelled trees and $\mathbf{S} := (S, \geq_s, \nu_s)$ a subtree of $\mathbf{T}_1$. Then $\mathbf{T} := \mathbf{T}_1[\,\mathbf{T}_2 \,/\, \mathbf{S}\,]$ denotes the labelled tree obtained from $\mathbf{T}_1$ when $\mathbf{S}$ is substituted by $\mathbf{T}_2$. Formally, we set $\mathbf{T} := (T, \geq, \nu)$ with $T := (T_1 - S) \cup T_2$, $\geq := \geq_1 \big|_{T_1 - S} \cup \geq_2 \cup \{(w_1, w_2) \mid w_1 > v,\, w_1 \in T_1 - S,\, w_2 \in T_2\}$, and $\nu := \nu_1\big|_{(T_1 - S)} \cup \nu_2$.

Composition: Let $l \in L$ be a label and $\mathbf{T}_1, \mathbf{T}_2$ be labelled trees. Then $l(\mathbf{T}_1, \mathbf{T}_2)$ denotes the labelled tree $\mathbf{T} := (T, \geq, \nu)$, where we have $T := T_1 \cup T_2 \cup \{v\}$ for a fresh node v, $\geq := \geq_1 \cup \geq_2 \cup (\{v\} \times (T_1 \cup T_2))$, and $\nu := \nu_1 \cup \nu_2 \cup \{(v, l)\}$. That is, $\mathbf{T}$ is the tree having a root labelled with l and which has $\mathbf{T}_1$ and $\mathbf{T}_2$ as (direct) subtrees.

Strictly speaking, in the above operations we have sometimes to consider trees with *disjoint* sets of nodes (for example, we have to assume in $\mathbf{T}_1[\,\mathbf{T}_2 \,/\, \mathbf{S}\,]$ that T_1 and T_2 are disjoint). As we consider trees only up to isomorphism, this can always easily be achieved and is usually not explicitely mentioned.

Using these operations, we can now define the tree-style syntax for $\mathcal{ALC}$.

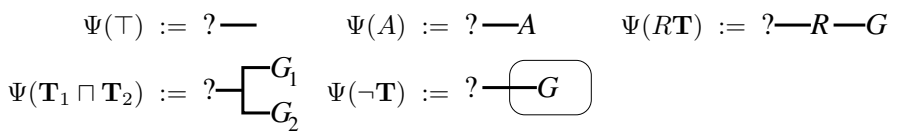
Definition 2.1 *Let a vocabulary $(\mathcal{A}, \mathcal{R})$ be given with $\top \in \mathcal{A}$. Let '$\sqcap$' and '$\neg$' be two further signs, denoting conjunction and negation. Let $(\Delta^{\mathcal{I}}, \mathcal{I})$ be a interpretation for the vocabulary $(\mathcal{A}, \mathcal{R})$. We inductively define the elements of $\mathcal{ALC}^{Tree}$ as labelled trees $\mathbf{T} := (T, \geq, \nu)$, as well as the interpretation $\mathcal{I}(\mathbf{T})$ of $\mathbf{T}$ in $(\Delta^{\mathcal{I}}, \mathcal{I})$.*

Atomic Trees: *For each $A \in \mathcal{A}$, the labelled tree A (i.e. the tree with one node labelled with A), as well as $\top$ are in $\mathcal{ALC}^{Tree}$. According to the interpretation of names in interpretations, we set $\mathcal{I}(A) = A^{\mathcal{I}}$ and $\mathcal{I}(\top) = \Delta^{\mathcal{I}}$.*
Negation: *Let $\mathbf{T} \in \mathcal{ALC}^{Tree}$. Then the tree $\mathbf{T}' := \neg\mathbf{T}$ is in $\mathcal{ALC}^{Tree}$. We set $\mathcal{I}(\mathbf{T}') = \Delta^{\mathcal{I}} - \mathcal{I}(\mathbf{T})$.*
Conjunction: *Let $\mathbf{T}_1, \mathbf{T}_2 \in \mathcal{ALC}^{Tree}$. Then the tree $\mathbf{T} := \sqcap(\mathbf{T}_1, \mathbf{T}_2)$ is in $\mathcal{ALC}^{Tree}$. We set $\mathcal{I}(\mathbf{T}) = \mathcal{I}(\mathbf{T}_1) \cap \mathcal{I}(\mathbf{T}_2)$.*
Exists Restriction: *Let $\mathbf{T} \in \mathcal{ALC}^{Tree}$, let R be a role name. Then $\mathbf{T}' := R\mathbf{T}$ is in $\mathcal{ALC}^{Tree}$. We set $\mathcal{I}(\mathbf{T}') = \{x \in \Delta^{\mathcal{I}} \mid \exists y \in \Delta^{\mathcal{I}} : xR^{\mathcal{I}}y \wedge y \in \mathcal{I}(\mathbf{T})\}$.*

*The labelled trees of $\mathcal{ALC}^{Tree}$ are called $\mathcal{ALC}$-*TREES*. Let $\mathbf{T} := (T, \geq, \nu) \in \mathcal{ALC}^{Tree}$. An element $v \in T$ respectively the corresponding subtree $\mathbf{T}_v$ is said to be* EVENLY ENCLOSED*, iff $|\{w \in T \mid w > v \text{ and } \nu(w) = \neg\}|$ is even. The notation of* ODDLY ENCLOSED *is defined accordingly.*

Of course, $\mathcal{ALC}$-trees correspond to the formulas of $\mathcal{ALC}$, as they are defined in the usual linear fashion. For this reason, we will sometimes mix the notation of $\mathcal{ALC}$-formulas and $\mathcal{ALC}$-trees. Particularly, we sometimes write $\mathbf{T}_1 \sqcap \mathbf{T}_2$ instead of $\sqcap(\mathbf{T}_1, \mathbf{T}_2)$. Moreover, the conjunction of trees can be extended to an arbitrary number of conjuncts, i.e.: If $\mathbf{T}_1, \ldots, \mathbf{T}_n$ are $\mathcal{ALC}$-trees, we are free to write $\mathbf{T}_1 \sqcap \ldots \sqcap \mathbf{T}_n$. We agree that for $n = 0$, we set $\mathbf{T}_1 \sqcap \ldots \sqcap \mathbf{T}_n := \top$.

Next a diagrammatic representation of $\mathcal{ALC}$-trees in the style of Peirce's EGs is provided. EGs as such correspond to *closed* FOL-formulas: They are evaluated to true or false. Nonetheless, they can be easily extended to RELATION GRAPHS [14, 6] which are evaluated to relations instead. This is done by adding a syntactical device to EGs which corresponds to free variables. The diagrammatic rendering of free variables can be done via numbered question markers, which are attached to the lines of identity of EGs. As $\mathcal{ALC}$-concepts correspond to FOL-formulas with exactly one free variable, we we assign to each $\mathcal{ALC}$-tree $\mathbf{T}$ a corresponding relation graph $\Psi(\mathbf{T})$ with exactly one (now unnumbered) query marker. Let A be an atomic concept, R be a role name, let $\mathbf{T}$, $\mathbf{T}_1$, $\mathbf{T}_2$ be $\mathcal{ALC}$-trees where we already have defined $\Psi(\mathbf{T}) = ?\text{—}G$, $\Psi(\mathbf{T}_1) = ?\text{—}G_1$, and $\Psi(\mathbf{T}_2) = ?\text{—}G_2$, respectively. Now Ψ is defined inductively as follows:



Considering our HAPPYMAN-example given in the introduction, the corresponding $\mathcal{ALC}$-tree, and a corresponding Peirce graph, is provided in Fig. 3. Due to our choice of constructors, we replaced $\forall f$ by $\neg\exists\neg f$ and $f \vee g$ by $\neg(\neg f \wedge \neg g)$ (for DL-concepts f, g). The rules of the forthcoming calculus can be best understood to be carried out on the Peirce graphs. The ongoing formal proofs with $\mathcal{ALC}$-trees will be depicted this way.

Finally we define semantic entailment between $\mathcal{ALC}$-trees.

Definition 2.2 *Let $\{\mathbf{T}_i \mid i \in I\}$ be a set of $\mathcal{ALC}$-Trees and let $\mathbf{T}$ be an $\mathcal{ALC}$-Tree. We set*

$$\{\mathbf{T}_i \mid i \in I\} \models \mathbf{T} \ :\Longleftrightarrow \bigcap_{i \in I} \mathcal{I}(\mathbf{T}_i) \subseteq \mathcal{I}(\mathbf{T}) \text{ for each interpretation } (\Delta^{\mathcal{I}}, \mathcal{I})$$

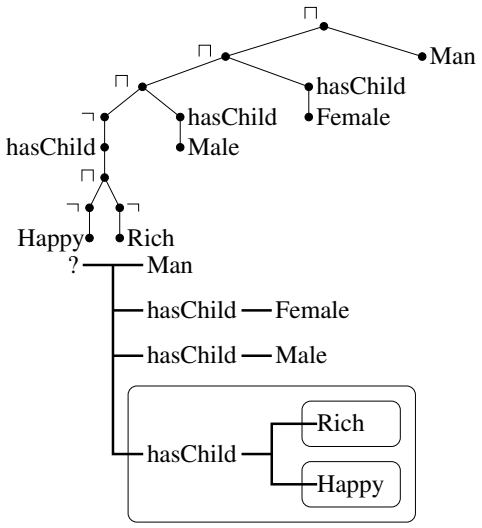


Figure 3: The $\mathcal{ALC}$-tree, and a corresponding Peirce graph for HAPPYMAN

For $I = \emptyset$, we set $\bigcap_{i \in I} \mathcal{I}(\mathbf{T}_i) := \Delta^{\mathcal{I}}$ for the respective model, and write $\models \mathbf{T}$. For $|I| = 1$, we write $\mathbf{T}' \models \mathbf{T}$.

3 The Calculus for $\mathcal{ALC}^{Tree}$

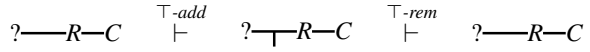
Peirce provided a set of five rules for the system of EGs, termed *erasure*, *insertion*, *iteration*, *deiteration*, *double cut*. These rules are formally elaborated and their soundness and completeness is proven in [6]. Moreover, it is discussed in [6] how these rules have to be extended for the system of relation graphs.

The class of relation graphs corresponding to $\mathcal{ALC}$ is a fragment of the full system of relation graphs. Of course, the rules for relation graphs are still sound rules for the $\mathcal{ALC}$-fragment. But it is not clear whether these rules are still complete. For two graphs G_1, G_2 of the $\mathcal{ALC}$-fragment with $G_1 \models G_2$, we have a proof for $G_1 \vdash G_2$ within the full system of relation graphs, i.e., a sequence of graphs starting with G_1, ending with G_2, where each graph in the sequence is derived from its predecessor by one of Peirce's five rules. However it may happen that we do not have a proof which consists only of graphs of the $\mathcal{ALC}$-fragment. In fact, in the calculus we provide, we need more than Peirce's five rules. Besides some trivial rules, like rules which capture the associativity of conjunction, we need special rules for handling of roles. The rules *iteration of roles into even* and *deiteration of roles from odd* (see below) are the most important example for this.

Next, the Peirce style rules for $\mathcal{ALC}^{Tree}$ are provided. These rules transform a given $\mathcal{ALC}$-tree into a new $\mathcal{ALC}$-tree. In order to make the calculus more understandable, we provide within the rule definitions some examples and diagrams that illustrate them. For each rule name, we provide an abbreviation which will be used in the proofs.

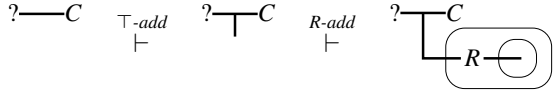
Definition 3.1 *The calculus for $\mathcal{ALC}$-Trees over a given vocabulary $(\mathcal{A}, \mathcal{R})$ consists of the following rules:*

Addition and Removal of $\top$ ***(*** $\top$***-add. and*** $\top$***-rem.):*** *Let* $\mathbf{T} := (T, \geq, \nu)$ *be an* $\mathcal{ALC}$*-tree, let* $\mathbf{S} \subseteq \mathbf{T}$ *be a subtree. For* $\mathbf{T}' := \mathbf{T}[\sqcap(\mathbf{S}, \top)) \,/\, \mathbf{S}\,]$ *we set* $\mathbf{T} \dashv\vdash \mathbf{T}'$ *(*$\mathbf{T} \dashv\vdash \mathbf{T}'$ *abbreviates* $\mathbf{T} \vdash \mathbf{T}'$ *and* $\mathbf{T}' \vdash \mathbf{T}$*). We say that* $\mathbf{T}'$ *is derived from* $\mathbf{T}$ *by* ADDING A $\top$-NODE*, and* $\mathbf{T}$ *is derived from* $\mathbf{T}'$ *by* REMOVING A $\top$-NODE*. For the Peirce graphs, this rule corresponds to adding a branch to a heavily drawn line, or removing it. A simple example is given below.*



These rules are "technical 'helper' rules that will be often combine with other rules that add or remove subtrees. Examples will be given below.

Addition and Removal of Roles (R-add. and R-rem.): *Let* $\mathbf{T}$ *be an* $\mathcal{ALC}$*-tree having* $\top$ *as a subtree. Let* R *be a role name. Then for* $\mathbf{T}[\,\neg R\neg\top \,/\, \top\,]$ *we set* $\mathbf{T} \dashv\vdash \mathbf{T}'$*. We say that* $\mathbf{T}'$ *is derived from* $\mathbf{T}$ *by* ADDING THE ROLE R*, and* $\mathbf{T}$ *is derived from* $\mathbf{T}'$ *by* REMOVING THE ROLE R*. A simple example for this rule with Peirce graphs is given below. Due to the symmetry of the rules, the inverse direction of this proof is a proof as well.*



Associativity of Conjunction (conj.): *Let* $\mathbf{T}$ *be an* $\mathcal{ALC}$*-tree with a subtree* $\mathbf{S}_1 \sqcap (\mathbf{S}_2 \sqcap \mathbf{S}_3)$*. For* $\mathbf{T}' := \mathbf{T}[\,(\mathbf{S}_1 \sqcap \mathbf{S}_2) \sqcap \mathbf{S}_3 \,/\, \mathbf{S}_1 \sqcap (\mathbf{S}_2 \sqcap \mathbf{S}_3)\,]$ *we set* $\mathbf{T} \dashv\vdash \mathbf{T}'$*. We say that* $\mathbf{T}'$ *is derived from* $\mathbf{T}$ *resp.* $\mathbf{T}$ *is derived from* $\mathbf{T}'$ *by* USING THE ASSOCIATIVITY OF CONJUNCTION.

Addition and Removal of a Double Negation (dn): *Let* $\mathbf{T} := (T, \geq, \nu)$ *be an* $\mathcal{ALC}$*-tree, let* $\mathbf{S} \subseteq \mathbf{T}$ *be a subtree. Then for* $\mathbf{T}' := \mathbf{T}[\,\neg\neg\mathbf{S} \,/\, \mathbf{S}\,]$ *we set* $\mathbf{T} \dashv\vdash \mathbf{T}'$*. We say that* $\mathbf{T}'$ *is derived from* $\mathbf{T}$ *by* ADDING A DOUBLE NEGATION *and* $\mathbf{T}$ *is derived from* $\mathbf{T}'$ *by* REMOVING A DOUBLE NEGATION.

Erasure from even, Insertion into odd (era. and ins.): *Let* $\mathbf{T} :=$ *be an* $\mathcal{ALC}$*-tree with a positively enclosed subtree* $\mathbf{S} \subseteq \mathbf{T}$*. Then for* $\mathbf{T}' := \mathbf{T}[\,\top \,/\, \mathbf{S}\,]$ *we set* $\mathbf{T} \vdash \mathbf{T}'$*. We say that* $\mathbf{T}'$ *is derived from* $\mathbf{T}$ *by* ERASING $\mathbf{S}$ FROM EVEN*. Vice versa, let* $\mathbf{T} =$ *be an* $\mathcal{ALC}$*-tree with an negatively enclosed subtree* $\top \subseteq \mathbf{T}$*. Let* $\mathbf{S} \in \mathcal{ALC}^{Tree}$*. Then for* $\mathbf{T}' := \mathbf{T}[\,\mathbf{S} \,/\, \top\,]$ *we set* $\mathbf{T} \vdash \mathbf{T}'$*. We say that* $\mathbf{T}'$ *is derived from* $\mathbf{T}$ *by* INSERTING $\mathbf{S}$ INTO ODD.

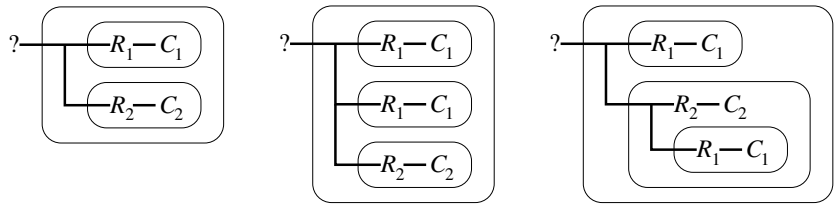
This is another set of rules which often go together with the addition and removal of $\top$*. Examples will be given later.*

Iteration and Deiteration (it. and deit.): *Let* $\mathbf{T} := (T, \geq, \nu)$ *be an* $\mathcal{ALC}$*-tree with with a subtree* $\mathbf{S} := (S, \geq_S, \nu_S) \subseteq \mathbf{T}$*. Let* s *be the greatest element of* $\mathbf{S}$*, let* t *be the parent node of* s *in* $\mathbf{T}$*. Let* $\nu(t) = \sqcap$ *(i.e.* t *is labelled with '*$\sqcap$*'), let* $v \in T$ *be a node with* $v < t$*,* $v \notin S$*,* $\nu(v) = \top$*, such that for each node* w *with* $t > w > v$ *we have* $\nu(w) = \neg$ *or* $\nu(w) = \sqcap$*. Then for* $\mathbf{T}' := \mathbf{T}[\,\mathbf{S} \,/\, \top\,]$ *we set* $\mathbf{T} \dashv\vdash \mathbf{T}'$*.*[1] *We say that* $\mathbf{T}'$ *is derived from* $\mathbf{T}$ *by* ITERATING $\mathbf{S}$ *and* $\mathbf{T}$ *is derived from* $\mathbf{T}'$ *by* DEITERATING $\mathbf{S}$*.*

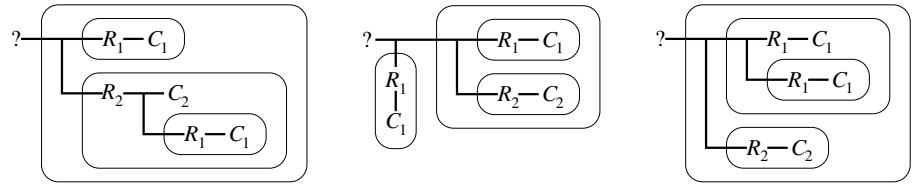
Iteration and Deiteration often combine with the addition and removal of $\top$*, and they are probably the most complex rules. To exemplify them, we consider the following six Peirce graphs. The second and the third graph can be derived from the first graph*

[1] More precisely and according to our convention, we set $\mathbf{T}' := \mathbf{T}[\,\mathbf{S}' \,/\, \top\,]$, where $\mathbf{S}'$ is an isomorphic copy of $\mathbf{S}$, having only fresh nodes.

by iterating the subgraph ?—(—R_1—C_1—) *(preceeded by the* $\top$*-addition rule).*

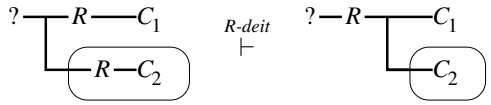


The next three graphs are not *results from the iteration rule. In the fourth graph, the condition that* $\nu(w) = \neg$ *or* $\nu(w) = \sqcap$ *holds for each node* w *with* $t > w > v$ *is violated. The fifth graph violates the condition* $v < t$*, and the sixth graph violates the condition* $v \notin S$*.*



Iteration of Roles into even, Deiteration of Roles from odd (R-it. and R-deit.): *Let* $\mathbf{T}$ *be an* $\mathcal{ALC}$*-tree. Let* $\mathbf{S}_a, \mathbf{S}_b, \mathbf{S}_1, \mathbf{S}_2$ *be* $\mathcal{ALC}$*-trees with* $\mathbf{S}_a := \sqcap(R\mathbf{S}_1, \neg R\mathbf{S}_2)$ *and* $\mathbf{S}_b := R \sqcap (\mathbf{S}_1, \neg\mathbf{S}_2)$*. Then, if* $\mathbf{S}_a \subseteq \mathbf{T}$ *is a positively enclosed subtree, for* $\mathbf{T}' := \mathbf{T}[\,\mathbf{S}_b \,/\, \mathbf{S}_a\,]$ *we set* $\mathbf{T} \vdash \mathbf{T}'$*, and we say that* $\mathbf{T}'$ *is derived from* $\mathbf{T}$ *by* DEITERATING THE ROLE R FROM ODD*. Vice versa, if* $\mathbf{S}_b \subseteq \mathbf{T}$ *is a negatively enclosed subtree, for* $\mathbf{T}' := \mathbf{T}[\,\mathbf{S}_a \,/\, \mathbf{S}_b\,]$ *we set* $\mathbf{T} \vdash \mathbf{T}'$*, and we say that* $\mathbf{T}'$ *is derived from* $\mathbf{T}$ *by* ITERATING THE ROLE R INTO EVEN*.*

Below, a simple example for the rule with Peirce's graphs is provided.



Based on these rules, we can now define formal proofs.

Definition 3.2 *Let* $\mathbf{T}_a, \mathbf{T}_b$ *be two* $\mathcal{ALC}$*-Trees. A* PROOF FOR $\mathbf{T}_a \vdash \mathbf{T}_b$ *is a finite sequence* $(\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_n)$ *with* $\mathbf{T}_a = \mathbf{T}_1$*,* $\mathbf{T}_b = \mathbf{T}_n$*, where each* $\mathbf{T}_{i+1}$ *is obtained from* $\mathbf{T}_i$ *by applying one of the rules of the calculus.*

Now let $\{\mathbf{T}_i \mid i \in I\}$ *be a set of* $\mathcal{ALC}$*-Trees and let* $\mathbf{T}$ *be an* $\mathcal{ALC}$*-Tree. We set*

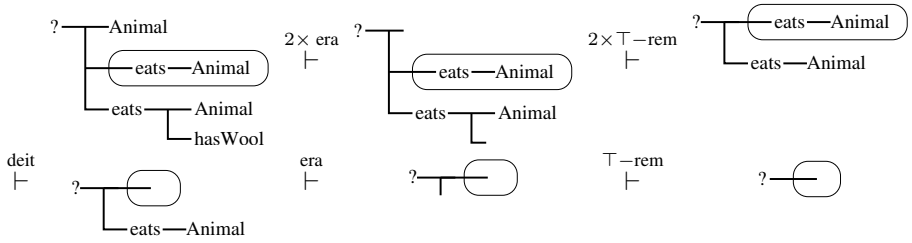
$$\{\mathbf{T}_i \mid i \in I\} \vdash \mathbf{T} \quad :\Longleftrightarrow \quad \text{there are } \mathcal{ALC}\text{-Trees } \mathbf{T}_1, \ldots, \mathbf{T}_n \in \{\mathbf{T}_i \mid i \in I\} \text{ with } \mathbf{T}_1 \sqcap \ldots \sqcap \mathbf{T}_n \vdash \mathbf{T}$$

Before the soundness and completeness of the calculus is proven, we first present an example of a proof, using the Peirce-style diagrams, and then derive some useful metarules. The example and the metarules will give some insights in how the calculus works.

A popular toy example for $\mathcal{ALC}$.reasoning is the mad cow ontology. Consider the following $\mathcal{ALC}$-definitions:

$$\begin{array}{rcl@{\qquad}rcl} Cow & \equiv & Animal \sqcap Vegetarian & Sheep & \equiv & Animal \sqcap hasWool \\ Vegetarian & \equiv & \forall eats.\neg Animal & MadCow & \equiv & Cow \sqcap \exists eats.Sheep \end{array}$$

The question to answer is whether this ontology is consistent. This question can be reduced to rewriting the ontology to a single concept $MadCow \equiv Animal \sqcap \forall eats.\neg Animal \sqcap \exists eats.(Animal \sqcap hasWool)$ and to investigate whether this concept is satisfiable, i.e., whether there exists as least one interpretation where this concept is interpreted by a non-empty set. We will show that this is not the case by proving with our calculus that the concept entails the absurd concept. The proof is given below.



We started with the Peirce graph for the given concept and derived the absurd concept, thus the ontology is not satisfiable.

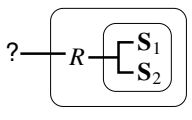
Next, we provide the above mentioned metarules, some of them will be used in the completeness proof.

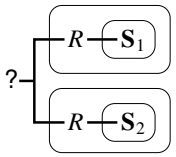
Each rule of the calculus is basically the substitution of a subtree of a given $\mathcal{ALC}$-tree by another subtree. Each rule can be applied to arbitrarily deeply nested subtrees. Moreover, if we have a rule which can be applied to positively enclosed subtrees, then we always have a rule in the converse direction which can be applied to negatively enclosed subtrees (and visa versa). Due to these structural properties of rules, we immediately obtain the following helpful lemma (it is adopted from [17]).

Lemma 3.1 *Let $\mathbf{S}_1, \mathbf{S}_2$ be two $\mathcal{ALC}$-trees with $\mathbf{S}_1 \vdash \mathbf{S}_2$. Let $\mathbf{T}$ be an $\mathcal{ALC}$-tree. Then if $\mathbf{S}_1 \subseteq \mathbf{T}$ is a positively enclosed subtree of $\mathbf{T}$, we have $\mathbf{T} \vdash \mathbf{T}[\,\mathbf{S}_2 \,/\, \mathbf{S}_1\,]$. Visa versa, if $\mathbf{S}_2 \subseteq \mathbf{T}$ is a negatively enclosed subtree of $\mathbf{T}$, we have $\mathbf{T} \vdash \mathbf{T}[\,\mathbf{S}_1 \,/\, \mathbf{S}_2\,]$.*

The next lemma corresponds to the equivalence of the $\mathcal{ALC}$-concepts $\forall R.(C \sqcap D)$ and $\forall R.C \sqcap \forall R.D$.

Lemma 3.2 (Splitting Roles) *Let $\mathbf{S}_1, \mathbf{S}_2$ be $\mathcal{ALC}$-trees, let $R \in \mathcal{R}$ be a role name. Then:*

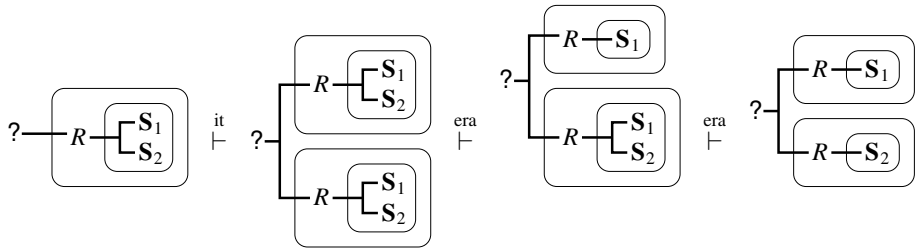
The trees 

and 

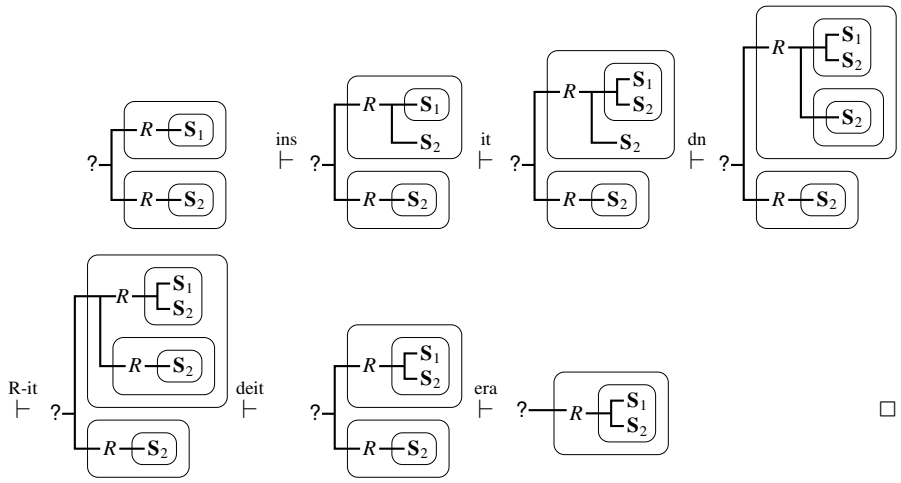
are equivalent.

Proof: We show the directions '$\Rightarrow$' and '$\Leftarrow$' separately. As we have already seen, the deiteration-rule and the erasure rule are usually followed by the $\top$-removal rule, and visa versa, the iteration rule and the insertion rule are usually preceeded by the $\top$-addition rule. In the proof, these two steps are combined without explicitely mentioning

the $\top$-removal/addition rule. Now '$\Rightarrow$' is proven as follows:



The other direction is as follows:

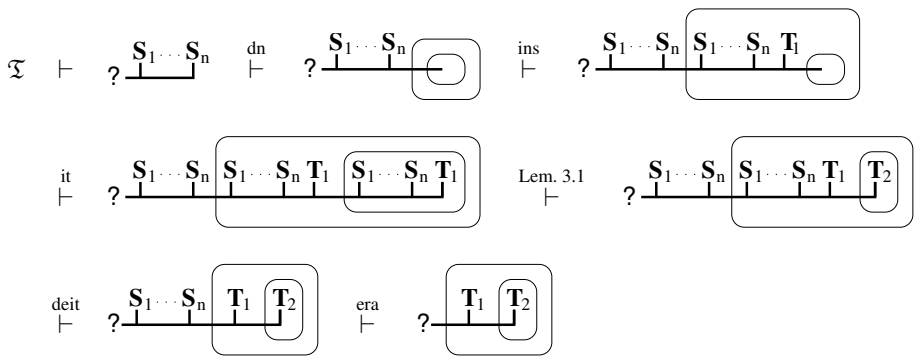


For $\mathcal{ALC}$, the full deduction theorem holds.

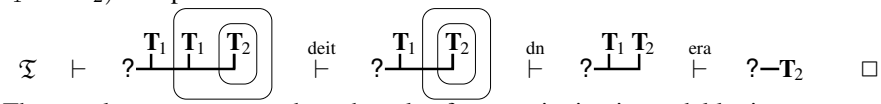
Theorem 3.1 (Deduction Theorem) *Let $\mathfrak{T}$ be a set of $\mathcal{ALC}$-trees, let $\mathbf{T}_1, \mathbf{T}_2$ be two $\mathcal{ALC}$-trees. Then we have:* $\mathfrak{T} \cup \{\mathbf{T}_1\} \vdash \mathbf{T}_2 \iff \mathfrak{T} \vdash \neg(\mathbf{T}_1 \sqcap \neg\mathbf{T}_2)$

Proof: Again applications of the $\top$-removal/addition rule are not explicitly mentioned.

'$\Rightarrow$': Let $\mathbf{S}_1, \ldots, \mathbf{S}_n \in \mathfrak{T}$ with $\mathbf{S}_1 \sqcap \ldots \sqcap \mathbf{S}_n \sqcap \mathbf{T}_1 \vdash \mathbf{T}_2$. We have:



'$\Leftarrow$': From $\mathfrak{T} \vdash \neg(\mathbf{T}_1 \sqcap \neg\mathbf{T}_2)$ and $\mathfrak{T} \cup \{\mathbf{T}_1\} \vdash \mathbf{T}_1$ we get $\mathfrak{T} \cup \{\mathbf{T}_1\} \vdash \mathbf{T}_1 \sqcap \neg(\mathbf{T}_1 \sqcap \neg\mathbf{T}_2)$. We proceed as follows:



The next lemma corresponds to the rule of neccessitation in modal logics.

Lemma 3.3 *If* $\mathbf{T}$ *is an* $\mathcal{ALC}$*-tree with* $\vdash \mathbf{T}$*, then we have* $\vdash \neg R \neg \mathbf{T}$ *as well.*

Proof: All rules of the calculus modify subtrees $\mathbf{S}$ of a given $\mathcal{ALC}$-tree $\mathbf{T}$, and their application depends only on whether $\mathbf{S}$ is positively or negatively enclosed. So if $(\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_n)$ with $\mathbf{T}_1 = \top$ and $\mathbf{T}_n = \mathbf{T}$ is a proof for $\mathbf{T}$, then $(\top, \neg R \neg \mathbf{T}_1, \neg R \neg \mathbf{T}_2, \ldots, \neg R \neg \mathbf{T}_n)$ is a proof for $\vdash \neg R \neg \mathbf{T}$. The additional first step is an application of the rule 'addition of roles'. □

Please note that for this lemma, it is vital that $\mathbf{T}$ is derived from the empty set (the empty sheet of assertion in Peirce's terminology). The proof of the lemma does not work if $\mathbf{T}$ is derived from some set $\mathfrak{T}$, and it can easily be seen that we generally cannot conclude $\mathfrak{T} \vdash \neg R \neg \mathbf{T}$ from $\mathfrak{T} \vdash \mathbf{T}$.

In the following, the soundness and completeness of the calculus is proven. In contrast to rules of most common calculi, the rules presented here are 'deep' rules, as they modify deeply nested subtrees. For this reason, the following lemma is helpful for proving the soundness of the rules.

Lemma 3.4 *Let* $\mathbf{T}_1$*,* $\mathbf{T}_2$*,* $\mathbf{S}_1$*,* $\mathbf{S}_2$ *be* $\mathcal{ALC}$*-trees with* $\mathbf{T}_2 = \mathbf{T}_1[\,\mathbf{S}_2 \,/\, \mathbf{S}_1\,]$.

1. *If* $\mathbf{S}_1 \models \mathbf{S}_2$ *and the substitution takes place in an even, then* $\mathbf{T}_1 \models \mathbf{T}_2$.
2. *If* $\mathbf{S}_2 \models \mathbf{S}_1$ *and the substitution takes place in an odd, then* $\mathbf{T}_1 \models \mathbf{T}_2$.
3. *If* $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *are semantically equivalent, then so are* $\mathbf{T}_1$ *and* $\mathbf{T}_2$.

Proof: The proof of this lemma is a straight-forward induction on $\mathcal{ALC}$-trees.

We are now prepared to prove the soundness of the calculus.

Theorem 3.2 *If* $(\Delta^{\mathcal{I}}, \mathcal{I})$ *is a model and if* $\mathbf{T}'$ *is obtained from* $\mathbf{T}$ *by one of the rules, we have* $\mathcal{I}(\mathbf{T}) \subseteq \mathcal{I}(\mathbf{T}')$.

Proof: The $\mathcal{ALC}$-trees $\mathbf{S}$ and $\sqcap(\mathbf{S}, \top)$ are obviously equivalent. So the soundness of the rule 'Addition or Removal of $\top$' follows immediately from Lemma 3.4(iii). The rules 'Addition or Removal of Roles', 'Associativity of Conjunction' and 'Addition or Removal of a Double Negation' are handled similarly.

Next, as we have $\mathbf{T} \models \top$, Lemma 3.4(i) yields the soundness of the erasure of a a positively enclosed subtree, and Lemma 3.4(ii) yields the soundness of the insertion negatively enclosed subtree.

Next we consider the iteration and deiteration of roles. Let $\mathbf{S}_a, \mathbf{S}_b$ be defined as in the rule. We will show $\mathbf{S}_a \models \mathbf{S}_b$. Let $a \in \mathcal{I}(\mathbf{S}_a)$. Then it follows $a \in \mathcal{I}(R\mathbf{S}_1)$ and $a \in \mathcal{I}(\neg R\mathbf{S}_2)$. Therefore there exists $b \in \Delta^{\mathcal{I}}$ with aRb and $b \in \mathcal{I}(\mathbf{S}_1)$, but there exists no $c \in \Delta^{\mathcal{I}}$ with aRc and $c \in \mathcal{I}(\mathbf{S}_2)$. Particularly we have $b \notin \mathcal{I}(\mathbf{S}_2)$. We conclude $b \in \mathcal{I}(\neg \mathbf{S}_2)$, so we have $b \in \mathcal{I}(\sqcap(\mathbf{S}_1, \neg \mathbf{S}_2))$ as well. Due to aRb, we finally obtain $a \in \mathcal{I}(\mathbf{S}_b)$. As we now have $\mathbf{S}_a \models \mathbf{S}_b$, Lemma 3.4(i) yields the soundness of deiterating a role R from an odd, and Lemma 3.4(ii) yields the soundness of iterating a role into an even.

Finally, we have to prove the soundness of the iteration and deiteration rule. First note the iteration rule removes v from T and adds the fresh nodes of S' to T, i.e., we have $T - \{v\} \subseteq T'$. To ease the technical presentation, let us assume that the greatest element of $\mathbf{S}'$ is v (instead of a fresh node), so that we have $T \subseteq T'$.

For a node $w \in T$, let $\mathbf{T}_w$ be the corresponding subtree of $\mathbf{T}$, and let $\mathbf{T}'_w$ be the corresponding subtree of $\mathbf{T}'$ (particularly, due to our assumption, we have $\mathbf{T}_v = \top$ and

$\mathbf{T}'_v = \mathbf{S}'$). We will prove that $\mathbf{T}_t$ and $\mathbf{T}'_t$ are semantically equivalent. So let $a \in \Delta^{\mathcal{I}}$. We have to show that,

$$a \in \mathcal{I}(\mathbf{T}_w) \quad \Longleftrightarrow \quad a \in \mathcal{I}(\mathbf{T}'_w) \tag{1}$$

holds for $w = t$. We have $\mathbf{T}' = \mathbf{T}[\,\mathbf{T}'_t \,/\, \mathbf{T}_t\,]$, so once Eqn. (1) is proven for $w = t$, we can now apply Lemma 3.4(iii) and obtain that $\mathbf{T}$ and $\mathbf{T}'$ are semantically equivalent, which yields the soundness of the iteration and deiteration rule. So it remains to show Eqn. (1).

In either $\mathbf{T}$ and $\mathbf{T}'$, the node t has two branches, one of which is $\mathbf{S}$. If we have $a \notin \mathcal{I}(\mathbf{S})$, we have $a \notin \mathcal{I}(\mathbf{T}_t)$ and $a \notin \mathcal{I}(\mathbf{T}'_t)$ as well, so Eqn. (1) holds. Now let us assume the alternate case, that we have $a \in \mathcal{I}(\mathbf{S})$. We will prove that Eqn. (1) holds for each w with $t \geq w \geq v$ by induction.

We have $\mathbf{T}_v = \top$, $\mathbf{T}'_v = \mathbf{S}$, $a \in \mathcal{I}(\top)$ and $a \in \mathcal{I}(\mathbf{S}')$, so Eqn. (1) holds for $w = v$. This proves the induction start.

For the induction step, let w be such that the induction hypothesis is proven for the child u of w with $u \geq v$. There are two cases to consider: $\nu(w) = \neg$ or $\nu(w) = \sqcap$.

For $\nu(w) = \neg$, we have $\mathbf{T}_w = \neg\mathbf{T}_u$ and $\mathbf{T}'_w = \neg\mathbf{T}'_u$. As we have $a \in \mathcal{I}(\mathbf{T}_u) \Leftrightarrow a \in \mathcal{I}(\mathbf{T}'_u)$ due to the induction hypothesis, we obtain that Eqn. (1) holds for w.

For $\nu(w) = \sqcap$, the node w has two children, one of them being u. Let u' be the child of w which is different from u. Then we have $\mathbf{T}_w = \sqcap(\mathbf{T}_u, \mathbf{T}_{u'})$ and $\mathbf{T}'_w = \sqcap(\mathbf{T}'_u, \mathbf{T}'_{u'})$. Moreover, we have $\mathbf{T}_{u'} = \mathbf{T}'_{u'}$, and $a \in \mathcal{I}(\mathbf{T}_u) \Leftrightarrow a \in \mathcal{I}(\mathbf{T}'_u)$ holds due to the induction hypothesis. From this we conclude that Eqn. (1) holds.

This finishes the induction, so we conclude Eqn. (1) for $w = t$, which in turn finishes the proof for the soundness of the iteration and deiteration rule. □

We are now prepared to prove the completeness of the calculus.

Theorem 3.3 *Let $\mathfrak{T} := \{\mathbf{T}_i \mid i \in I\}$ be a set of $\mathcal{ALC}$-Trees and let $\mathbf{T}$ be an $\mathcal{ALC}$-Tree. Then we have:*

$$\{\mathbf{T}_i \mid i \in I\} \models \mathbf{T} \quad \Longrightarrow \quad \{\mathbf{T}_i \mid i \in I\} \vdash \mathbf{T}$$

Proof: We assume that there is no derivation of $\mathbf{T}$ from $\mathfrak{T}$, and we show that $\mathfrak{T} \not\models \mathbf{T}$.

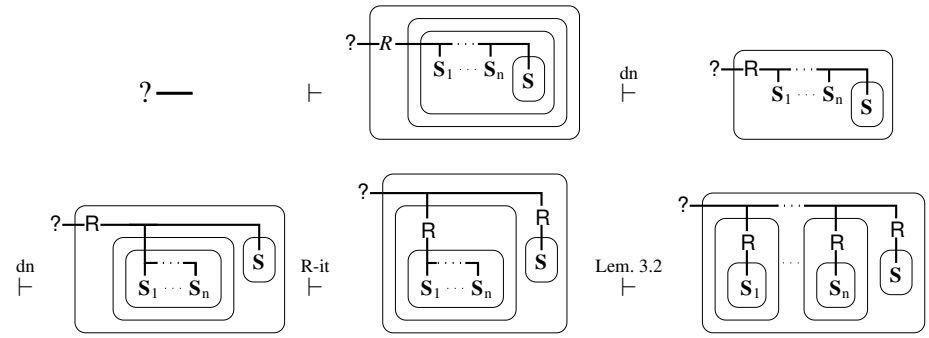
We call a set $\mathfrak{S}$ of $\mathcal{ALC}$-Trees INCONSISTENT, if we have $\mathfrak{S} \vdash \neg\top$. Due to the deduction theorem, $\mathfrak{S}$ is inconsistent if and only if there are $\mathbf{S}_1, \ldots, \mathbf{S}_n \in \mathfrak{S}$ with $\vdash \neg(\mathbf{S}_1 \sqcap \ldots \sqcap \mathbf{S}_n)$. We assume that there is no proof of $\mathbf{T}$ from $\mathfrak{T}$. Then $\mathfrak{T} \cup \{\neg\mathbf{T}\}$ is consistent.

For a set $\mathfrak{S}$ of $\mathcal{ALC}$-Trees and a role name R, let $\mathfrak{S}^R := \{\mathbf{S} \mid \neg R\neg\mathbf{S} \in \mathfrak{S}\}$. We first prove the following property:

$$\text{If } \mathfrak{S} \text{ is consistent, where } R\neg\mathbf{S} \in \mathfrak{S}, \text{ then } \mathfrak{S}^R \cup \{\neg\mathbf{S}\} \text{ is also consistent.} \tag{2}$$

It is easier to show the contraposition of Eqn. (2), so we assume that $\mathfrak{S}^R \cup \{\neg\mathbf{S}\}$ is not consistent. Then there exists finitely many elements $\mathbf{S}_1, \ldots, \mathbf{S}_n$ of $\mathfrak{S}^R$ such that there is a proof of $\neg(\mathbf{S}_1 \sqcap \ldots \mathbf{S}_n \sqcap \neg\mathbf{S})$ (from the empty set). Now Lemma 3.3 yields that

we have a proof of $\neg R\neg\neg(\mathbf{S}_1 \sqcap \ldots \mathbf{S}_n \sqcap \neg\mathbf{S})$ as well. We proceed as follows:



So $\mathfrak{S}$ is also $\mathfrak{T}$-inconsistent, thus Eqn. (2) holds.

Now, using a standard argument based on the axiom of choice, every consistent set can be extended to a maximal consistent set, i.e., a consistent set which cannot be properly extended to another consistent set.

Next, for a maximal consistent set $\mathfrak{S}_\mathrm{m}$, we have

$$\mathbf{S} \in \mathfrak{S}_\mathrm{m} \iff \neg\mathbf{S} \notin \mathfrak{S}_\mathrm{m} \tag{3}$$

$$\mathbf{S} \sqcap \mathbf{S}' \in \mathfrak{S}_\mathrm{m} \iff \mathbf{S} \in \mathfrak{S}_\mathrm{m} \text{ and } \mathbf{S}' \in \mathfrak{S}_\mathrm{m} \tag{4}$$

for arbitrary $\mathcal{ALC}$-Trees $\mathbf{S}, \mathbf{S}'$. We only prove Eqn. (3), the proof of Eqn. (4) is done similarly.

Due to $\mathbf{S} \sqcap \neg\mathbf{S} \overset{\text{deit}}{\vdash} \mathbf{S} \sqcap \neg\top \overset{\text{era}}{\vdash} \top \sqcap \neg\top \overset{\top\text{-rem}}{\vdash} \neg\top$, and as we can infer any tree from $\neg\top$, we cannot have both $\mathbf{S} \in \mathfrak{S}_\mathrm{m}$ and $\neg\mathbf{S} \in \mathfrak{S}_\mathrm{m}$. On the other hand, let us suppose we have $\neg\mathbf{S} \notin \mathfrak{S}_\mathrm{m}$. Then we have $\mathfrak{S}_\mathrm{m} \nvdash \neg\mathbf{S}$. Assume $\mathfrak{S}_\mathrm{m} \cup \{\mathbf{S}\}$ is inconsistent. Then we have $\mathbf{S}_1, \ldots, \mathbf{S}_n \in \mathfrak{S}_\mathrm{m}$ with $\vdash \neg(\mathbf{S}_1 \sqcap \ldots \sqcap \mathbf{S}_n \sqcap \mathbf{S})$. Thm. 3.1 yields $\mathbf{S}_1 \sqcap \ldots \sqcap \mathbf{S}_n \vdash \neg\mathbf{S}$, i.e., we have $\mathfrak{S}_\mathrm{m} \vdash \neg\mathbf{S}$. This contradicts $\mathfrak{S}_\mathrm{m} \nvdash \neg\mathbf{S}$. So if $\neg\mathbf{S} \notin \mathfrak{S}_\mathrm{m}$, then $\mathfrak{S}_\mathrm{m} \cup \{\mathbf{S}\}$ is consistent, thus $\mathbf{S} \in \mathfrak{S}_\mathrm{m}$ due to the maximality of $\mathfrak{S}_\mathrm{m}$. Hence Equation (3) is proven.

Now let $(\Delta^\mathcal{I}, \mathcal{I})$ be defined as $\Delta^\mathcal{I} := \{\mathfrak{S}_\mathrm{m} \mid \mathfrak{S}_\mathrm{m} \text{ is maximal consistent}\}$, $\mathcal{I}(A) := \{\mathfrak{S}_\mathrm{m} \mid \mathbf{T}_A \in \mathfrak{S}_\mathrm{m}\}$ and $\mathcal{I}(R) := \{(\mathfrak{S}_\mathrm{m}, \mathfrak{T}_\mathrm{m}) \mid \mathfrak{S}_\mathrm{m}^R \subseteq \mathfrak{T}_\mathrm{m}\}$. We prove by induction over the construction of $\mathcal{ALC}$-trees that for $\mathbf{S} \in \mathcal{ALC}^{Tree}$ and $\mathfrak{S}_\mathrm{m} \in \Delta^\mathcal{I}$ we have

$$\mathfrak{S}_\mathrm{m} \in \mathcal{I}(\mathbf{S}) \iff \mathbf{S} \in \mathfrak{S}_\mathrm{m} \tag{5}$$

For a concept name A, Eqn. (5) holds by the definition of the model. For a tree $\neg\mathbf{S}$, we have $\mathfrak{S}_\mathrm{m} \in \mathcal{I}(\neg\mathbf{S}) \overset{\text{Def. } \mathcal{I}}{\iff} \mathfrak{S}_\mathrm{m} \notin \mathcal{I}(\mathbf{S}) \overset{\text{I.H.}}{\iff} \mathbf{S} \notin \mathfrak{S}_\mathrm{m} \overset{\text{Eqn. (3)}}{\iff} \neg\mathbf{S} \in \mathfrak{S}_\mathrm{m}$. For a tree $\mathbf{S} \sqcap \mathbf{S}'$, Eqn. (5) is similarly proven using Eqn. (4). It remains to consider role names.

Let $R \in \mathcal{R}$. We first prove Eqn. (5) for $\mathcal{ALC}$-trees $\mathbf{S}' := \neg R\neg\mathbf{S}$ (instead of $\mathbf{S}' := R\mathbf{S}$). Due to our induction, we can assume that Eqn. (5) is proven for all $\mathcal{ALC}$-trees which have less occurrences of R than $\mathbf{S}'$. Def. 2.1 yields

$$\mathfrak{S}_\mathrm{m} \in \mathcal{I}(\neg R\neg\mathbf{S}) \iff \text{for all } \mathfrak{T}_\mathrm{m} \text{ with } (\mathfrak{S}_\mathrm{m}, \mathfrak{T}_\mathrm{m}) \in \mathcal{I}(R) \text{ we have } \mathfrak{T}_\mathrm{m} \in \mathcal{I}(\mathbf{S}) \tag{6}$$

Suppose first we have $\mathbf{S}' \in \mathfrak{S}_\mathrm{m}$. If $\mathfrak{T}_\mathrm{m} \in \Delta^\mathcal{I}$ is arbitrary with $(\mathfrak{S}_\mathrm{m}, \mathfrak{T}_\mathrm{m}) \in \mathcal{I}(R)$, then $\mathbf{S} \in \mathfrak{T}_\mathrm{m}$ by the definition of the model. The induction hypothesis yields $\mathfrak{T}_\mathrm{m} \in \mathcal{I}(\mathbf{S})$, so Eqn. (6) yields $\mathfrak{S}_\mathrm{m} \in \mathcal{I}(\mathbf{S}')$. Next suppose $\mathbf{S}' \notin \mathfrak{S}_\mathrm{m}$. Eqn. (3) yields $R\neg\mathbf{S} \in \mathfrak{S}_\mathrm{m}$. Let $\mathfrak{T}' := \mathfrak{S}_\mathrm{m}^R \cup \{\neg\mathbf{S}\}$. Then $\mathfrak{T}'$ is consistent due to Eqn. (2). Let $\mathfrak{T}_\mathrm{m} \supseteq \mathfrak{T}'$ be a maximal consistent set. Then $\mathfrak{T}_\mathrm{m} \in \Delta^\mathcal{I}$, and $(\mathfrak{S}_\mathrm{m}, \mathfrak{T}_\mathrm{m}) \in \mathcal{I}(R)$. Since $\neg\mathbf{S} \in \mathfrak{T}' \subseteq \mathfrak{T}_\mathrm{m}$,

we have $\mathbf{S} \notin \mathfrak{T}_\mathrm{m}$. The induction hypothesis yields $\mathfrak{T}_\mathrm{m} \notin \mathcal{I}(\mathbf{S})$, thus $\mathfrak{S}_\mathrm{m} \notin \mathcal{I}(\mathbf{S}')$ due to Eqn. (6). Hence Eqn. (5) is proven for $\mathbf{S}' = \neg R \neg \mathbf{S}$.

To finish the proof of Eqn. (5), let us finally observe that for $\mathcal{ALC}$-trees of the form $R\mathbf{S}$, we have $\mathfrak{S}_\mathrm{m} \in \mathcal{I}(R\mathbf{S}) \iff \mathfrak{S}_\mathrm{m} \notin \mathcal{I}(\neg R \neg \neg \mathbf{S}) \overset{\text{s.a.}}{\iff} \neg R \neg \neg \mathbf{S} \notin \mathfrak{S}_\mathrm{m} \iff R\mathbf{S} \in \mathfrak{S}_\mathrm{m}$. So Eqn. (5) is proven.

Now let $\mathfrak{S}_\mathrm{m} \in \Delta^\mathcal{I}$. We have: $\mathfrak{S}_\mathrm{m} \in \bigcap_{\mathbf{S} \in \mathfrak{T}} \mathcal{I}(\mathbf{S}) \iff \mathfrak{S}_\mathrm{m} \in \mathcal{I}(\mathbf{S})$ for all $\mathbf{S} \in \mathfrak{T} \overset{\text{Eqn. (5)}}{\iff} \mathbf{S} \in \mathfrak{S}_\mathrm{m}$ for all $\mathbf{S} \in \mathfrak{T} \iff \mathfrak{T} \subseteq \mathfrak{S}_\mathrm{m}$. This yields $\bigcap_{\mathbf{S} \in \mathfrak{T}} \mathcal{I}(\mathbf{S}) = \{\mathfrak{S}_\mathrm{m} \in \Delta^\mathcal{I} \mid \mathfrak{S}_\mathrm{m} \supseteq \mathfrak{T}\}$. As $\mathfrak{T} \cup \{\neg \mathbf{T}\}$ is consistent, there exist a maximal consistent set $\mathfrak{T}^0_\mathrm{m} \supseteq \mathfrak{T} \cup \{\neg \mathbf{T}\}$. On the one hand, we now have $\mathfrak{T}^0_\mathrm{m} \in \bigcap_{\mathbf{S} \in \mathfrak{T}} \mathcal{I}(\mathbf{S})$. On the other hand, we have $\mathbf{T} \notin \mathfrak{T}^0_\mathrm{m}$, thus $\mathfrak{T}^0_\mathrm{m} \notin \mathcal{I}(\mathbf{T})$ by Eqn. (5). So we obtain $\bigcap_{\mathbf{S} \in \mathfrak{T}} \mathcal{I}(\mathbf{S}) \not\subseteq \mathcal{I}(\mathbf{T})$, which means $\mathfrak{T} \not\models \mathbf{T}$. □

4 Conclusion and Further Research

This paper provides the first steps toward a diagrammatic representation of DLs, including diagrammatic inference mechanisms. To the best of our knowledge, this is the first attempt to providing *diagrammatic* reasoning facilities for DLs. The results presented in this paper show promise in investigating relation graphs further as diagrammatic versions of corresponding DLs.

The approach can to be extended to other variants of DL as well. For instance, a major task is to incorporate nominals, or number restrictions (either unqualified or qualified). Similarly, constructors on roles, like inverse roles or role intersection, have also to be investigated.

In the long term, our research advocates developing a major subset of DL as a mathematically precise diagrammatic reasoning system. While the intention is to render DL more user-friendly through a diagrammatic correspondence, such systems will need to be evaluated against the traditional textual form of DL in order to measure any readability improvement. Cognition experiments with such a evaluation are planned as future work.

5 Acknowledgements

We like to thank the anonymous referees for their valuable suggestions, and particularly to the ones who pointed out a gap in the former proof of the completeness.

References

[1] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[2] S. Brockmans, R. Volz, A. Eberhart, and P. Löffler. Visual modeling of owl dl ontologies using uml. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *International Semantic Web Conference*, volume 3298 of *Lecture Notes in Computer Science*, pages 198–213. Springer, Berlin – Heidelberg – New York, 2004.

[3] P. Coupey and C. Faron. Towards correspondence between conceptual graphs and description logics. In M.-L. Mugnier and M. Chein, editors, *ICCS*, volume 1453 of *LNAI*, pages 165–178. Springer, Berlin – Heidelberg – New York, 1998.

[4] F. Dau. Types and tokens for logic with diagrams: A mathematical approach. In K. E. Wolff, H. D. Pfeiffer, and H. S. Delugach, editors, *Conceptual Structures at Work: 12th International Conference on Conceptual Structures*, volume 3127 of *Lecture Notes in Computer Science*, pages 62–93. Springer, Berlin – Heidelberg – New York, 2004.

[5] F. Dau. Fixing shin's reading algorithm for peirce's existential graphs. In D. Barker-Plummer, R. Cox, and N. Swoboda, editors, *Diagrams*, volume 4045 of *LNAI*, pages 88–92. Springer, Berlin – Heidelberg – New York, 2006.

[6] F. Dau. Mathematical logic with diagrams, based on the existential graphs of peirce. Habilitation thesis. To be published. Available at: `http://www.dr-dau.net`, 2006.

[7] F. Dau. Some notes on proofs with alpha graphs. In P. Øhrstrøm, H. Schärfe, and P. Hitzler, editors, *ICCS*, volume 4068 of *Lecture Notes in Computer Science*, pages 172–188. Springer, Berlin – Heidelberg – New York, 2006.

[8] F. Dau and P. Eklund. Towards a diagrammatic reasoning system for description logics. Submitted to the Journal of Visual Languages and Computing, Elsevier. Available at `www.kvocentral.org`., 2006.

[9] B. R. Gaines. An interactive visual language for term subsumption languages. In *IJCAI*, pages 817–823, 1991.

[10] W. Hartshorne and Burks, editors. *Collected Papers of Charles Sanders Peirce*, Cambridge, Massachusetts, 1931–1935. Harvard University Press.

[11] J. Howse, F. Molina, S.-J. Shin, and J. Taylor. On diagram tokens and types. In M. Hegarty, B. Meyer, and N. H. Narayanan, editors, *Diagrams*, volume 2317 of *Lecture Notes in Computer Science*, pages 146–160. Springer, Berlin – Heidelberg – New York, 2002.

[12] R. Kremer. Visual languages for konwledge representation. In *Proc. of 11th Workshop on Knowledge Acquisition, Modeling and Management (KAW'98) Banff, Alberta, Canada*. Morgan Kaufmann, 1998.

[13] J. T. Nosek and I. Roth. A comparison of formal knowledge representation schemes as communication tools: Predicate logic vs semantic network. *International Journal of Man-Machine Studies*, 33(2):227–239, 1990.

[14] S. Pollandt. Relation graphs: A structure for representing relations in contextual logic of relations. In U. Priss, D. Corbett, and G. Angelova, editors, *Conceptual Structures: Integration and Interfaces*, volume 2393 of *LNAI*, pages 24–48, Borovets, Bulgaria, July, 15–19, 2002. Springer, Berlin – Heidelberg – New York.

[15] D. D. Roberts. *The Existential Graphs of Charles S. Peirce*. Mouton, The Hague, Paris, 1973.

[16] S.-J. Shin. *The Iconic Logic of Peirce's Graphs*. Bradford Book, Massachusetts, 2002.

[17] J. F. Sowa. *Conceptual structures: information processing in mind and machine*. Addison-Wesley, Reading, Mass., 1984.

[18] J. J. Zeman. *The Graphical Logic of C. S. Peirce*. PhD thesis, University of Chicago, 1964. Available at: `http://www.clas.ufl.edu/users/jzeman/`.

A PROLOG-based Approach to Representing and Querying Software Engineering Models

Harald Störrle *
mgm technology partners GmbH
Munich, Germany

Abstract

Striving toward the vision of Model Driven development (MDD), we face many open questions connected to the elementary tasks involved in working with models. Probably the most basic task is querying models for properties, elements, and submodels. Current tools and interfaces for model querying are either restricted in their expressiveness or require a high level of expertise in the underlying metamodels and/or query languages. As the application of MDD is gaining more widespread acceptance and more and more developers are involved with MDD efforts, this state is becoming a bottleneck. In this paper, we propose a Prolog-based model representation and query interface for models to overcome this bottleneck.

Keywords: Model Driven Development (MDD), Query-View- Transformation (QVT), Knowledge Based Software Engineering (KBSE), Industrial Applications

1 Introduction

The MoMaT approach has been developed over the last years, partially in an academic setting, and partially in two very large scale industrial projects with German federal and state agencies. In these projects, very large models have been created and demand for advanced model operations soon became pressing. We have thus turned to MDD/MDA technology.

Model Driven Development/Architecture (MDD/MDA, [6, 17]) has been proposed as a measure to raise the level of abstraction in software development, and thus to increase developer productivity. In a MDA setting, models are programs, thus modeling languages are programming languages (cf. [20]). Today, there are many different practically relevant modeling languages, most of which are predominantly visual modeling languages. Examples are the Unified Modeling Language (UML, [16]), Entity-Relationship-Diagrams (ERD, [4]), Event Process Chains (EPCs, [5]), Integration Definition for Function Modeling (IDEF, [13]), or Use Case Maps (UCM, [3]). In principle, such models may be used for a multitude of purposes, such as reporting, model transformations, model consistency checking, formal analysis, code generation, pattern detection, versioning, size measurement and so on. See Figure 1 for a synopsis of model operations.

However, one of the basic tasks in an MDD setting is querying models for properties, elements, and submodels. This task is executed, on the one hand, by developers

*Harald.Stoerrle@mgm-tp.com

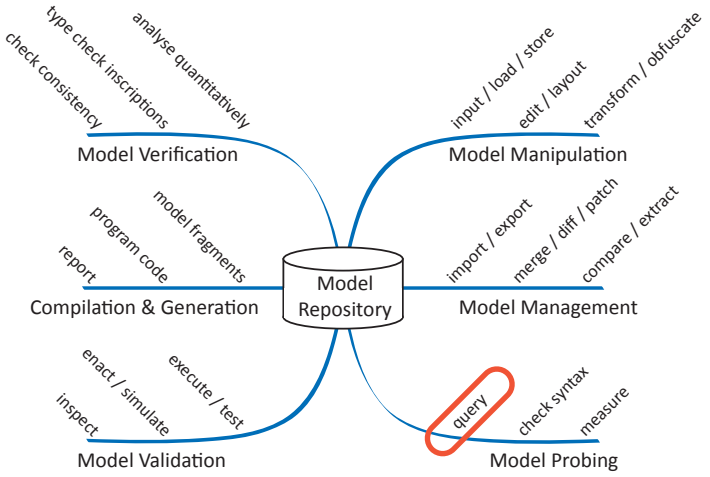


Figure 1: Classes of model operations (potentially) relevant to industrial modeling

working on the models. It is, on the other hand, also a basic building block underlying most other more complex functionalities like model transformations and model measurements. However, current tools and interfaces for model querying are either restricted in their expressiveness or require a high level of expertise in the underlying metamodels and/or query languages, both of which reduce their versatility. As the application of MDD is gaining more widespread acceptance and more and more developers are involved with MDD efforts, accessing models in an effective way is becoming a bottleneck.

Over the last years, we have created a system called the Model Manipulation Toolkit (MoMaT) which allows us to experiment with models in general. In MoMaT, models are imported into a Prolog-based model repository by a set of transformers for several modeling languages like UML, ARIS/EPC, and Use Case Maps (in this paper, we will only focus on UML models, though). Figure 2 provides an overview of MoMaT. In other papers, we have reported on using MoMaT for model version management operations (see [23, 24]).

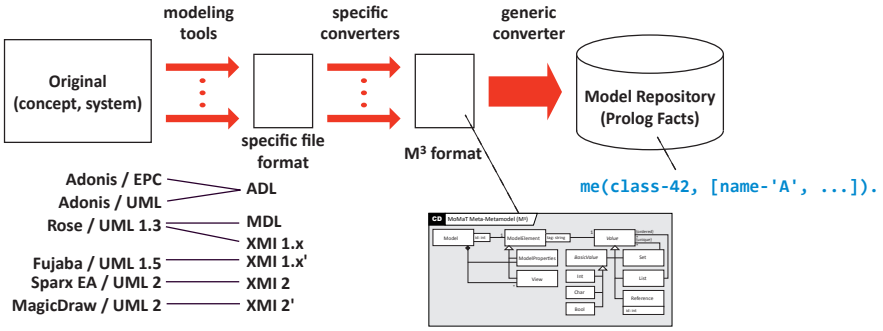


Figure 2: Schematic overview of MoMaT.

Our approach is derived from our experiences in two very large scale industrial projects with the German federal Tax Authority and the German Public Pension Authority. In these projects, very large models have been created and it soon turned out that without adequate query facilities they turn into "black holes" swallowing information but not giving it away again.

1.1 Related work

The related work may be subdivided into textual and visual query languages, and approaches that are specific to certain languages and/or tools or generally applicable (see the following schema).

	textual query language	visual query language
schema specific	query interfaces/ APIs, OCL [15]	QVT [19] Implementations like ATL, UMT [14], MOLA [9]), Porres' toolkit [18]
general	SQL, QBE, XPATH, SHORE [12]	Visual OCL [2], Constraint Diagrams [10], Query Models [22]

Most CASE tools provide APIs or predefined queries to allow users access to the models. Valuable as such facilities may be for the working software engineer, they are restricted to the specific settings in which they are implemented; application to other languages, tools, or data structures is difficult if not impossible. The OCL [15] is somewhat more general in that, theoretically, it should fit to any UML model/tool combination. In practical settings, this is not true, however. Also, OCL is extremely difficult to read and write[1] and there is very scarce tool support.

The OMG's Query-View-Transformations standard (QVT, [19]) has been created with similar goals as MoMaT. There are several implementations of QVT like the Atlas Transformation Language (ATL, `http://www.eclipse.org/m2m/atl/`), the UML Model Transformation Tool (UMT, see `http://umt-qvt.sourceforge.net/` and [14]), and the Model Transformation Language and tool MOLA, see [9]). All QVT implementations are based on the UML metamodel as their underlying data structure, which effectively excludes their application to non-UML languages. It also ties the respective tools to a particular version of the UML. The framework proposed in [18]) is a non-QVT system based on Tcl which the authors themselves deem applicable only for small and medium sized systems. None of these approaches are implemented in and using the facilities of Prolog. Also, to the best of our knowledge, none of them has been used successfully over a longer term in industrial applications.

When models are stored in a relational database, traditional relational query languages like SQL or QBE may be use to access them.[2] For XML data structures or databases, XPATH and similar approaches provide APIs with query facilities. The SHORE system (see [12]) is an approach to storing software design documents in a XML database using Prolog as the query language.

Visual OCL [2], Constraint Diagrams [10], and Query Models [22] each use a modeling language to specify queries for this language. By analogy, these approaches may thus used for other modeling languages. It is not clear, however, how such queries might be executed, much less, if used for a different language. While this paper does not yet fill this gap, it outlines a path toward this goal.

Besides this comparison scheme, Gruhn and Laue [7] present an approach where Prolog is used to access and query software engineering models, but also to represent them: they encode EPCs (in a rather ad hoc way) into Prolog facts which they then use

[1] A more detailed comparison between SQL, OCL, and MoMaT has to be deferred until we have introduced MoMaT.

to check some consistency conditions. This approach is limited to EPCs, unfortunately. There are also several approaches to encode programming language code in Prolog, e. g. for the Java language (cf. [21]), then define particular properties as Prolog rules and then check these properties by evaluating respective goals.

2 Model Representation

In MoMaT, models are represented as Prolog facts. More or less, every individual model element is represented as a single fact. The encoding into this representation is done in two steps (see Figure 2).

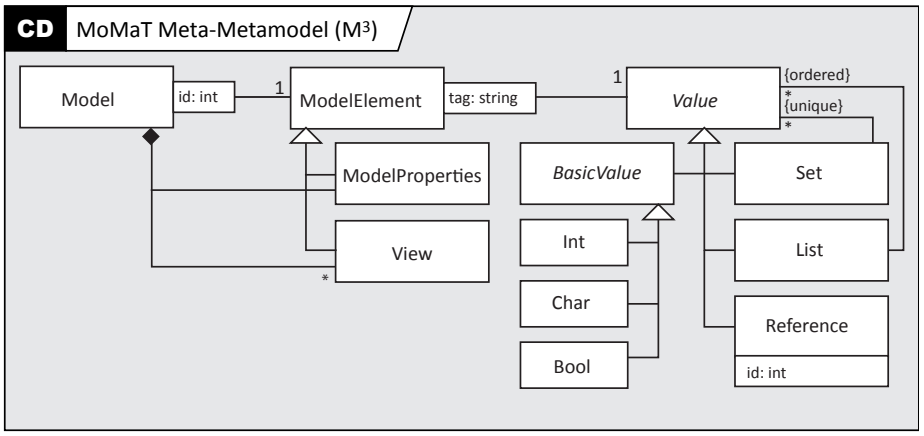


Figure 3: A Meta-metamodel as a normal form for arbitrary modeling languages.

First, specific formats are converted into a common format in order to accommodate different tools, languages, and formats. For instance, many modeling tools are capable of creating models in different languages, or of exporting models in different file format versions. Also, most tools will interpret standards in significantly different ways, will contain different specific bugs and so on, such that providing specific converters is inevitable for different tools. The common format is described in terms of a minimal, unifying meta-metamodel called the MoMaT meta-metamodel, or M^3 for short (see Figure 3). It can be seen as a least common denominator for a wide range of modeling languages.

For UML, the mapping into M^3 is simple indeed, since UML comes with a meta-model. All UML metaclasses are mapped to ModelElement. All meta-attributes and meta-associations a are mapped to tags, their types are mapped to the corresponding subclasses of Value (Reference for object types). The top level package of a UML model is mapped to Model.

The second step now simply interprets models in M^3 format as Prolog facts (see Figure 4). In MoMaT, each model element—that is, each instance of the class ModelElement of M^3—is represented as a Prolog clause of the form

```
me(type-id, [tag-value, ...]).
```

where `type` is the metaclass in the source language (such as “Feature” or “Class” in the

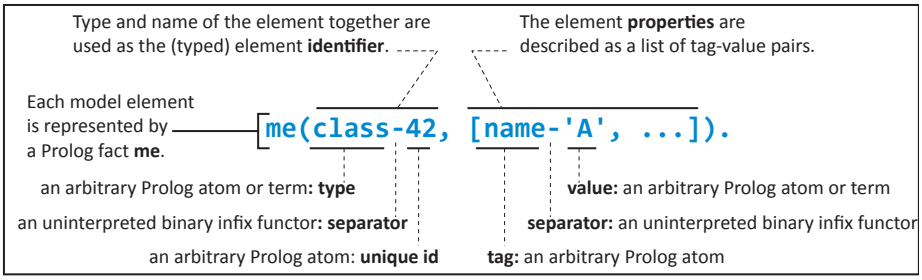


Figure 4: Representing model elements in Prolog

UML metamodel), `id` is an arbitrary unique identifier, `tag` is any atom representing a property of the model element, and `value` is the value for this tag. For instance, an abstract class with object identifier 42 and name "x" would be represented as

```
me(class-42, [name-x, isAbstract-true]).
```

This encoding is visualised in Figure 4. The value of an attribute may be an arbitrary Prolog-term.

A model is then simply a named container for a set of model elements. We use Prolog's built in module mechanism to represent models. In Prolog, modules may be defined just as well at compile time or at run time. Additional information pertaining to the model as such may be represented by a `model/2` clause. Similarly, views may be defined inside models with `view/2` clauses. The arguments of `view` and `model` are similar to those of `me`.

Figure 5 shows an example. Here, a model `m1` is defined. It is an analysis level model authored by user stoerrle and has reached the quality assurance status approved. It contains ten model elements and one view named `m1`. The view presents all model elements of the model.

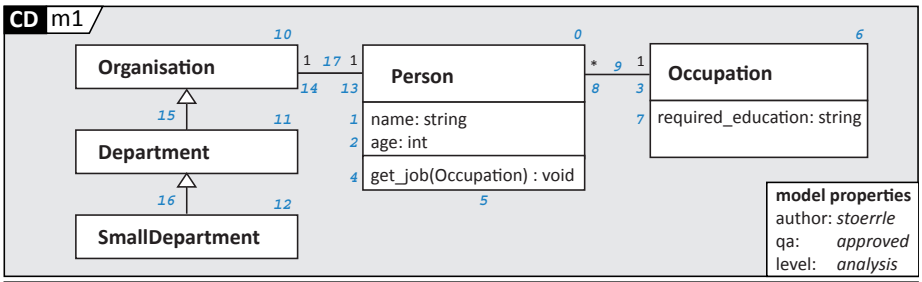
By using the Prolog module mechanism for representing models, all features of modules may be used for models as well, including nesting models, importing and exporting models, dynamic and static (i. e. compile time and run time) definition of models and so on.

3 Model queries

Simple queries select one or more model elements or their attributes based on basic selection criteria like identifier, name, or a complex combination of features. Based on the representation defined above, this may be achieved by Prolog queries like the following.[2]

```
1 ?- m1:me(METACLASS-0, VAL).
  METACLASS = class
  VAL = [name-'Person', attributes-ids([1,2,3]), operations-id(4)]
```

[2]This is actually a transcript of the SWI-Prolog top level slightly edited for readability. It is executed on the model described in Fig. 5. Recall that in Prolog, identifiers starting with an upper case letter are variables. The underscore is the don't-care-variable. Lists are enclosed in square brackets, In SWI-Prolog, `?-` is the top-level prompt. The Prolog idiom `me/3` declares that the predicate `me` is binary (and of course similar for all other attributes and arities).




```
:-module(m1, [me/2, view/2, model/2]).
   model(m1,  [level-analysis, author-stoerrle, qa-approved]).
   view(cd-1, [type-class_diagram, name-'Sample model',language-'UML 2',
               version-'2.1.1',elements-ids([0,1,2,3,4,5,6,7,8,9])]]).
   me(class-0,           [name-'Person',attributes-ids([1,2,3,14]),
                          operations-id(4)]).
   me(feature-1,         [name-name, type-string]).
   me(feature-2,         [name-age, type-int]).
   me(feature-3,         [multiplicity-1, type-id(6)]).
   me(operation-4,       [name-'get_job', parameters-ids([5]), result-void]).
   me(parameter-5,       [type-id(6)]).
   me(class-6,           [name-'Occupation', attributes-ids([7, 8])]).
   me(feature-7,         [name-'required_education', type-string, default-""]).
   me(feature-8,         [type-id(0),multiplicity-'*']).
   me(association-9,     [ends-ids([3, 8])]).
   me(class-10,          [name-'Organisation', attributes-ids([13])]).
   me(class-11,          [name-'Department']).
   me(class-12,          [name-'SmallDepartment']).
   me(feature-13,        [multiplicity-1, type-id(0)]).
   me(feature-14,        [multiplicity-1, type-id(10)]).
   me(generalization-15, [from-id(11), to-id(10)]).
   me(generalization-16, [from-id(12), to-id(11)]).
   me(association-17,    [ends-ids([14, 15])]]).
```

Figure 5: A sample transformation from a model containing a UML class diagram (top) into a Prolog module with clauses for each model element (bottom). The extra numbers in the class diagram refer to the identifiers of the Prolog code.

```
2 ?- findall(ID, m1:me(class-ID,_), IDS).
  IDS = [0,6,10,11,12]

3 ?- findall(ID-VAL, m1:me(class-ID,VAL), RES).
  RES = [0-[name-'Person', attributes-ids([1,2,3]), operations-id(4)],
         6-[name-'Occupation', attributes-ids([7,8])]]
```

The first query identifies the type and property set of element `0` inside model `m0`. The second query identifies all elements of type `class` using Prolog's built-in `findall/3` to obtain all solutions to the second argument with a single call. The third query selects all classes of model `m1`.

For the remaining examples, we need to introduce the predicates `get_me/4`, `part_of/4`, and `get_neighbours/4`, which are typical examples for predicates of the query-API of MoMaT.[3]

```
get_me(Model, Tag-Val, METACLASS-ID, VAL):-  % matches all elements of
  Model:me(METACLASS-ID, VAL),               % model containing the
  memberchk(Tag-Val,VAL).                    % Tag-Val pair
```

Using these predicates, the fourth query identifies the class named `Occupation` in `m1`. The fifth query identifies all features of type `string` in `m1`.

```
4 ?- get_me(m1, name-'Occupation', METACLASS-ID, _).
  METACLASS = class
  ID = 6

5 ?- findall(ID, get_me(m1,type-string,feature-ID,VAL), RES).
  RES = [1,7] ;
```

Query no. six identifies all elements related directly to element 6 by relationships of type `association`. There are very similar predicates for other relationships like containment, association, or generalization.

```
6 ?- get_neighbours(m1, association, 6, N).
  N = [0]
```

The predicate `get_neighbours/4` computes all neighbours of a given element that are related to this element by a particular kind of relationship as follows.

```
part_of(Model, Kind, SUPER, SUB):-             % gets ids of Kind-parts
  get_me(Model, Kind-ids(Parts), _-SUPER, _),  % of SUPER as SUB
  member(SUB, Parts).

get_neighbours(Model, Kind, Element, Neighbours):-
  get_me(Model, ends-ids(Features), Kind-_, _),
  get_me(Model, attributes-ids(ATTRS), class-Element, _),
  intersection(ATTRS, Features, [_|_]),!,
  maplist(part_of(m1,attributes),Containers,Features),
  select(Element, Containers, Neighbours).
```

Query no. 7 counts the number of dynamic and static features in a model, providing an example of how complex queries may be defined ad hoc on the command line.

```
7 ?- findall(OP, get_me(m1,operations-ids(OP),class-_,_), _OPs),
     findall(AT, get_me(m1,attributes-ids(AT),class-_,_), _ATs),
     count([_OPs, _ATs], Number_of_Features).
  Number_of_Features = 6
```

[3]The remaining MoMaT predicates are defined in the appendix. All other predicates are standard Prolog library predicates.

Of course, the queries presented so far have been rather straightforward. However, using Prolog we may execute also much more complex queries like "Identify all superclasses of a given class (transitively)" (see query 8 below).

```
8 ?- pre_closure(rels(m1,generalization),[12],[],C).
  C = [10, 11]
```

Similar queries may be used to identify all elements (transitively) related to a given element by a certain kind of relationships such as associations (query 9) and dependencies, e. g. in order to determine change impacts. A different query using similar techniques may select the shortest path of associations in a set of of classes, or the shortest path of DataFlowEdges between actions in an activity diagram. For lack of space, we cannot present the latter two queries in detail here.

```
9 ?- pre_closure(get_neighbours(m1, association),[6], [], N).
  N = [0, 10]
```

An interesting class of complex query are queries concerning more than one model. Our first example is the detection of design patterns, which can be implemented simply by a sub-model operation (query 10): all detectable patterns must be described structurally, in exactly the same way as our example is described. In this case, there is no occurrence of the composite pattern. Another frequent query is to determine all references from (elements of) one model to (elements of) another model (query 11), in order to trace change impacts across model boundaries.

```
10 ?- submodel(m1, patterns:composite, Mapping).
   Mapping = []

11 ?- findall(ID, Model:me(ID-_,_), IDS),  references(m1, REFS),
      subtract(REFS, IDS, EXT).
   REFS =[]
   IDS = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17]
   EXT = []
```

As our last example, consider query 12, where a sequence of models is scanned for the first model in which a given element is defined. Such a query might be used to track the introduction of errors related to some model element.

```
12 ?- sublist(exists(M_ID, 14),[m0, m1, m2], Exists),
      head(Exists,First).
   Exists = [m1, m2]
   First = m1
```

The predefined predicate `sublist/3` returns in its third parameter the sublist of the second parameter such that all the sublist's elements satisfy the predicate provided as the first parameter. Since the ordering of the original list is maintained, the first element of the resulting list is the first model in which model element `14` appears. `exists/2` is defined as `exists(ME_ID, M_ID):- M_ID:me(_-ME_ID,_)`..

4 SQL and OCL as alternative query languages

One may argue that the Prolog code necessary for implementing the queries proposed above is not very readable, and, in fact, Prolog as a language is too uncommon to

be considered for practical applications. There are mainly three alternatives for query languages: tool-specific predefined queries, APIs, SQL, and OCL.

Many commercial tools provide selections of predefined queries (e. g. Telelogic Tau, BOC Adonis). While sufficient in many situations, this approach is not (easily) extensible. Query APIs, on the other hand, are proprietary and may thus not be used in other tools.

SQL [4] is much more popular and widespread than Prolog. In fact, it is *the* paradigmatic query language. Thus, many tools storing models in (object-) relational database tables provide SQL-like facilities for querying (e. g. Aonix StP, BOC Adonis). Most of the queries we have presented above may be expressed in SQL3 (cf. [11]), but many database products do not implement this standard completely and faithfully. So, the following SQL-query is equivalent to query 8 above, but will not terminate on IBMs DB/2 or Oracle 11.

```
WITH RECURSIVE CLOSURE(ClassId, GeneralClassId ) AS
  ( SELECT
    FROM   CLASSES
    WHERE  ClassId = '1'
    UNION ALL
    SELECT cla.GeneralClassId, clos.ClassId
    FROM   CLOSURE clos, CLASSES cla
    WHERE  clos.GeneralClassId = cla.ClassId )
SELECT DISTINCT ClassId
FROM   CLOSURE;
```

Another obvious alternative model query language is Object Constraint Language (OCL, [15]). However, even simple OCL queries tend to be even more convoluted and less readable than Prolog code. See the following OCL equivalents of queries 2, 7, and 8.

```
2) package uml context Package
def: getAllClasses() : Set(Class) =
   self.packagedElement->asSet()->select ( t |t.oclIsKindOf(Class))
     .oclAsType(Class)->asSet()
endpackage

7) package uml context Package
def: getAllGeneralizations() : Set(Element) =
   self.getAllClasses().ownedElement.oclAsType(Element)
      ->asSet() -- Property (Association, Attribute), Generalization
endpackage

8) package uml context Class
def: DITantiCycle(list:Set(Class)) : Set(Class)  =
   if self.hasGeneralization()
   then
      if list->includes(self)
      then list
      else self.generalization.general.oclAsType(Class).
           DITantiCycle( list->union(Set{self}) )
             ->asSet()->asOrderedSet()->at(1))
      endif
   else Set{}
   endif
endpackage
```

The most compelling advantage Prolog has over OCL is of course the much better tool support available for Prolog, including a range of IDEs, debuggers, visualisation tools, efficient compilers and so on. For OCL, there are just a few tools like [8, 1].

5 Evaluation

5.1 Applicability and usability

While the roots of MoMaT lie in academia, it has been applied successfully in industrial settings (though only by expert modelers in touch with the author). The main benefit of our approach is in its technical simplicity and the high-level declarative style of programming in an interactive environment that supports an explorative mode of work.

We have applied MoMaT in a number of different settings concerning languages, formats, and tools:

- a large UML 2 subset (class, object, activity, assembly, and use case models) using ADONIS with proprietary ADL and XML formats;
- class and use case models using Fujaba and MagicDraw with different XMI formats;
- EPCs using ADONIS with the proprietary ADL format.

Models from all these sources may be processed using MoMaT, and we are very confident that we will have no problems processing any other type or format of models similarly. In fact, our approach seems to be unique in that it is applicable also to visual languages other than UML.

MoMaT is targeted at expert modelers and has been used by such people in industrial contexts successfully. First feedback by said users indicated that OCL would have been too complicated to be used. Of course, such subjective reports by people personally acquainted to the author are not representative. Proper evaluations comparing the usability of MoMaT with competing approaches are an open issue.

5.2 Performance comparison

Although we can not provide a complete evaluation of our approach im comparison to all other approaches mentioned in section 1.1, we have some inital measurements. A competing research group from our department is implementing an Eclipse/EMF-based OCL interpreter. In a model query shootout with them, we agreed on a set of eight simple queries[4], created a range of synthetic class models, and executed the queries on the models in both tools.

Classes	10	100	1,000	10,000
Model Elements	325	3,431	29,250	312,584
XMI file size of model [Mbyte]	0.06	0.61	5.42	56.5

The class models contained between 10 and 10,000 classes, each of which had a number of attributes (ranging randomly from one to nine). Over the whole range of models and queries, MoMaT had a consistent performance advantage of about factor ten. A detailed study into this rather unexpected finding has not yet been conducted. In particular, we have not yet evaluated memory consumption in detail. However, it seems that current OCL tools and Java's XML-libraries generally require significant

[4]The queries were: determine number of model elements, depth of inheritance tree, ratio of abstract classes, set of classes participating in exactly two associations, existence of a class with a specified name, number of instances of a given class, number of associations a given class participaates in, and number of neighbours associated to a given class.

resources. We have not yet compared MoMaT with with other OCL tools on the market but would be surprised to find significantly different results.

6 Conclusions

MoMaT provides a powerfull textual query facility for models expressed in arbitrary languages, provided there is a mapping from the languages conceptual design to the M^3. While MoMaT is textual in nature, it opens up a path to defining visual queries as well: as soon as there are is a facility of transforming a (incomplete) model from any given modeling tool into the MoMaT format, such model fragments may be used to find matching submodels using MoMaT. Thus, models may be used as queries in MoMaT, so if there is a tool to create models, there is also a tool to create queries. Of course, for practical appications we would need an integrated work bench, but that is just an implementation task.

References

[1] Dresden OCL toolkit. `http://dresden-ocl.sourceforge.net`.

[2] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. A Visualisation of OCL using Collaborations. In Martin Gogolla and Chris Kobryn, editors, *Proc. 4^{th} Intl. Conf. on the Unified Modeling Language (<<UML>>'01)*, number 2185 in LNCS. Springer Verlag, 2001. available at `tfs.cs.tu-berlin.de/vocl`.

[3] Ray J. A. Buhr. *Practical Visual Techniques in System Design. With Applications to Ada*. Prentice Hall, 1990.

[4] Chris J. Date. *An Introduction to Database Systems*. Addison-Wesley, 6^{th} edition, 1995.

[5] Rob Davis. *Business Process Modelling with ARIS: A Practical Guide*. Springer Verlag, 2001.

[6] David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. OMG Press, 2003.

[7] Volker Gruhn and Ralf Laue. Validierung syntaktischer und anderer EPK-Eigenschaften mit PROLOG. In Markus Nüttgens, Frank J. Rump, and Jan Mendling, editors, *Proc. 5. GI Ws. Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (EPK 2006)*, volume 224 of *CEUR Workshop Proceedings*, pages 69–85, Bonn, December 2006. Gesellschaft für Informatik.

[8] Christian Hein, Tom Ritter, and Michael Wagner. Open source Library for OCL (OSLO). `http://oslo-project.berlios.de/`.

[9] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model Transformation Language MOLA. In Uwe Aßmann, editor, *Proc. 2^{nd} Working Conf. Model Driven Architecture: Foundations and Applications (MDAFA 2004)*, pages 12–26, 2004. available at `www.ida.liu.se/~henla/mdafa2004`.

[10] Stuart Kent. Constraint Diagrams: Visualizing Invariants in Object-Oriented Models. In *Proc. Intl. Conf. Object-Oriented Programming Object Oriented Programming, Systems, and Languages 1997 (OOPSLA'97)*, pages 327–341. ACM Press, 1997.

[11] Jim Melton. *Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features*. Elsevier Science Inc., New York, NY, USA, 2002.

[12] Michael Meyer and Helge Schulz. SHORE: Ein Werkzeug für übergreifende Vernetzung und Auswertung von Dokumenten. In Franz Ebert, Jürgen und Lehner, editor, *Proc. Ws. Software-Reengineering*. Gesellschaft für Informatik e.V. May 1999.

[13] National Institute of Standards and Technologies (NIST). Integration Definition for Function Modeling. Technical report, Computer Systems Laboratory, National Institute of Standards and Technologies (NIST), 1993. available at `www.omg.org/techprocess/sigs.html`, see also `www.idef.com`.

[14] Jon Oldevik. UML Model Transformation Tool (UMT). Overview and user guide, v0.8. Technical report, SINTEF, 2004.

[15] OMG. UML 2.0 OCL Specification (OMG Final Adopted Specification, ptc/2003-10-14). Technical report, Object Management Group, October 2003. available at `www.omg.org`, downloaded at December 2^{th}, 2004.

[16] OMG. UML 2.1.1 Superstructure Specification (formal/ 2007-02-03). Technical report, Object Management Group, February 2007. available at `www.omg.org`, downloaded at May 25^{th}, 2007.

[17] MDA Guide Version 1.0.1. Technical report, Object Management Group, June 2003. available at `www.omg.org/mda`, document number omg/2003-06-01.

[18] Ivan Porres. A Toolkit for Model Manipulation. *Intl. J. Software and Systems Modeling*, 2(4), 2003.

[19] QVT-Partners. Revised submission for MOF 2.0 Query/ Views/ Transformations RFP (version 1.1, 2003-08-18). Technical report, August 2003. available at `www.omg.org/mda`, download at November 1^{st}, 2004, see also `umt-qvt.sourceforge.net` and `qvtp.org`.

[20] Bran Selic. The pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, September/October 2003.

[21] Daniel Speicher, Robias Rho, and Günther Kniesel. JTransformer – eine logikbasierte Infrastruktur zur Codeanalyse. In Rainer Gimnich, Volker Riediger, and Andreas Winter, editors, *Proc. 9. Ws. Software-Reengineering (WSR 2007)* , volume 27, pages 21–22, May 2007.

[22] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Query Models. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *Proc.* 7^{th} *Intl. Conf. Unified Modeling Language (<<UML>>'04)*, number 3273 in LNCS, pages 98–112. Springer Verlag, 2004.

[23] Harald Störrle. A approach to cross-language model versioning. In Udo Kelter, editor, *Proc. Ws. Versionierung und Vergleich von UML Modellen (VVUU'07)*. Gesellschaft für Informatik, May 2007. appeared in Softwaretechnik-Trends 2(27)2007.

[24] Harald Störrle. A formal approach to the cross-language version management of models. In Ludwik Kuzniarz, Miroslaw Staron, and Tarja Systa, editors, *Proc. Nordic Ws. Models Driven Engineering (NW-MODE'07)*. IT University of Göteborg, August 2007. to appear.

A Selection of MoMaT library predicates

```
closure(P, X, Closure):-
  pre_closure(P, [X], [], Closure1),!,
  union([X], Closure1, Closure2),
  sort(Closure2,Closure).
pre_closure(_, [], SoFar, SoFar):-!.
pre_closure(P, Args, SoFar, Closure):-!,
  maplist(P, Args, P_of_Args1),
  flatten(P_of_Args1, P_of_Args),
  subtract(P_of_Args, SoFar, New),
  append(New, SoFar, Next),
  pre_closure(P, New, Next, Closure).

external_references(Model, EXTERNALS):-
  findall(ID, Model:me(ID-_,_), IDS),
  references(m1, REFS),
  subtract(REFS, IDS, EXTERNALS).

references(Model, EID, REFS):-
  Model:me(_-EID,VAL),
  maplist(collect_ids, VAL, REFS0),
  flatten(REFS0, REFS1),
  list_to_set(REFS1, REFS).
references(Model, REFS):-
  findall(REF, refs(Model, REF), REF_LIST),
  flatten(REF_LIST,REFS).
refs(Model, REF):-
  Model:me(_-_,VAL),
  maplist(collect_ids, VAL, REF).

collect_ids([],[]):-!.
collect_ids([_-id(ID)|Rest], [ID|RID]):-
  collect_ids(Rest, RID).
collect_ids([_-ids(IDS)|Rest], ALL_IDS):-
  collect_ids(Rest, RID),
  append(IDS, RID, ALL_IDS).

rel(Model, Kind, From, To):-
  get_me(Model, from-id(From), Kind-_, VAL),
  memberchk(to-id(To), VAL).
rels(Model, Kind, From, Targets):-
  findall(To, rel(Model, Kind, From, To), Targets).
```

Visual Languages: A Matter of Style

Sacha Berger François Bry Tim Furche

Institute for Informatics, University of Munich, Germany
{sacha.berger,francois.bry,tim.furche}@ifi.lmu.de

Christoph Wieser

Salzburg Research, Austria
christoph.wieser@salzburgresearch.at

Abstract

Styling has become a widespread technique with the advent of the Web and of the markup language XML. With XML, application data can be modeled after the application logic regardless of the intended rendering. Rendering of XML documents is specified using style sheet languages like CSS. Provided the styling language offers the necessary capabilities, style sheets can similarly specify a visual rendering of modeling and programming languages. The approach described in this article considers visual languages that can be defined as a 1-to-1 visualization of (an abstract syntax of) a textual language. Though the approach is obviously limited by the employed style sheet language, its advantages are manifold: (a) visualization is achieved in a *systematic* manner from a *textual* counterpart which allows the same paradigms to be used in several languages and ensures a close conceptual relation between textual and visual rendering of a language; (b) visual languages are much *easier to develop* than in ad-hoc manners; (c) the capability for *adaptive styling* (based on user preference such as disabilities or usage context such as mobile devices) is inherited from Web style sheet languages such as CSS.

To make CSS amenable to visual rendering of a large range of data modeling and programming languages, this article first introduces limited, yet powerful extensions to CSS. Then, it demonstrates the approach on a use case, the logic-based Web query and transformation language Xcerpt. Finally, it is argued that the approach is particularly well-suited to logic-based languages in general.

1 Introduction

Styling has become a widespread technique with the advent of the Web and of the markup language XML. The success of style sheet languages such as CSS is based on the ability to separate the conceptual or logical structure of Web data (be it in HTML or XML format) from the visual presentation of that data. Such a separation is convenient for adaptive presentation of content based on user preferences or usage context (in particular, for human as well as machine users such as search engine bots), for agile management and rapid development of Web sites, and for separating the concerns of content and presentation.

Many of the reasons why styling has succeeded for visualizing data apply also to the visualization of *programs* (i.e., of data modeling and programming languages), though interactive features become possibly even more important. The advantages of styling for data are inherited: easy conception of new visual languages; adaptive styling allowing different presentations based on user, device, etc.; systematic relation between abstract concepts, visual, and textual rendering of the language limiting impedance mismatch when switching between different renderings of a language. A further advantage is that the approach inherently permits "round-trips": A program developed so far as text (visually, resp.) can be further developed visually (as text, resp.).

Obviously, this approach is limited by the capabilities of the style sheet language employed. We choose in this article CSS for its widespread use and impressive visualization abilities: recent developments in the area of Web design and rich interfaces for the Web as well as the development of CSS 3.0 demonstrate the versatility of CSS-based visualization. The days of strictly hierarchical visualization are over with features such as absolute positioning supported by all mainstream browsers. The only remaining limitations of CSS are the rather rigid box model (which makes, e.g., ad-hoc curves impossible) and the limited interactivity features. The first limitation is starting to get addressed by recent proposals to add free-style drawing to HTML and CSS (cf. `canvas` element). The resulting flexibility in visualization is demonstrated by applications such as Yahoo! Pipes[1].

A first step to address the limitations to interactivity is proposed in this article: a *limited, yet far reaching extension* to the style sheet language CSS that makes it better suited for the rendering of not only data but also programs where interactive behavior becomes even more central. This extension (as well as the entire approach) is demonstrated on a use case, the logic-based Web query and transformation language Xcerpt.

The visualization considered in this article is deliberately simple, so as to be realizable with a rather limited extension, called here CSS^{NG}, of the dominant Web style sheet language CSS. The generality of the approach should, nonetheless, become evident: Instead of CSS or CSS^{NG} a style-sheet language offering other visualizations could be used.

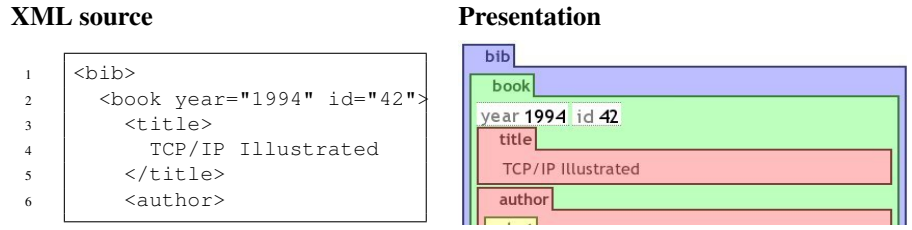


Figure 1: XML document (left side) and rendering using CSS^{NG} (right side).

CSS^{NG} is a novel extension of CSS 3, the latest version of CSS, introducing just a few novel constructs for *interactive or dynamic rendering* and for *markup visualization*. This limited extension of CSS 3 turns out to enable rather advanced visualization of programs. Even though CSS^{NG} is a limited and conservative extension of CSS, it adds considerably to the power of CSS allowing (a) to specify many forms of (interactive or) dynamic styling; (b) to generalize markup visualization; (c) to integrate the keyboard as input device (where CSS 3 mostly treats only a pointer input device such as a mouse).

[1] http://pipes.yahoo.com/pipes/

Thus, CSS^{NG} allows for a declarative, concise, and simple specification of dynamic document rendering in particular, when compared to current state-of-the-art techniques such as ECMA Script [10]. The same applies for markup visualization where currently far more complex technologies such as XSLT [13] must be employed.

2 CSS in a Nutshell

CSS 3 and its predecessors have been developed to simplify changes of the content as well as of the presentation of HTML and XML documents by separating content from presentation. It specifies formatting using rather simple guarded rules with formatting instructions. The following rule demonstrates a well-known *static* styling feature:

```
a        { text-decoration: underline; }
```

Intuitively, the rule reads as "**if** an element matches `a`, **then** format it by underlining its contained text". The left-hand, or *selector*, of the CSS rule selects HTML anchors (denoted as `a` elements). The *declaration* on the right-hand side assigns the styling parameter to XML elements matched by the selector of the rule.

Also, some *dynamic* styling features are offered in CSS 3. For instance, the background color of an HTML anchor can be switched to yellow while the mouse cursor is hovering (`:hover`) over it:

```
a:hover  { background-color: yellow; }
```

Markup especially in XML documents often conveys application relevant information (e.g., the role of a person associated with a book—author, editor, publisher, reviewer, etc.). Therefore, it might be useful to visualize it. However, CSS 2.1 and CSS 3 offer quite limited means for markup visualization which, in current Web application, often forces the use of other, less declarative technology to complement CSS such as ECMA script or server-side scripting languages. The following subsections 2.1 to 2.3 briefly introduce novel static CSS^{NG} rules mainly aiming at visualizing XML markup. Finally Section 2.4 introduces the rule-based interface for dynamic document styling. Full details on how CSS^{NG} extends CSS 3 can be found in [14].

2.1 Markup Insertion

CSS 3 allows the insertion of plain text specified in a CSS style sheet. The CSS emphpseudo-elements `::before` and `::after` cause insertion of text before and after a selected XML or HTML element.

CSS^{NG} extends these pseudo-elements of CSS 3. In addition to inserting plain text in CSS 3, the CSS^{NG} functions `element(NAME,ATTRIBUTES, VALUE)` and `attribute(NAME,VALUE)` provide in addition means for inserting XML elements and attributes before and after XML elements. The following example inserts a elements with a title-attribute of value "Tab" and content "element" before each element in an XML document. See Figure 3 how this can then be employed to visualize these new elements as "tabs" for hiding or unhiding information.

```
*::before { content: element("a",
                             attribute("title","Tab"),
                             "element") }
```

2.2 Markup Querying

CSS 3 provides the function `attr(`X`)` for querying the content of a known XML attribute X of an XML element. The name of an XML element and its XML attributes can not be queried. Implementing markup visualization as in Figure 1, i.e., where the name of an element is used as content of a newly created element to make the markup visible, without generalized markup querying means one rule for every XML element type like the XML `bib` element in Fig. 1.

CSS^{NG} adds the function `element-name()` yielding the name of the currently selected XML element. Furthermore, one XML element can host several XML attributes. Therefore, CSS^{NG} offers *attribute rules* selecting XML attributes instead of XML elements. The CSS^{NG} functions `attribute-name()` and `-value()` query XML attribute names and values in the context of a selected XML element. The example in Figure 2 implements a tab in front of each XML element listing the XML element name and all of the XML elements' attributes including their values as shown in Figure 1.

XML source (see Figure 1)

```
... <book year="1994" id="42"> ... </book> ...
```

CSS^{NG} style sheet

```
*::before { content:
  element("span",       element("span", element-name())
                  *  { element("span", attribute-name() " "
                                       attribute-value() )
                     } )
}
```

Resulting XML tree

```
... <span>
      <span>book<span>
      <span>year 1994</span>
      <span>id 42</span>
    </span>
    <book year="1994" id="42"> ... </book> ...
```

Figure 2: Generation of tabs. The presentation in Figure 1 is obtained by rendering the resulting XML tree using further CSS 3 means.

2.3 Depth-dependent Styling

Styling depending on breadth (i.e., on position among siblings) is planned in CSS 3 [7]. Tables, for instance, can be styled using alternating background colors for each line. CSS^{NG} additionally offers styling depending on the depth (i.e., position among ancestors) of an XML element in an XML document: `:nth-descendant(an+b)` restricts selections to XML elements having $an + b$ ancestors.

Figure 3 demonstrates the visualization of a highly nested XML document with colors repeating on every third level. On the left side this rendering is realized using

CSS^{NG} and alternatively using CSS 3. Thanks to its depth-dependent styling features, the upper CSS^{NG} style sheet needs only three rules. The CSS 3 style sheet below needs one rule for every level. Hence, styling in CSS 3 is possible up to a certain depth only as shown on the right side of Figure 3 using the CSS 3 style sheet on the lower right side of Figure 3. Such a styling is also useful for applications such as the visualization of threads in a discussion forum.

CSS^{NG}

```
1  *:nth-descendant(3n+1) { background-color: A; }
2  *:nth-descendant(3n+2) { background-color: B; }
3  *:nth-descendant(3n+3) { background-color: C; }
```

CSS 3

```
1  *                      { background-color: A; }
2  * *                    { background-color: B; }
3  * * *                  { background-color: C; }
4  * * * *                { background-color: A; }
5  * * * * *              { background-color: B; }
6  * * * * * *            { background-color: C; }
7
8  * * * * * * *          { background-color: A; }
9  * * * * * * * *        { background-color: B; }
10 ...
```

Presentation using CSS 3

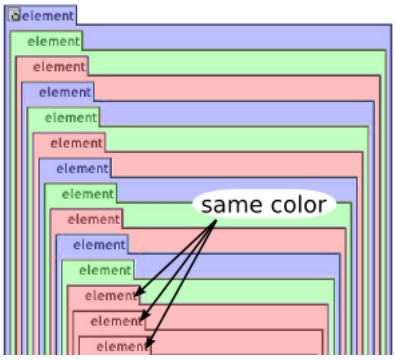


Figure 3: Comparing Depth-dependent Styling using CSS^{NG} and CSS 3.

2.4 Dynamic Styling Generalized

Dynamic styling is necessary to support (basic) interactivity, i.e., to change formatting (position, color, font, etc.) based on user input such as mouse clicks or move. CSS 3 is limited to the dynamic pseudo-class `:hover`. This construct allows dynamic styling in the local context of the mouse cursor only as demonstrated in Section 2. This is not sufficient to implement a behavior like folding a tab as demonstrated in Section 5: when the mouse cursor moves away, the cursor does no longer hover over the selected XML element, and its tab would be automatically unfolded.

CSS^{NG} introduces dynamic pseudo-classes for *all* HTML intrinsic events [1] such as `onclick` or `onkeypress` (see [14] for sample applications). Instead of using HTML intrinsic event attributes like for scripting languages, CSS^{NG} allows a standalone specification of dynamic styling in separate CSS^{NG} files that can be applied for multiple documents. The following example in Figure 4 shows a rather simple dynamic CSS^{NG} rule.

```
a:onclick(10) { background-color: green; }
```

Figure 4: Dynamic Styling of an adaptive hyperlink (CSS^{NG}).

The rule in Figure 4 implements an adaptive hyperlink. After 10 clicks on the hyperlink the background color changes to green meaning that the hyperlink on the Web page is often visited by a specific user.

This extension makes it possible to apply dynamic styling on different sections of an XML document at the same time. For instance if two hyperlinks were clicked ten times in a Web page, both will be presented with different background colors.

Similar extensions using HTML intrinsic events have been already proposed by the W3C [8]. The following paragraphs introduce the novel capabilities of CSS^{NG}:

Recurrence Patterns. All CSS^{NG} dynamic pseudo classes support *recurrence patterns*, `an+b`, as parameters. For instance the CSS^{NG} selector `*:onclick(3n+1)` detects the first, the fourth, the seventh, etc. click on an arbitrary XML element. More generally, a CSS^{NG} selector fires, if $an + b$ events occurred before.

On one hand such recurrence patterns allow to reuse CSS^{NG} rules for folding and unfolding as demonstrated in the following paragraph. On the other hand recurrence patterns allow to "delay" the application of rules up until a number of events, for instance clicks, as demonstrated in the previous Section (see adaptive hyperlink above).

Dynamic Styling Combined. A noticeable feature of the (novel) dynamic pseudo-classes of CSS^{NG} is their compatibility with CSS 3 *combinators*, which allow to specify tree patterns.

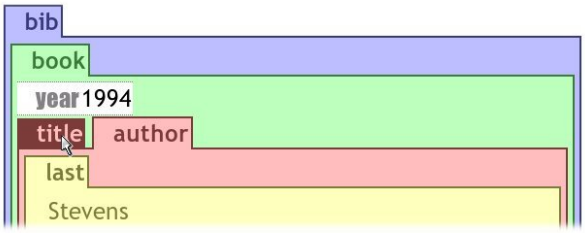


Figure 5: Folded visualization of an XML element `title`. The corresponding unfolded example is shown in Figure 1.

A CSS 3 selector is an alternating sequence of so-called *simple selectors* (already informally introduced in Section 2) and combinators. For instance, the combinator + means that the simple selector on its left side must be a preceding sibling of the simple selector on the righthand side. The CSS declaration (in curly braces) is only applied to the XML element matched by the matching simple selector.

The following example (see Figure 6) implements *alternating folding and unfolding* for the visualization of arbitrary (simple selector *) XML elements (see Figure 5). A click on a tab of a visualized XML element like `title` folds its visualization. Another click on a tab unfolds it (see `title` in Figure 1):

```
1  tab:onclick(2n+1) + * {display:none}     Fold on odd number of clicks.
2  tab:onclick(2n+2) + * {display:block}    Unfold on even number of clicks.
```

Figure 6: Combined dynamic styling in CSS^{NG} (rendering in Figure 5).

In the example above, the lefthand *selector* of the first CSS^{NG} rule above is composed of the two *simple selectors* `tab:onclick(2n+1)` and * combined with the CSS 3 combinator, +. The visualization of an XML element matched by the simple selector * disappears, if a mouse click was performed on its preceding sibling XML element, while its tab stays visible.

Structure-Independent Styling. A static CSS 3 styling rule is applied to all XML elements matching its selector. A dynamic CSS 3 styling rule is applied only to XML elements being in the context of an input device such as an XML element lying under the mouse cursor. CSS^{NG} abolishes this restriction and allows (novel) so-called *monorama* and *panorama* selections as demonstrated in Figure 7: The `Author` element on the left side is highlighted, while the mouse cursor is hovering over the `Author` element on the right side.

```
1  Author                { background-color: black; }
2  Author:hover ? Author { background-color: white; }
```

Figure 7: Highlighting of Xcerpt variables.

The CSS 3 rule in line 1 defines the standard background black for XML `Author` elements. In line 2 the CSS^{NG} combinator `?`, called *if*, is applied as follows: If an XML `Author` element is hovered in an XML document, set the background color of all XML `Author` elements to white.

A proof-of-concept prototypical implementation of CSS^{NG} was implemented as part of a diploma thesis [14] and presented [8].

3 Styling of Logic Languages

The approach described in the previous section to conceive a visual language as a rendering, or styling, of a textual language seems for the following two reasons especially convenient for logic languages:

- Logic languages are declarative, i.e. they focus on both the structural and conceptual organization of the data.
- Logic languages are often "answer closed" in the sense of query languages: queries or conditions resemble data and data (i.e., answers) can be used in place of queries. This makes style sheet languages developed for data visualization easily adaptable for program visualization since they are already able to visualize the data.
- Logic languages are often referentially transparent allowing mostly context-independent visualization of language constructs. In particular, this allows visual aids such as highlighting of related parts in a program or rule (e.g., variable occurrences or predicate symbols).
- Logic languages come in families that share traits, like e.g. modal languages, rule-based languages, logic programming languages, frame logic languages. With the approach proposed, "visualizations" can be rather easily developed and applied to various languages of a same language family.

For these reasons, it is the firm belief of the authors that the approach proposed in this article has the potential to boost the development and testing of visual languages, especially of visual logic languages.

4 visXcerpt — the Visual Twin Sibling of Xcerpt

As an example of the visualization of a textual language using the presented approach and CSS^{NG}, the Web query and transformation language Xcerpt [12] and its visual counterpart visXcerpt [3] are presented. Xcerpt is a rule based deductive language in the spirit of SQL or Datalog but for semi-structured data. As a textual language, it comes in two syntax flavors — an abbreviated syntax and an XML syntax. Rules consist of a head, also called construct pattern and a body consisting of logically connected query patterns. Query and construction share values by means of shared variables, rules query each other heads employing forward or backward chaining. Construct patterns may contain special grouping constructs to collect multiple variable bindings in one result, queries may consist of incomplete query patterns with incompleteness in breadth and/or depth and/or order, reflecting the incertitude about size and structure of documents on the web. Patterns are hence like *"examples"* of web data searched for in given documents.

The central part of visXcerpt, the visual rendering of Xcerpt, is the visualization of Web data, of XML documents. As Xcerpt itself comes in XML syntax, half the job is done by visualizing XML.[2] Further aspects, like partiality, grouping constructs and variables are then added to get a full featured visualization of query and construct patterns. Rules are just represented as horizontal aligned head and body, related by an arrow, though more involved visualizations (e.g., grouping by related root labels) can be realized with CSS^{NG}.

Term Visualization. Web data and patterns are considered to have a term like structure. Terms are rendered as boxes with their name as a tab on the top, the box contains all tabbed boxes of the subterms in the order they occur. The rendering is conceived to be suitable for most web browsers, considering that they are a wide spread technology with high adaptability to various screen sizes and resolutions. Order is given by a left-to-right and top-to-bottom flow layout, but the layout directions should be adapted to local writing habits of the user's culture. Width is given by the width of the display or browser employed. Nested boxes are further distinguished using colors, hence colors represent nesting depth. To be able to make a reasonable selection of well assorted, distinguishable and pleasant color themes, colors of upper levels are recycled for deeper nestings.

Graph Visualization On the Web, graph structures also need to be represented, e.g. RDF [11] data representing graph shaped structures or hyperlink structure. In textual representations of graph structures, references are used along a spanning tree of the graph. The presented approach of visualizing such graph structures is to model the references as hyperlinks in a web browser. This way, even very large graph structures can be represented and access to any references item is achieved by user interaction with constant complexity – a click on a hyperlink. While browsers often provide some means of navigating *back* along edges represented by hyperlinks, it is arguably useful to explicitly give hyperlinks for reverse traversal of edges, as hence the user is not restricted in his backward movement along edges he just visited.

[2] To some extent, this applies to any language as we can always consider for styling the XML serialization of the abstract syntax tree of a program.

Information focusing For large documents, it is of vital necessity to give users the ability to hide temporarily unneeded information or to focus on relevant data. This is achieved by means of folding in or out terms behind their name tagged tabs. While elements are aligned vertically, tabs are first aligned horizontally and then vertically, saving even more space. The concept is strongly inspired by tree browser visualization as e.g. seen on the well known Windows file browser.

At this level, pure static visualization starts to merge with user interaction. A visualization with adequate support of user interaction, especially of editing, is indeed much more useful than a static visualisation.

Textual Xcerpt Program, and **visXcerpt rendering of it.**

```
CONSTRUCT
 results[
  all result[
   var Title,
    var Author
 ] ]
FROM
 in(resource="file:bib.xml")
 proceedings04[[
  papers[[
   paper[[
    var Title as title[[]] ,
    var Author as author[[]]
   ]]
  ]]
 ]]
END
```

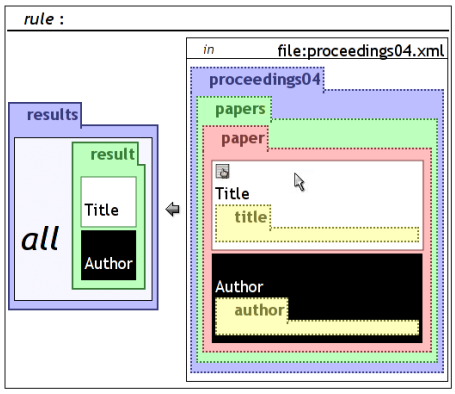


Figure 8: A single rule Xcerpt program (in abbreviated textual syntax) along with its visXcerpt rendering — the query part exploits a partial pattern (indicated by dotted lines in the visualization) to search for papers in a proceedings database, constructing title/author pairs all grouped in a list of results. All *Title* variables are highlighted as the mouse is hovering above one of them in visXcerpt.

A Special Purpose Editor Model For textual languages, copy-and-paste and text typing based editors are wide spread. Central to textual editing, is a cursor concept, that usually is a separator of the one dimensional program. For the presented visual approach, a separator seemed not intuitive, hence a context metaphor is used for editing: each box is a context, it is possible to cut, copy or delete it with or without its sub boxes, it is possible to paste the content of the cut/copy buffer into, before, after or around a context and hence term. The rich copy and paste model is accompanied by a template concept, giving access to all program constructs and possibly example terms or structures that can be altered, reduced or extended.

5 Realizing CSS^{NG}: CSS & XSLT

As a proof-of-concept, we chose to implement CSS^{NG} by a combination of XSLT transformations and reductions to standard XHTML and CSS to allow for maximum portability and fast implementation.

All data formats and transformations except **CSS^{NG} Parser** are based on W3C standards. Except for the CSS and CSS^{NG} parsers, all other program transformations are implemented in XSLT [13]. The XSLT transformations essentially evaluate the (static) rules in the CSS^{NG} stylesheet statically and adorn the XHTML elements to allow the use of standard CSS (and ECMA Script for the dynamic styling). The **Styler** is the heart of the system. It processes all XHTML elements in the document tree of an **(Un)styled Document** recursively. Each XHTML element passes through one test for each CSS^{NG} rule in a CSS^{NG} style sheet. If a test succeeds, the XHTML `style` attribute of the current XHTML element is modified. The tests are implemented in XPath [9]. Since tests are executed from the perspective of each XML element, CSS^{NG} selectors need to be translated to XPath selecting XML elements in reverse direction as demonstrated in the following example (see Figure 9):

CSS^{NG}

```
div:onclick(2n+1) + *
```

XPath

```
self::*/preceding-sibling::div [
   span[@class='onclick'] mod 2 = 1 ]
```

Figure 9: Translation of CSS Selectors in XPath (CSS^{NG}).

6 Outlook and Conclusion

The presented approach — obtaining a visual language by mere rendering or styling of a textual language — has been explored with the textual query language Xcerpt. To the largest extend, this has been achieved using standard CSS, for the most salient features however an extension of CSS has been conceived.

6.1 Conclusion

visXcerpt has been prototypically implemented and successfully applied for the presentation of Xcerpt [6] [5], widely easing the comprehension of the concepts of Xcerpt. visXcerpt's editor model turned out to be convenient for Xcerpt programming tasks from the area of HTML content extraction, creation and wrapping, over XML data transformation to Semantic Web and hybrid Web and Semantic Web reasoning [4].

CSS^{NG} as an extension of CSS turned out to be easily realizable without heavy computational overhead compared to CSS 2 and CSS 3. It proved itself to be not only a tool for the implementation of visXcerpt, but especially for sophisticated visualization of XML data with easily realizable domain specific behavior.

The approach of conceiving a visual language based on a textual back-end turned out convenient in both cases, for the creator of the visual language as well as for the programmer using the language — creating a visual language as a rendering of a textual one was reasonably easy, and programmers using it where pleased to be able to switch between textual and visual representation.

To the best of the knowledge of the authors similar generic approaches of developing visual languages as mere rendering using CSS and extensions have not been proposed so far.

6.2 Outlook

Further interesting research in the area of Xcerpt/visXcerpt is to investigate about type support, not only in the textual language for checking and validation of programs [2], but also in the editing process. This could help novice users to by just providing editing features that lead from one valid program to another, as well as providing a type based template approach over the example based approach. In the area of generic visualization of textual languages, it is needed to systematically investigate further features/functionalities that would be desirable for visual languages and what existing styling languages would be a convenient basis for adding these features. It would be interesting to develop a few style-sheet languages which could render various textual modeling and/or programming languages as visual languages after various visualization paradigms. The Semantic Web logic languages RDF, OWL and the new Rule Interchange Format (RIF) would be promising candidates for such investigations.

References

[1] S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, and S. Zilles. *HTML 4.01*. W3C, 1999.

[2] Berger, Coquery, Drabent, and Wilk. Descriptive typing rules for xcerpt. In *Proc. of 3rd Workshop on Principles and Practice of Semantic Web Reasoning*, 2005.

[3] S. Berger. Conception of a Graphical Interface for Querying XML. Diploma thesis, Institute for Informatics, LMU, Munich, 2003.

[4] S. Berger, F. Bry, O. Bolzer, T. Furche, S. Schaffert, and C. Wieser. Querying the standard and Semantic Web using Xcerpt and visXcerpt. In *Proc. of European Semantic Web Conference*, 2005.

[5] S. Berger, F. Bry, and T. Furche. Xcerpt and visXcerpt: Integrating Web Querying. In *Proc. of Programming Language Technologies for XML*, 2006.

[6] S. Berger, F. Bry, S. Schaffert, and C. Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. In *Proceedings of 29th Intl. Conference on Very Large Databases*, 2003.

[7] B. Bos. *Cascading Style Sheets Under Construction*. W3C, 2005.

[8] F. Bry and C. Wieser. Web Queries with Style: Rendering Xcerpt Programs with CSS-NG. In *Proc. of 4th Workshop on PPSWR*, 2006.

[9] J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0*. W3C, 1999.

[10] ECMA. *Standard ECMA-262, ECMAScript Language Specification*, 1999.

[11] O. Lassila and R. R. Swick. *Resource Description Framework*. W3C, 1999.

[12] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. of Extreme Markup Languages*, 2004.

[13] W3C. *Extensible Stylesheet Language (XSL) 1.0*, 2001.

[14] C. Wieser. CSS^{NG}: An Extension of the Cascading Styles Sheets Language (CSS) with Dynamic Document Rendering Features. Diploma thesis, Institute for Informatics, LMU, Munich, 2006.

Visualizing a Logic of Dependability Arguments

C Gurr
School of Systems Engineering
The University of Reading
Reading, UK
Email: C.A.Gurr@reading.ac.uk

Abstract

This paper offers general guidelines for the development of *effective* visual languages. That is, languages for constructing diagrams that can be easily and readily interpreted and manipulated by the human reader. We use these guidelines first to examine classical AND/OR trees as a representation of logical proofs, and second to design and evaluate a visual language for representing proofs in *LofA*: a Logic of Dependability Arguments, for which we provide a brief motivation and overview.

1 Introduction

Throughout the histories of both mathematics and engineering, diagrams have been used to model and reason about systems, whether physical or conceptual – such as logic. Both historical and more recent work in this area has given rise to a plethora of diagrammatic languages, often based upon a simple core language, such as "graphs" consisting of nodes with edges linking them. Graphs have the advantage of a very simple syntax and thus are easy to read (modulo good layout), yet are rather inexpressive and so, like many diagrammatic languages, are typically extended and embellished significantly. Such extensions often risk swamping the simplicity of the underlying graphs with overloaded symbolism and a confusion of textual annotations (for example, Fig. 1 versus Fig. 2). Ideally we would wish to know how, and how far, we might extend such attractively simple languages without losing their "salience". That is, their ability to be easily and correctly interpreted and manipulated by the human reader.

Assessing the salience of diagrams requires a theory of diagrammatic languages that explains how meaning can be attached to the components of a language both naturally (by exploiting intrinsic graphical properties) and intuitively (taking consideration of human cognition). The outlines of such a theory, constructed by analogy to theories of natural languages as studied in computational linguistics, were first introduced in [3, 4] and later drawn upon to offer guidelines for designing maximally salient diagrams and diagrammatic languages in [2]. This approach, dubbed "Computational Diagrammatics"[1], separates and clarifies issues of diagram morphology, syntax, semantics, pragmatics etc, facilitating the design of diagrammatic languages that maximise expressiveness without sacrificing readability.

[1]The author is indebted to Professor Michael A. Gilbert for first proposing this term.

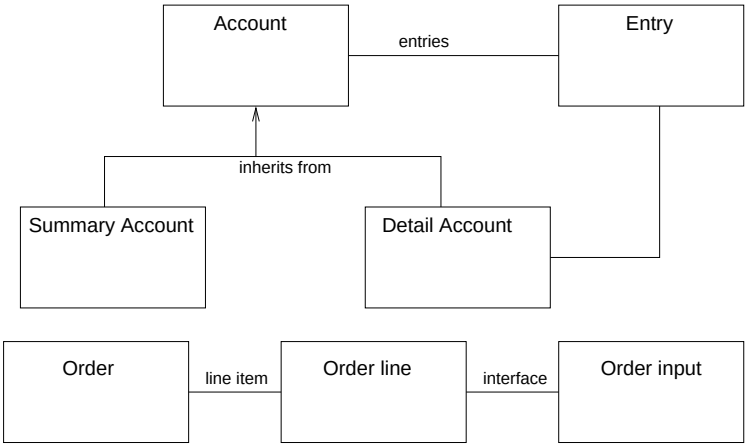


Figure 1: Simple UML Class diagram

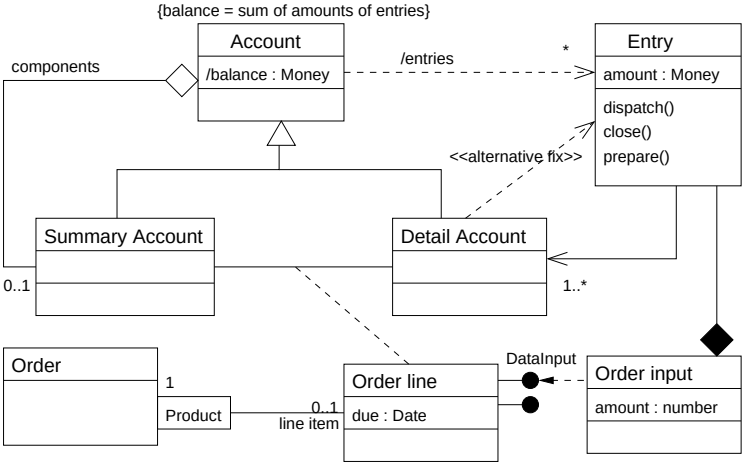


Figure 2: Enhanced UML Class diagram

There are typically many means by which a diagram, or diagrammatic language, may capture structure in a diagram. Naturally, when designing a new diagram or new diagrammatic language we would wish to determine which of the options available would be the most effective means of matching structure to content. That is, ensuring that the inherent structure in a diagram closely matches that of its semantic interpretation in a manner which is readily accessible to the reader.

This paper presents an overview of the visual language design guidelines proposed in [2] and an original analysis of the classical representation of logical AND/OR trees in the light of these guidelines, highlighting two significant issues. In Section 3 we present a specialised logic of argumentation first introduced in [1]. In section 4 we illustrate the application of our guidelines by developing an original visual representation language for this logic and discuss broader implications for for developing effective visualisations of more general logical arguments/proofs.

2 Effectiveness of Visual Representations

We may describe the syntactic components of a visual language as being comprised of basic graphical primitives and compound shapes. Graphical properties may be applied to the primitives. A part of what gives visual languages their power is the ability of: (i) categorizations of certain primitives, and (ii) the graphical properties; to carry semantic information.

Basic graphical primitives are shapes and lines. Compound graphical objects can be constructed from these primitives, two notable compound shapes being arrowed-lines (composed of a line and a shape – the arrow-head – at either end) and bordered shapes (composed of a shape and a line which traces the boundary of that shape). The latter type compound graphical object permits us to consider shapes with, for example, thick, textured and/or coloured borders.

2.1 Properties of Graphical Primitives

The primary properties of graphical objects, a variation of that suggested in [5], are: value (e.g. greyscale shading); orientation; texture (e.g. patterns); colour; and size. These are applied to lines and shapes as in Table 1. Examining these properties, we

	Valu	**Orie**	**Text**	**Colo**	**Size**
Line	lim	√		√	√
Shape	√	√	√	√	√

Table 1: Properties of Graphical Objects

note that *value* and *texture* are not clearly distinguishable when applied to lines. Lines may be of variable weight (e.g. dotted, dashed or solid) which we deem to be *value*. However, texture may only be applied if all lines are of significant width, in which case they should be considered as shapes rather than lines. We consider *size* applied to lines primarily to indicate their width, although we note that it may also be applied to their length. Finally, orientation is a property over which some care must be taken. Certain tokens deemed of different shapes are in fact the same shape in a different orientation, for example squares and diamonds.

We make the following observations of the characteristics of these properties:

Value (greyscale shading) is discrete and ordered. For lines we consider value to equate to dotted, dashed or solid; and thus its use is somewhat limited ("lim"). For shapes we note that the combination of (any pair of) value, texture and colour is non-trivial.

Orientation is continuous and ordered. As noted above, some care must be taken with orientation and its usage should be minimal for points.

Texture is discrete and nominal (non-ordered).

Colour in theory, the colour spectrum is continuous and ordered. However, this is not intuitive in human perception. Hence, as graphic designers are aware, colour is best used as set of easily discernible values which are thus interpreted as discrete and nominal.

Size is continuous and ordered. However, in many cases – particularly for lines, where size=thickness – the most perceptually effective use of size is with some small number (typically 3-7) of discrete, easily discriminable values. Furthermore, we consider here that points have only a minimal ("min") variation in size, as too great a variation would suggest that they are shapes rather than points.

2.2 Guidelines for Visual Language Design

An effective visual representation is one in which the desired content is readily accessible to the reader and in which desired reasoning tasks are as simple and straightforward as possible. As indicated above, a significant benefit of visual languages is that elements of their syntax can intrinsically carry semantic information. Hence we may define a visual language that is *well-matched*, meaning that its syntax is defined to maximise the desired semantic information that it carries. However, there are a variety of elements of visual language design, over and above the syntax, which may convey semantic information. A summary of these, and a set of guidelines is introduced in [2] as follows:

1. **Morphology as types:** define a clear partitioning of basic diagram shapes into meaningful, and readily apparent, categories according to semantic type.

2. **Properties of graphical elements:** utilise graphical properties such as colour, size and texture to further partition diagram elements into a more refined type structure.

3. **Matching semantics to syntax:** select diagrammatic relations to represent semantic relations for which they have matching intrinsic properties.

4. **Extrinsic imposition of structure:** Impose constraints, or add enhancements or augmentations, to account for those cases where no direct matching of syntactic and semantic elements can be found. However, do this with care to ensure that salience is not unacceptably diminished.

5. **Pragmatics for diagrams:** Utilise, and allow diagram developers the freedom to utilise, any and all as yet unused diagrammatic morphology, properties and syntax to embody further structure in diagrams.

Clearly, there will often be cases where a choice of options exist for matching the semantics of a language. Furthermore, certain choices will conflict with others; as with the combination of any pair of colour, texture and value indicated earlier. To assist in resolving some of these choices, when constructing a visual representation (or, more generally, a visual language) that we wish to be effective, we must consider:

1. **The audience:** who are the intend readers/users of the diagram? In particular, are they commonly from some domain with existing visual languages and hence have prior conceptions of the meanings of certain symbols or other elements of diagrammatic syntax?

2. **The task:** what content needs to be identified in a diagram, and what reasoning tasks is the diagram intended to support?

Note however, that as certain graphical properties and syntactic relations may interfere, often a balance or trade-off is required when selecting the most appropriate syntactic match for some semantic aspect. Experience in graphic design (e.g [6, 7]) suggests a rule of thumb that *task* concerns outweigh *semantic* concerns; that is – where a trade-off is required, the preference should be whichever option supports greater salience of task-specific features. For example, in certain electronic or logical circuit diagrams it is often considered acceptable to duplicate the representation of some node if this improves the layout by reducing the complexity and/or number of connections to that node. In this case a semantic consideration – that each node is uniquely represented in the diagram – is over-ridden by the desire to simplify the task of identifying and tracking connections.

Finally we note that typically, for any non-trivial semantic domain and intended tasks, not all information may be captured directly through diagram syntax. Consequently the use of *labelling languages* for labels which may potentially contain significant semantic information is necessary for most practical diagrammatic languages. However, in an effort to increase the expressiveness, the unprincipled use of sophisticated labelling languages can perturb the directness of a diagrammatic language. Examples are legion of languages which are diagrammatic at core, but have had their expressiveness so enhanced through sophisticated labelling languages that any benefit to readers' interpretation of the "diagrammatic aspects" is negated. Hence we issue the warning: treat labels with care.

2.3 Issues in Visualising Logical Proof

To illustrate issues raised through the application of the above guidelines in the visualisation of logic, consider a typical AND/OR tree for visualising a logical proof, such as the example of Figure 3 which represents the proof tree for the proposition *A* in the propositional theory:

$A \Leftarrow B \vee C.$
$B \Leftarrow D \wedge E.$
$C \Leftarrow F \vee G.$
$D \Leftarrow True, E \Leftarrow True, F \Leftarrow True, G \Leftarrow True.$

Note that for brevity the tree has been simplified by omitting the *True* leaf nodes.

With respect to the above guidelines for effective visual language design, there are two notable issues with this classical representation. The first is that the significant semantic distinction between AND and OR nodes in the tree is represented only with the relatively minor syntactic distinction of having a horizontal bar across the lines leading

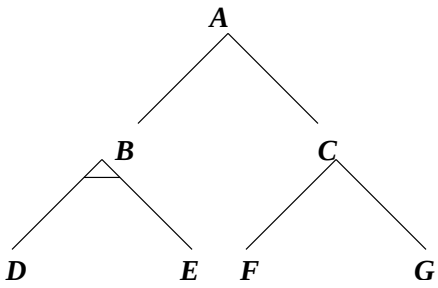


Figure 3: AND/OR Proof Tree

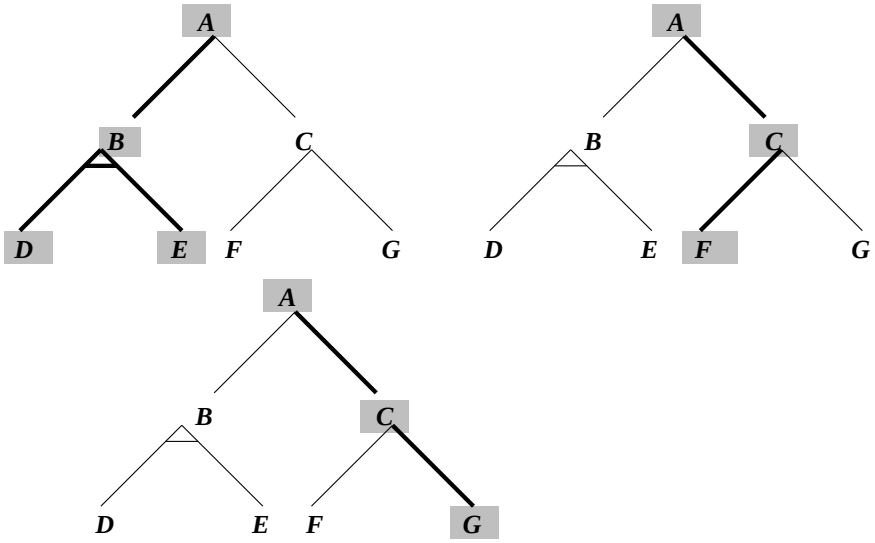


Figure 4: Disjunctive Proofs

to the children of an AND node – as with the $B \wedge E$ node in Figure 3. The second issue becomes apparent when considering the task of identifying the propositions that are active in any particular proof of the proposition A.

There are, in fact, three alternative proofs of the proposition A wrt the above theory. Visual languages are notably highly specific and generally do not lend themselves naturally to displaying such disjunctive information. Typically we have two options when displaying disjunction, either (i) we display multiple diagrams, one for each disjunct (as in Figure 4); or (ii) we introduce some new part of the notation which represents a semantic *abstraction* over the disjunctive information. As an example of this latter form, we might for example introduce a system of colour codings – where each disjunct is assigned a distinctive colour and each node and branch may be assigned multiple colours as appropriate. However, such an approach introduces further issues that make it undesirable for our purposes here.

In the example of Figure 4 we have chosen to represent disjunctive information (the alternative proofs) with multiple instances of the tree of Figure 3, each representing one

of the proofs. To assist the task of identifying which propositions are active in a single proof, we have used both *value* – that is, greyscale shading – to highlight the relevant nodes; and *size* – that is, thickness of line – to highlight the relevant branches in the trees.

3 *LofA:* A Logic of (Dependability) Arguments

Many classes of highly dependable systems, both computer-based and otherwise, are required to construct an argument to satisfy some, typically independent, third party that the system achieves some minimum specification or standard measure of dependability. That is, an argument that the system can be justifiably said to meet some specified level of – for example – security, reliability, quality or safety. A system developer's motivation for providing a dependability argument may be the desire to attain some internationally recognised quality standard for a business, a process or a product. Alternatively, as is often the case with safety- and security-related systems, there may be a mandatory regulatory requirement both to attain some specified standard, and to provide a dependability argument that argues the case for the attainment of that standard.

The logic *LofA: A Logic of Arguments*, introduced in [1], was developed to assist in the representation, evaluation and negotiation of dependability arguments. At heart, *LofA* is a relatively simple propositional logic using a Horn Clause representation, which is augmented with a number of non-logical warrants (derived from ideas in the Philosophy of Argumentation [8]) and an extensible set of feature values. An argument is a proof in *LofA*. Two key non-logical warrants are: (i) Argument by Authority; and (ii) Multi-Legged Argument. The most notable feature which can be utilised in a *LofA* theory is an expression of *confidence* in the argument. Typically, one imagines the author of an argument will assign confidence values to any evidence (leaf-nodes in a proof) and the confidence of a parent node in the argument (*LofA* proof) is automatically derived from its children. Further available features include temporal and numerical values (resulting in relatively simple propositional temporal and many-valued logics respectively) and also completeness values (useful for representing partially-constructed arguments).

The full expressiveness of *LofA* was first introduced in [1]. However, for the purposes of this exposition we shall consider only the simplest variant of the language, denoted *LofA*^{0}, in which propositional, definite Horn Clauses are supplemented with the above two non-logical warrants and the single feature *confidence*. For convenience we represent a propositional atom A and its associated confidence value c (a numerical value in the range 0–1) as the unary predicate atom $A(c)$. Thus a *LofA*^{0} theory is a collection of clauses of the form: $A \Leftarrow B_1 \wedge \ldots \wedge B_n$ where A is a unary predicate atom, as above, which we refer to as the *head* of an argument, and each of $B_1, \ldots, B_n$ is a *warrant*. A warrant is any one of:

1. A unary predicate atom, such as $P(c)$
2. $Auth(s, e, c)$: representing an 'Argument by Authority', where s is an authoritative statement by (presumed) expert e, and c its confidence.
3. $Multi(L)$: representing a 'Multi-Legged Argument', where L is a list $M_1, \ldots, M_n$ of warrants, supposedly offering alternative warrants for the proposition which this multi-legged argument itself warrants (in *LofA*^{0} these sub-

warrants are simply assumed to be independent, while more sophisticated variants of *LofA* insist on a supplementary argument to justify some measurable degree of independence of the sub-warrants).

4. $Ev(P, c)$: An item of evidence, where P is the predicate name of an atom representing some piece of evidence used to warrant an argument, and c the associated confidence value of this evidence.

Clearly, leaf nodes in a *LofA*^{0} argument are either *Auth* nodes or *Ev* nodes. The latter "Evidence" form of warrant is, in fact, a simple syntactic sugar. Rather than have evidence (i.e. an item that needs no further warrant) represented as:

$P(C) \Leftarrow true$ – where C is some constant

we have replaced, in the body of each clause, each occurrence of the atom $P(x)$ with the warrant $Ev(P, C)$. That is, we have simplified the clauses by application of a single resolution step.

The confidence value of proposition A in a clause $A \Leftarrow B_1 \wedge \ldots \wedge B_n$ is $c_1 * \ldots * c_n$, where $c_1, \ldots, c_n$ are the confidence values of $B_1, \ldots, B_n$ respectively. The confidence value c_n of a multi-legged argument of legs $M_1, \ldots, M_n$ with confidence values of $l_1, \ldots, l_n$ respectively, is defined recursively as $c_i = c_{i-1} + (1 - c_{i-1}) * l_i$ and $c_0 = 0$. For example, a multi-legged argument with three legs having respective confidence values $0.5, 0.8$ and 0.6, will have derived confidence value $0.5 + (1 - 0.5) * 0.8 + (1 - (0.5 + (1 - 0.5) * 0.8)) * 0.6 = 0.5 + 0.4 + 0.06 = 0.96$.

To illustrate the above, consider the following argument that *LofA* is a valuable argumentation tool:

> *LofA* is a valuable argumentation tool, as it is: (i) a relatively simple language; (ii) sufficiently expressive for the representation of dependability arguments; and (iii) has also shown itself to be "fit for purpose". Proposition (i) we consider self-evident; proposition (ii) is asserted as being true by the author; while proposition (iii) is justified both by the fact that a range of key dependability arguments have already been represented in *LofA*, and by the fact that *LofA* has proved an effective educational tool in the teaching of dependability argumentation.

Simplifying the above propositions for brevity, and ignoring confidence values for the moment, we may encode these in *LofA*^{0} as the following three clauses:

LofA_valuable $\Leftarrow Ev($*Simple*$) \wedge$ *Sufficient* $\wedge$ *Fit_for_purpose*.
Sufficient $\Leftarrow Auth($*"I say so", 'C Gurr'*$)$.
Fit_for_purpose $\Leftarrow$
$Multi([Ev($*Represented_key_arguments*$),\ Ev($*Effective_teaching_tool*$)]\ )$.

Suppose that we assert confidence values for the evidence and authority warrants as follows: *Simple* = 1, *"I say so"* = 0.9, *Represented_key_arguments* = 0.5, *Effective_teaching_tool* = 0.8; Then the derived confidence values of intermediary propositions will be: *Sufficient* = 0.9, *Fit_for_purpose* = 0.5 + (1-0.5)*0.8 = 0.5 + 0.4 = 0.9 and the derived confidence value for *LofA_valuable* is 1*0.9*0.9 = 0.81.

4 Visualizing *LofA* Arguments

In Section 2.3 we noted two issues with the classical representation of AND/OR proof trees: (i) that the visual distinction between AND/OR nodes was too weak; and (ii)

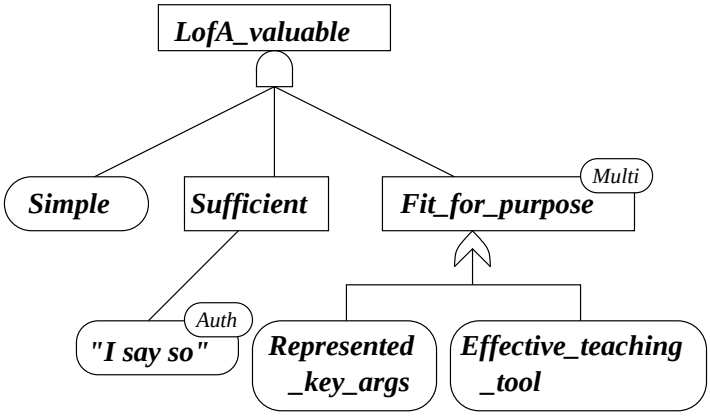


Figure 5: *Lofa* Argument Tree

that disjunctive proofs could not be represented in a single visualisation without introducing extra syntactic abstractions. Adopting the guidelines for visual language design summarised earlier, we have addressed these two issues in the visualisation of *LofA* arguments, as illustrated by Figure 5, as follows.

Firstly, to make clear the categorical distinction between differing node types, we adopt the convention of using categorically different node *shapes* to clearly distinguish the types. Hence, leaf nodes (Evidence and Authority nodes) are depicted as boxes with rounded corners, while non-leaf nodes (AND nodes and Multi nodes) are depicted by boxes with square corners. Furthermore, non-logical warrants (Authority and Multi nodes) are depicted as a compound shape, consisting of the appropriate box with a named box inset in the top-right corner. Finally, where a node has multiple children, the top of the branches leading to the children are clearly labelled with one of two shapes depending on whether they are an AND node or a Multi node – the latter being, effectively, a variant of a logical OR node. These two labelling AND- and OR-shapes are adopted from the classic representation of equivalent nodes in logical circuits. These have been chosen specifically to match the domain knowledge of the most likely readers of *LofA* arguments, who are typically familiar with this specific representation of AND and OR nodes from either logical or electrical circuits, or Fault Tree Diagrams, each of which are common in the assessment of highly dependable systems. Note also the somewhat more subtle variation in the representation of branches between AND and Multi nodes, that branches below AND nodes are drawn as (angled) straight lines with no bends, in contrast to those below Multi nodes. Thus we have used *orientation* to some effect here. In addition, for both AND and Multi nodes the branches below such nodes issue from a single point – and are subsequently split in the case of Multi nodes – rather than issuing from multiple points as is permitted in logical circuit diagrams. This is a further subtle, yet deliberate, indicator of the equally subtle point that the validity – as opposed to the confidence – of the argument represented by an AND or Multi node is equally dependent upon the validity of each of its children.

The second issue raised with AND/OR trees – that disjunctive arguments cannot be readily represented in a single diagram – is avoided in this visualisation of *LofA* arguments through the simple fact that such disjunctive arguments (where only a subset of propositions contribute to a given proof) do not arise in *LofA*. The *LofA* equivalent of

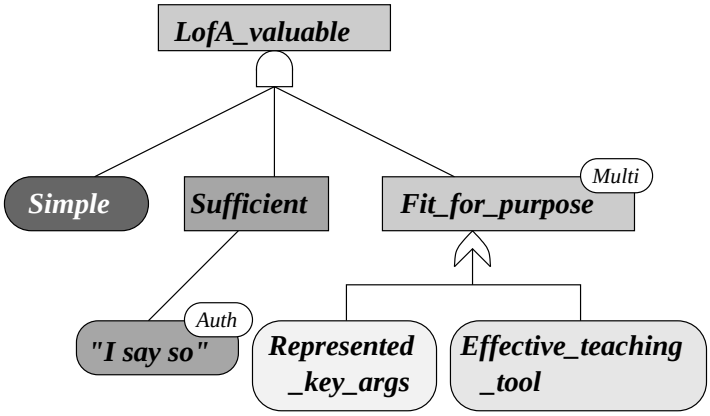


Figure 6: Visually annotated *Lofa* Argument Tree

the OR node is the Multi node, where sub-warrants each offer an alternative argument for the node, yet these are considered as a *collective* effort to increase confidence, rather than independent alternative justifications.

Having constructed this visual language for *LofA*, we may next explore how any currently unused graphical attributes – notably properties applied to graphical primitives – may be used to enhance the language wrt making certain tasks especially easy to perform. For example, consider the confidence values suggested above for our illustrative *LofA* argument. An assessor of this argument would wish to be able to readily identify both (i) what confidence is derived for the top-level argument, and (ii) how do subsidiary nodes contribute to this derived value? We may make such information readily visible in our language through application of an appropriate graphical property to the nodes. For example, using *value* (greyscale shading), where darker nodes indicate greater confidence, we may visually annotate the argument of Figure 5 to produce that of Figure 6. The choice of value to represent confidence in Figure 6 is appropriate as the primary semantic characteristics of confidence (that it is discrete and ordered) are well-matched by our choice of graphical property to represent it. Thus, for example, identifying which nodes in the argument of Figure 6 have greater or lesser confidence is an almost immediate process for the reader. Having made such a representational choice, however, constrains our further choices should we wish to extend the visual language.

Consider, for example, the feature of *completeness* with which we may extend $LofA^0$. In its simplest form, completeness is a binary value (complete/incomplete) applied to any leaf node in an argument. All leaf nodes in an argument are incomplete until determined to be complete by the argument's developer. A non-leaf node is complete iff all of its children are complete. This permits the argument developer to construct a *partial* argument. That is, one for which the structure of the argument is known, but for which not all evidence has been fully assembled and/or assessed. Extending the above illustrative $LofA^0$ argument, we could assert that the evidence node *Represented_key_arguments* is currently incomplete, and hence that its parent Multi node and the top-level argument are similarly incomplete.

We have a number of means available to add such completeness information into the argument representation of Figure 6 with a well-matched visual representation.

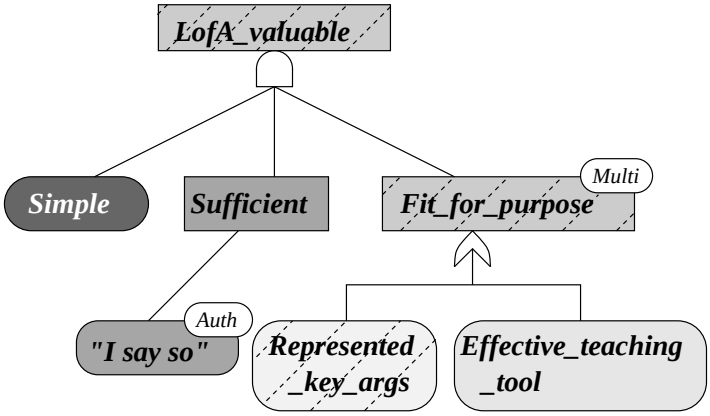


Figure 7: Further annotated *Lofa* Argument Tree

We might adopt a categorical colour coding, such as green for complete and red for incomplete. However, this choice may be problemmatic, as we have noted. Not least because colour can interfere with value, which we have already used, but there are also issues with the accessibility of colour to the human reader – both due to printing issues (this article is intended to be able to be reproduced in black and white) and concerns that not all human readers can discern all colours. Consequently we choose instead to adopt *texture* (that is, patterning) in nodes to represent incompleteness. In the argument tree of Figure 7 a value of *incomplete* for a node is indicated through a dashed patterning of the relevant node. Hence it is an easy matter for the reader to identify both which evidence nodes are incomplete, and how this impacts the (in)completeness of the remainder of the argument.

Finally, we consider briefly what further annotations could be made to argument trees such as that of Figure 7. We have applied no graphical properties to the branch-lines connecting nodes (other than limited use of orientation), so colour or size could be applied meaningfully there. Clearly, the use of colour in nodes also is a possibility – although as mentioned above this could only be used in certain situations and with the greatest of care. Making the orientation of nodes semantically meaningful would not seem wise, although we might make their size meaningful. However, care would be needed here also as employing significant variations in size of node would likely lead to issues with layout. Were the nodes to be considered compound shapes consisting of both the shaped node and its bordering line, we might apply further visual annotations to these borders – potentially both colour and size (thickness) – which would again permit us to represent further, well-matched, properties in a visually salient manner. Only last of all should we consider adding *textual* annotations to this representation; for example if we wished to depict the absolute (numerical) confidence values and not merely their relative values as currently displayed. As noted previously, adding increasingly semantically sophisticated textual labels is a very strong temptation to all visual language designers, but one which can very rapidly negate the intrinsic advantages of a *visual* representation.

5 Conclusions and Future Work

The study of dependability argumentation is both a rich and interesting area. Dependability arguments are often complex in nature, combining disparate forms and sources of evidence in detailed arguments, which must then be reviewed by, and negotiated with, a broad audience exhibiting diverse competencies and interests. The issue of *representing* dependability arguments is a pressing one, and current approaches – while often expressive – generally lack sufficient semantics or depth to provide the necessary support for the analysis and evaluation of arguments.

Adopting a formal approach to representing dependability arguments has many advantages, most notably in the precision and exactness that such an approach brings to questions of the meaning and validity of an argument. However, given the relative sophistication and detail contained in a typical dependability argument, the expressiveness required of any language for representing such arguments goes significantly beyond what typical formal logics presently offer. Ongoing work in reviewing practical dependability arguments illustrates what is required of a representation for it both to be sufficiently expressive, and to support the variety of evaluations and manipulations demanded in dependability domains. Studies of the representation of dependability arguments have much to learn from philosophical students of argumentation theory, an illustrative example being issues of validity and plausibility with regard to appeals to authority – a major aspect of many dependability cases.

Diagrams and new diagrammatic languages are frequently cited as being "natural" and "intuitive" notations, permitting "easy" and "accurate" communication of complex structures and concepts. Indeed, such claims have been readily applied to the many graph-based notations popular in dependability argumentation. However, the veracity of these claims is seldom tested and often the design of such diagrammatic languages follows no clear or obvious principles of usability, readability or effectiveness for the human user. We intend to continue to develop and evaluate our effective diagrammatic languages for representing dependability arguments. After all, it is clear that ultimate responsibility for acceptance or rejection of any dependability argument will always rest with a human agent or agency. As such, it is both in our interests and within our responsibility to ensure that such agents are presented with the most comprehensive, comprehensible and accessible representations of dependability arguments that it is within our power to provide.

References

[1] C Gurr. Argument representation for dependable computer-based systems. *Informal Logic*, 22(3):293–321, 2002.

[2] C Gurr. Computational diagrammatics: diagrams and structure. In D Besnard, C Gacek, and C.B. Jones, editors, *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, pages 143–168. Springer, 2005.

[3] C Gurr, J Lee, and K Stenning. Theories of diagrammatic reasoning: distinguishing component problems. *Mind and Machines*, 8(4):533–557, December 1998.

[4] C A Gurr. Effective diagrammatic communication: Syntactic, semantic and pragmatic issues. *Journal of Visual Languages and Computing*, 10(4):317–342, August 1999.

[5] R E Horn. *Visual Language: Global Communication for the 21st Century*. MacroVU Press, Bainbridge Island, WA, 1998.

[6] E R Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire CT, 1983.

[7] E R Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, 1990.

[8] D Walton. *Informal Logic*. Cambridge University Press, New York, 1989.