Tableaux for Diagrammatic Reasoning

Octavian Patrascoiu[1], Simon Thompson [1], and Peter Rodgers[1]
[1] *Computing Laboratory, University of Kent, United Kingdom*
{O.Patrascoiu, S.J.Thompson, P.J.Rodgers}@kent.ac.uk
http://www.cs.kent.ac.uk/rwd/

Abstract

Diagrammatic notations, such as the Unified Modeling Language (UML), are in common use in software development. They allow many aspects of software systems to be described diagrammatically, but typically they rely on textual notations for logical constraints. In contrast, spider diagrams provide a visual notation for expressing a natural class of set-theoretic statements in a diagrammatic form. In this paper we present a tableau system for spider diagrams, and describe an implementation of the system. In a software development context, the system allows users to explore the implications of design choices, and thus to validate specifications; beyond this, the tableau algorithm and system are of general interest to visual reasoners.

1. Introduction

Tableaux provide an intuitive mechanism for exploring the models and counter-models of logical formulas, and in particular they give mechanisms for deciding satisfaction and validity for a wide class of logics. To users, tableaux are of value not only as decision procedures but also by providing a mechanism by which a user can explore the consequences of a statement or set of statements.

This is particularly important when a statement is used as the specification of or a constraint on a software system. In software development, it is a well-known problem that specifications can suffer from incompleteness, inconsistency, or inappropriateness to the problem domain. It is therefore crucial that specification writers have the chance to engage and interact with their specifications in as many different ways as possible.

Obviously specifications should be checked for syntactic and type correctness, and this can be done in a routine way. In order to understand the semantics of the formulas, other mechanisms are needed. A decision procedure will allow a user to find out whether a specification is satisfiable, but this does not answer the question of whether the intention of the specifier has been realized. To achieve this it is necessary to tease out the significance of the formula, and specifically

- to investigate the possible models of the formula, and
- to explore the consequences of the formula: in other words, to discover its ‘implications’.

Tableaux can provide both of these for the language of spider diagrams. A spider diagram gives a diagrammatic representation of a statement about a finite number of sets, their membership and their interrelationships. For instance, in the context of specification such diagrams can be used to describe the relations between objects and classes.

The language of spider diagrams is equivalent to monadic predicate logic with equality [18]. It would therefore be possible to turn diagrammatic representations into textual statements and to apply decision procedures or tableau methods to the translations of diagrams. This would be perfectly adequate in the case of a decision procedure, but where feedback to the user is necessary – about the form of models, or the consequences of a formula, say – then it is crucial to work with a visual representation in order to provide recognizable visual feedback. Hence the system developed in this paper.

The paper begins in Section 2 with an overview of diagrammatic reasoning, tableaux and spider diagrams and their reasoning rules. Section 3 presents the central tableau algorithm for spider diagrams, illustrates it by a number of examples, discusses heuristics and optimizations and concludes by evaluating the system The conclusion reviews the work presented in the paper, and explores prospects for future work.

2. Spider Diagrams

The motivation for spider diagrams comes from the belief that visual representations of logical statements can aid understanding of the underlying meaning, and are more acceptable to people who are unfamiliar with standard textual mathematical notations. A further reason is that many essentially visual systems have to resort to textual notation for indicating logical expressions over visual diagrams. An example is the Unified Modeling Language (UML) [19], which is used for describing the design of object-oriented systems. UML is entirely diagrammatic, except for the language used to describe complex constraints on collections of objects: this is the Object Constraint Language (OCL) [12]. Researchers in diagrammatic reasoning are developing candidates for replacing OCL with a visual notation; spider diagrams are one such example.

2.1. Background

The work described here is performed as part of the Reasoning with Diagrams project [15], which engaged in developing spider diagrams and similar diagrammatic reasoning methods. Spider diagrams are an extension to work of Shin [16]. Shin presented formal systems of Venn-Peirce diagrams: Venn diagrams extended with annotations to indicate empty and non-empty sets. Venn-Peirce diagrams admit purely diagrammatic reasoning and Shin proved that they could be equipped with sets of logical rules that are both sound and complete.

In related work, Hammer [6] presented a sound and complete system of Euler diagrams [3]. Sound and complete sets of diagrammatic inference rules have also been developed for several systems of spider diagrams [7].

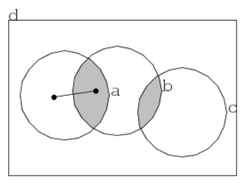


Figure 1 A spider diagram

Spider diagrams [5] are themselves a subset of the constraint diagram notation [8]. Spider diagrams represent the interrelationships and membership of a finite collection of sets. In Figure 1 diagram **d** contains three sets **a**, **b** and **c** which are represented by *contours* (simple closed plane curves). *Regions* are given by intersection, union and complementation, and a *zone* is a region which properly contains no other regions.

The figure contains regions corresponding to **a**$\cap$**b**, **a**$\cup$**b** and so forth but not to **a**$\cap$**c**, say; **a**$\cap$**b** is a zone, but **a**$\cup$**b** is not, since it properly contains the zone representing **a**$\cap$**b** (amongst others). The figure contains a single *spider*, which has two *feet*, and which *inhabits* the region **a**, with a foot in the zones **a**$\cap$**b** and **a-b**. Spiders have a single foot in each of one or more zones.

Applied to UML, relationships between classes and states can be expressed as contours. Constraints are represented as graphs where nodes appear in appropriate set intersections.

The interpretation of Figure 1 is given by three sets **a**, **b** and **c**, which are subsets of a universal set, U, say.

The absence of a region corresponding to a set-theoretic combination, such as **a**$\cap$**c**, implies that the combination must be empty: in this case the sets **a** and **c** must be disjoint.

Spiders provide lower bounds on the cardinality of sets: the spider that inhabits the region **a** implies that the region contains at least one element.

Shading is used to provide upper bounds. The shading of the zone **b**$\cap$**c** implies that **b** and **c** are disjoint. The shading of the zone **a**$\cap$**b** implies that it contains at most one element, that potential element being given by the spider with one foot in the zone.

There are no upper bounds on the cardinality of any unshaded zone in a spider diagram. In this particular case, it is possible for the set **b-(a**$\cup$**c)** to contain any number of elements (including none).

The diagram shown in the Figure 1 is *unitary*; a general spider diagram is given by a propositional combination – using conjunction, disjunction and negation – of unitary diagrams. A full formal definition of the syntax and semantics of spider diagrams is given in [18].

2.2 Semantic Tableaux

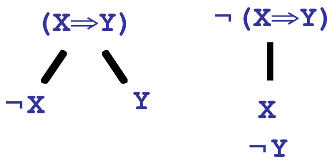
Semantic tableaux, [1] Section 2.6, provide an intuitive and efficient mechanism for deciding satisfiability and validity for a variety of logics. A semantic tableau for a formula is a tree, labeled at each node by a set of formulas: branches of the tree represent possible models for the formula.

The tableau for a propositional formula is built by repeatedly applying decomposition rules to any compound formula, until only literals and their negations (or *atoms*) remain.

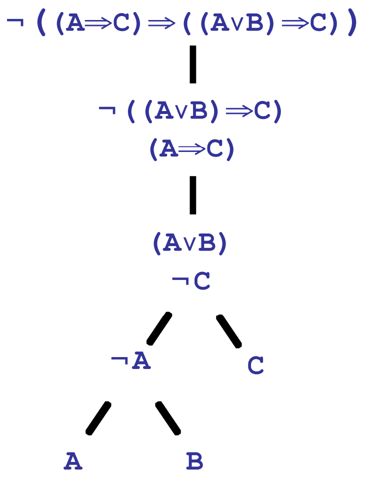
- A conjunction, such as A$\wedge$B, will be replaced by the pair of formulas A, B; this reflects that fact that any model of A∧B will have to make both conjuncts true.

- A disjunction like C∨D, will give rise to a split: one branch labeled C and the other D, reflecting that to satisfy a disjunction it is sufficient to satisfy one of the disjuncts.

Rules which do not cause a branch are called α–rules and branching rules are called β-rules. For each connective (e.g. implication, ⇒) there are two rules: one that decomposes the formula (X⇒Y) and the other decomposing its negation, ¬(X⇒Y). In this case, we have the rules:



Taking a larger example, we next draw the tableau for the formula ¬((A⇒C)⇒((A∨B)⇒C)). First we decompose the formula itself, giving the two formulas ¬((A∨B)⇒C) and (A⇒C). Either could be expanded, but it is usually sensible to apply α–rules before β–rules, thus delaying branching; we therefore expand ¬((A∨B)⇒C). At the next stage, two formulas remain, both with β–rules; we expand (A⇒C) and then (A∨B).

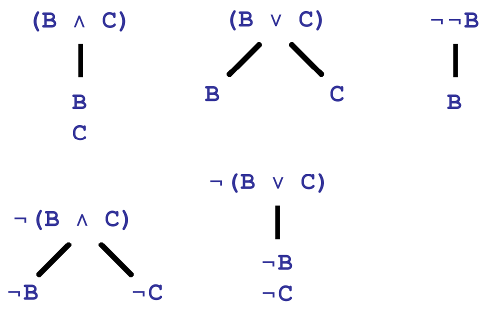


The tableau has three branches, and so embodies three potential models. Not all give models: consider the leftmost branch: that has atoms A, ¬A, C, which can't be satisfied simultaneously; similarly the rightmost branch is closed.

The central branch has atoms ¬A, B, ¬C, indicating that the formula at the root is satisfied when A and C are false and B true. Note that in building the rightmost branch it was unnecessary to expand (A(B), since the branch was already closed,

Building this tableau has shown that the root formula is satisfiable; we can also conclude that the un-negated formula (A⇒C)⇒((A∨B)⇒C) is not valid, since its negation is satisfiable. In this way, tableaux provide a decision procedure for validity as well as for satisfiability.

For completeness we include the rules for conjunction, disjunction and negation here:



Next we look at logical equivalences between spider diagrams which will form the basis of the extension of tableaux to spider diagrams.

2.3 Reasoning Rules for spider diagrams

To build tableaux for spider diagrams we use transformation rules that allow us to transform one diagram into another logically equivalent diagram by removing, adding, or modifying diagrammatic elements. The rules are summarized below; they are based on the rules given by Shin in [16], which developed earlier work of Pierce [13].

Rule 1: Add a contour. A new contour can be drawn inside a bounding rectangle without changing the meaning of the diagram if each zone is split into two zones, inside and outside of the new contour. Each foot of a spider is replaced with a connected pair of feet, one in each new zone. Shaded zones become corresponding shaded regions.

Rule 2: Add a zone. The rule is used to add a zone absent from a diagram. The added zone is shaded to indicate that it is empty.

Rule 3: Split a spider. If a unitary diagram *d* has a spider whose habitat is formed by *n* zones, then we may replace *d* with a disjunction of *n* unitary diagrams $d_1, \ldots, d_n$, each of which contains a one-footed spider inhabiting one of the zones touched by the spider *s*.

Rule 4: Expand negation. The explicit negation of a unitary diagram containing only one-footed spiders is replaced by a disjunction of (un-negated) unitary diagrams. The constraints placed on the models by shading and one footed spiders represent a conjunction of simple constraints; hence the disjunction resulting from expanding the negation.

Rules 1-4 provide the basis for diagrammatic reasoning with spider diagrams. Other rules used in building the tableau are the standard equivalences of propositional logic and compound rules built by iteratively applying combinations of rules 1-4.

Rule 5: Expand a compound diagram. This rule encapsulates the application of the tableau rules for propositional and adds the children associated by the reasoning process to ‘and’ and ‘or’ nodes. It also applies the de Morgan laws to transfer the negation on unitary diagrams. The children are computed according to propositional tableau rules [1].

Rule 6: Add contours. This rule applies Rule 1 repeatedly to add a list of contours to a collection of unitary diagrams.

Rule 7: Split spiders. Splits all the spiders used in a collection of unitary diagrams; Rule 3 is invoked several times.

Rule 8: Equalize contours. This rule is applied to a collection of unitary diagrams, which will appear in a number of different logical combinations within a tableau. Contours are added to the individual diagrams so that each diagram contains the same set of contours: the union of the initial contour sets. This rule is therefore equivalent to repeated application of Rule 1.

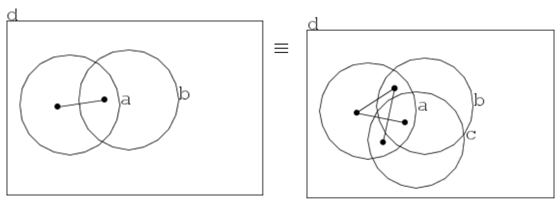
Rule 9: Equalize zones. This is the analogue of Rule 8 for zones rather than contours, and it corresponds to repeated application of Rule 2. Before adding an extra zone, contours in the diagrams need to be equalized.

Rule 10: Equalize diagrams. Invokes Rule 8 and Rule 9 to equalize both the contours and the zones.

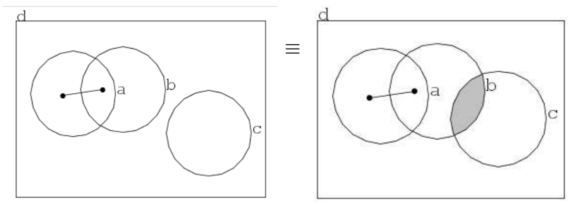
Rule 11: Expand all compound diagrams. Uses Rule 5 repeatedly until there are no more ‘and’ and ‘or’ compound diagrams.

Every unitary spider diagram is satisfiable; contradictions only occur in compound diagrams. In particular, from a unitary diagram we can read off a model by collapsing each spider to one of its feet and reading that as element of the model.

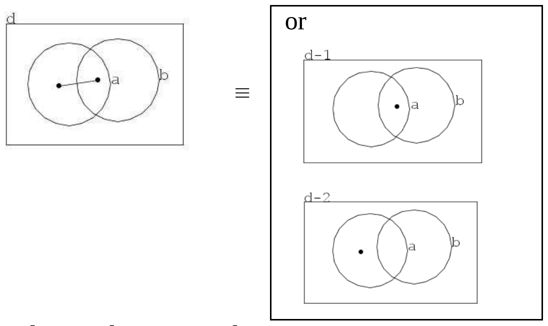
Contradictions can be explicit, as in the situation where a compound diagram contains both a diagram and its explicit negation; on the other hand, an implicit contradiction occurs in a conjunction of diagrams with conflicting constraints on a particular zone. Shading gives an upper bound on the cardinality of a zone whereas spiders provide lower bounds, and these two can conflict.



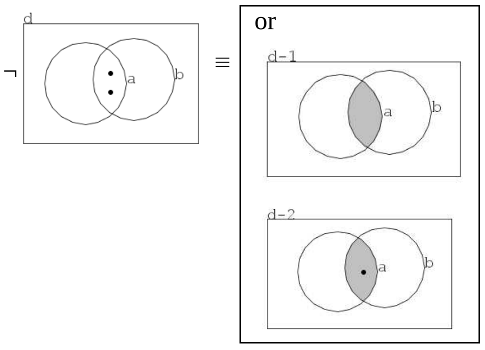
Rule 1: Adding contour *c* to diagram d



Rule 2: Adding zone b∩c to diagram d



Rule 3: Splitting a spider



Rule 4: Negate a unitary diagram

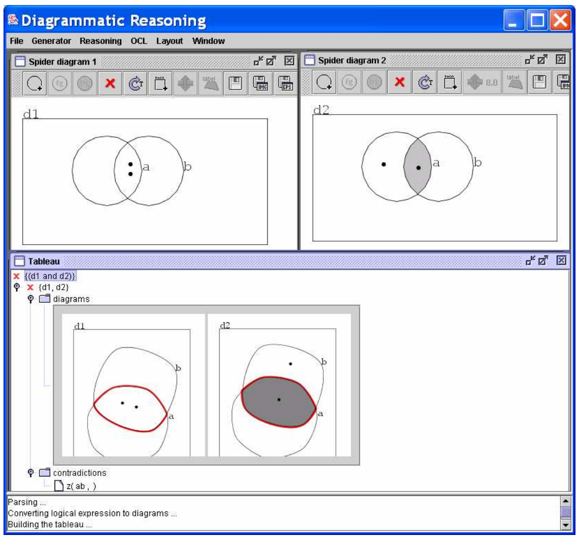
Figure 2 Illustrating Rules 1-4

3. System Definition

This section presents a tableau system for the diagrammatic reasoning framework presented in Section 2. We begin with the definition of some terms, and then we present the rules used in the system. Then we present the algorithm that decides whether a diagram is satisfiable and analyze the satisfiability of some formulas to illustrate the algorithm. In order to design the algorithm that builds the diagrammatic tableau for spider diagrams we refer to the work presented in [1], as a framework for propositional tableaux, and [4], which presents a reasoning system for spider diagrams.

As presented in Section 2.2, a tableau is a tree, labeled with sets of formulas at each node. When spider diagrams, with diagrams as literals, replace formulas it is necessary to present the tree in a different form. We have chosen to use the JTree mechanism, which presents trees using the 'file browser' metaphor.

The tableau system is shown in the screenshots in Figure 3 and Figure 4. The upper panes of the window show the constituent unitary spider diagrams of the diagram in question; in the lower pane the tableau is shown as a JTree. Figure 3 shows a contradiction between the two diagrams d1 and d2 by outlining in bold the zone ($a \cap b$) with contradictory constraints. Figure 4 shows the effect of equalizing contours and zones between two diagrams (Rule 10 above).



```
Parsing ...
Converting logical expression to diagrams ...
Building the tableau ...
```

Figure 3 Equalized single-footed literals

3.1. Definitions

Definition 1. A *literal* is a unitary diagram or the negation of a unitary diagram. A unitary diagram is a positive literal and the negation of a unitary diagram is a *negative literal*. Any diagram d is the *complement* of ¬ d and ¬ d is the *complement* of d. For any diagram d, (d, ¬ d) is a *complementary pair of literals*.

Definition 2. A diagram that only contains spiders with one foot is a *single-footed* diagram. Otherwise it is a *non-single-footed* diagram.

Definition 3. Two diagrams d_1 and d_2 are *equalized* if they contain the same set of zones and contours. Otherwise they are *non-equalized*.

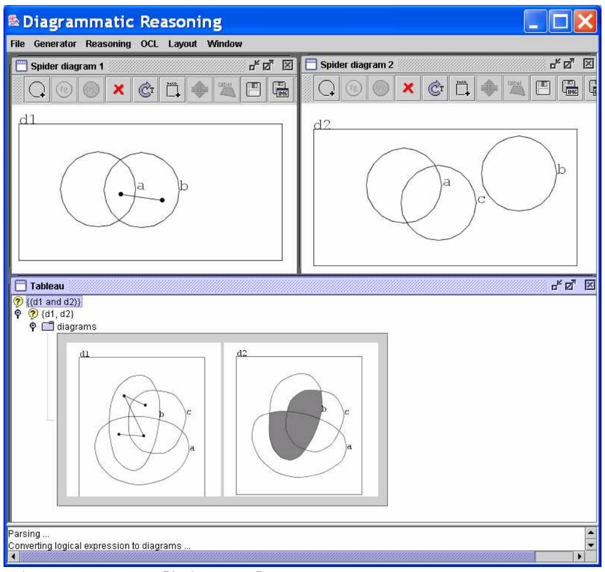
3.2. Algorithm Definition

This section presents a tableau algorithm for deciding satisfiability and hence validity for spider diagrams as presented in Section 2. This method extends semantic tableaux for the propositional calculus. We now give the construction of the semantic tableau for our diagrammatic reasoning system; the algorithm derives from the one presented in [1], Section 2.6.

Algorithm 1 (Construction of a diagrammatic tableau for spider diagrams)

Input: A diagram *d* of the spider diagrams calculus

Output: A diagrammatic tableau *T* for *d* with all the leaves marked.



```
Parsing ...
Converting logical expression to diagrams ...
```

Figure 4 Equalizing rule

A diagrammatic tableau *T* for *d* is a tree for which all the nodes will be labeled with a non-empty set of diagrams. At the beginning the *T* consists of a single node, the root, labeled with the set *{d}*. The tableau is built by choosing an unmarked leaf *l* labeled with the set of diagrams *D(l)* and applying one of the following rules. The construction terminates when all the leaves are marked with ⊙ or ✗.

- If *D(l)* contains at least one compound diagram, *choose* a compound diagram *d* from *D(l)*. Iteratively create children for leaf *l* applying the α- and β-rules rules for propositional logic, as presented in [1].
- If *D(l)* contains at least one negative literal, *choose* a negative literal *nd* from *D(l)*. Create children for leaf *l* applying the *negation* rule on *nd*.
- If *D(l)* contains only positive literals, and they are not equalized, create children for leaf *l* using the *equalizing* rule presented above, by which contours are introduced into diagrams.
- If *D(l)* contains at least one non-single-footed literal, create children for leaf *l* using the *splitting spiders* rule presented above, under which a diagram containing a spider is split into a disjunction of diagrams containing only single-footed spiders.
- If *D(l)* is a set of positive single-footed literals use the contradiction rules to mark the leaf *l*. If there is a contradiction among the diagrams from *D(l,)* the leaf is closed and marked with ✗. Otherwise, it is open and marked with ⊙.

The algorithm is not deterministic since during the expansion process of compound diagrams there is a choice of which formula to expand within the label of chosen leaf. Beside this, equalizing, negation, and spider-splitting rules generate compound diagrams, which generates non-determinism.

A diagrammatic tableau whose construction has terminated is called *completed diagrammatic tableau*. A completed diagrammatic tableau is *closed* if all the leaves are marked with ✗. If at least one leaf is marked ⊙, the diagrammatic tableau is *open*. Nodes below which the tableau is not completely expanded are marked ?.

The proof that the construction of a diagrammatic tableau terminates is straightforward, and is similar to the proof for semantic tableau in propositional logic [1][2]. A corollary of that result is that the order of application of the tableau rules does not affect the result of the decision procedure.

In practice, the construction of diagrammatic tableau can be made more efficient by using some of the ideas presented in [1]:

- Significant savings in space terms can be obtained if all the nodes share a diagram repository and reference elements using pointers.
- Using heuristics can make the tableau smaller. For example it is best to use α-rules before β-rules, and to split spiders only after the diagrams are equalized to avoid duplication of formulas.

Adding some derived rules will shorten the contradiction checking process. Examples include:

- If *D(l)* is a set of literals and contains at least a *false* diagram, the leaf is closed and marked with ✗. Otherwise other rules should be applied.
- If *D(l)* is a set of *true* literals, the leaf is closed and marked with ⊙. Otherwise other rules should be applied.
- If *D(l)* contains both a diagram and its negation, then the leaf should be closed and marked with ✗.

3.3. System Features

In this section we review the design of the tableau system, drawing attention to the various features and the motivation for their inclusion. A key aspect of a system of this sort is its usability, and to support this the system can be driven both automatically and with user intervention. We discuss system features in three broad categories now.

Logical aspects

Comprehensive set of literals and logical operators. The implemented system supports true, false, and user-defined unitary diagrams together with compound diagrams built using the propositional operators 'not', 'and', (inclusive) 'or', implication and equivalence. It can easily be extended to support other logical operators like exclusive or,'xor'.

Mixed visual and textual notation. Unitary diagrams are created using a diagram editor while the compound diagrams are described using a textual notation. A parser reads the textual notation and builds an internal representation. The internal model was built using Model Driven Architecture [10] and the Kent Modeling Framework [9].

User interaction

Viewing and editing diagrams. It is possible to view and edit a population of diagrams with ease. This can be particularly important when debugging well-formedness constraints expressed using diagrammatic languages.

Familiar browsing metaphor. The system uses the J-Tree library which provides a standard interface to tree structures such as file hierarchies. This model is familiar to users from file and directory browsers.

Application of rules over diagrams. It is possible to apply transformation rules over diagrams. Selective application is supported (e.g. adding a contour, a zone, or splitting a given spider), so that one can focus on particular diagrams, without being distracted by having to check ones which are not the current focus. The feedback from rule application has been designed to be as helpful as possible.

On-the-fly application of the rules. In developing well-formedness constraints it is often very useful to be able to experiment with constraints and sub-diagrams. The system is capable of reasoning about sub-diagrams that can be then integrated into a large-scale diagram.

Backtracking. The process of diagrammatic tableau construction is non-deterministic. So, at some point one might realize that the applied rule is not appropriate. The system allows the user to go back to previous step and choose another rule.

System implementation and visualization

Diagram layout. Diagrams can be displayed using automatic layout techniques [11][14]. However, the layout process is time-consuming and so is optional. Instead, the user can view a fast embedding of a diagram, which is poorly laid out, or just view the abstract syntax as a textual collection of zones and spiders.

Visual layout. Displaying trees is always a problem because on the one hand the number of nodes tends to grow on lower levels whilst on the other the graphical space is limited to a scrollable screen. In an attempt to deal with this, and to avoid under-use of screen real estate, the system provides a mixture of vertical and horizontal display directions: tree nodes which do not contain graphic information are displayed using the vertical dimension while graphic information is displayed on the horizontal dimension.

Syntax checking for diagrams. The system detects syntactic and semantic errors prior to the construction of the tableau. The graphic editor manages the syntax errors that appear in unitary diagrams. The parser is responsible for reporting the syntax errors in the textual description of the compound diagrams. This ensures the fact the system will process only well-formed diagrams.

Link between abstract and concrete levels. After a diagram has been read using concrete syntax notations, it is transformed into an abstract representation. The reasoning is performed at the abstract syntax level. The results obtained at the abstract level are then reported to the user at the concrete level. This increases the usability and the extensibility of the reasoning system.

Diagram storage. The system offers the possibility of persistent storage for diagrams. This is a useful facility, especially in the case of large-scale systems.

3.4. Example

Figure 5 contains the tableau after the expansion of the top-level "or" and some marking. A contradiction has been detected in one branch, in the zone shown with a thick border. However, the tableau as a whole is still in an "undefined" state: more rules need to be applied in order to decide if the tableau is open or not.

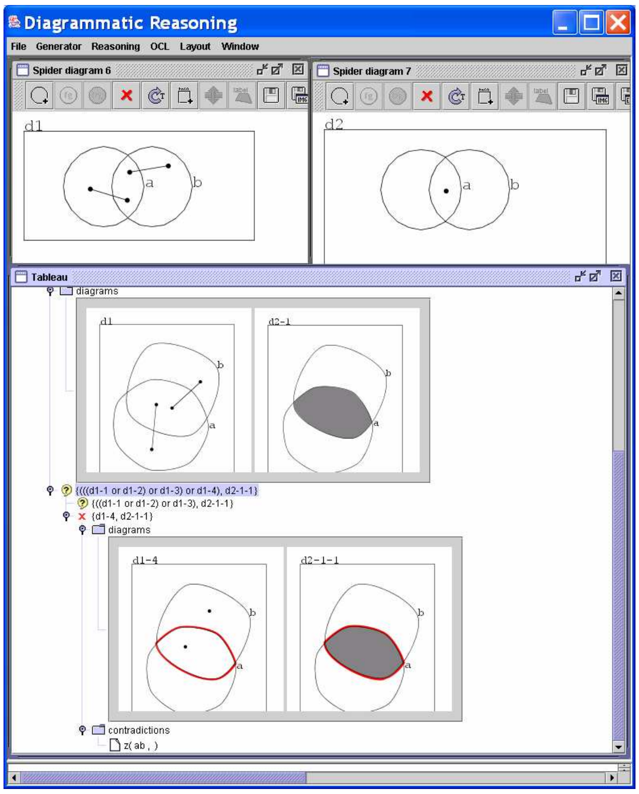


Figure 5 Indicating a contradiction between unitary diagrams

4. Conclusion and Further Work

In this paper we have described a system that supports reasoning with spider diagrams. This system allows users to construct their own spider diagrams and to explore the construction of a diagrammatic tableau. In addition, it can expand automatically all the compounds diagrams within a tableau node. The diagrammatic tableau is displayed using a mixture of vertical and horizontal display directions: tree nodes that do not contain graphic information are displayed using the vertical dimension while graphic information is displayed on the horizontal dimension.

Our plan is to extend the work in this paper to the considerably more expressive constraint diagram reasoning system [8]. Ideally, we will be able to design and implement an algorithm to construct tableaux for constraint diagrams. This is only possible for a decidable system. Restricted forms of the constraint diagram notation, which include arrows and universal spiders, yield decidable systems [17].

We also plan to use a heuristic approach to generate even shorter tableaux. The heuristic algorithm would search for an optimal operation to apply. If it fails to find a solution, it could be because more operations are required, or because there is no solution.

Currently the output from our tool appears in mixture of textual and diagrammatic notation. In order to present tableaux to users as a tree of diagrams we need to create concrete diagrams from their abstract descriptions. In [3] the authors give an algorithm for drawing a class of spider diagrams from abstract descriptions. The quality of the diagram layout has been improved using iterative methods and layout metrics [11][14]. More research is required on drawing strategies for diagrams in the context of tableaux so that the diagrams appear sufficiently similar after rule application.

Acknowledgement. This work has been funded by the UK EPSRC (Engineering and Physical Sciences Research Council) under grants GR/R63509/01 and GR/R 63516/01.

5. References

[1] Ben-Ari M. Mathematical logic for computer science. Springer-Verlag 2001.

[2] Fitting M. First-order logic and automated theorem proving, Springer-Verlag 1996.

[3] Flower, J., and Howse, J. Generating Euler diagrams, In Proceedings of Diagrams 2002, pages 61-75, Springer-Verlag, 2002.

[4] Flower J, G. Stapleton G. Automated theorem proving with spider diagrams CATS’04, Computing: The Australasian Theory Symposium , Dunedin , New Zealand, January 2004.

[5] Gil J., J. Howse, and S. Kent. Formalising Spider Diagrams, Proc. IEEE Symp on Visual Languages (VL99), IEEE Press, 130-137. 1999.

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

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

[8] Kent S. Constraint Diagrams: Visualizing Invariants in OO Modelling. In Proceedings of OOPSLA97, pages 327-341. ACM Press, October 1997.

[9] Kent Modeling Framework. www.cs.kent.ac.uk/projects/kmf

[10] Model Driven Architecture www.omg.org/mda

[11] Mutton P., P. Rodgers, and J. Flower. Drawing Graphs in Euler Diagrams. Proc. Diagrams 2004, pages 66-81. Springer-Verlag LNAI.

[12] Object Constraint Language. Object Management Group http://www.omg.org document ad/03-01-07.

[13] Pierce C., Collected Papers. Vol. 4. Harvard University Press.

[14] Rodgers P., P. Mutton and J.Flower. Dynamic Euler Diagram Drawing. Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'04), pages 147-156. IEEE, September 2004.

[15] The Reasoning with Diagrams project: www.cs.kent.ac.uk/projects/rwd/

[16] Shin S-J. The Logical Status of Diagrams. CUP. 1994.

[17] Stapleton G., J. Howse, and J. Taylor. A constraint diagram reasoning system. In Proceedings of Distributed Multimedia Systems, International Conference on Visual Languages and Computing (VLC '03). pp. 263-270, Miami, USA, 2003.

[18] Stapleton G., J. Howse, J. Taylor and S Thompson. What Can Spider Diagrams Say? Proc. Diagrams 2004, pages 112-127, Springer-Verlag LNAI.

[19] Unified Modeling Language. Object Management Group http://www.omg.org.