SparQ User Manual V0.8

Diedrich Wolter
Jan Oliver Wallgrün
Frank Dylla
Lutz Frommberger

Further contributions by:

Dominik Lücke   Thomas Schneider

Contents

About SparQ...

SparQ is a toolbox for representing space and reasoning about space based on qualitative spatial relations. Its development started within the R3-[Q-Shape] project of the Spatial Cognition Research Center in Bremen, Germany in 2006. Financial support by the the Deutsche Forschungsgemeinschaft (DFG) is gratefully acknowledge. By now, contributors have moved on and so has SparQ.

SparQ disseminates results from the qualitative spatial and temporal reasoning community which consists of researchers from a various disciplines including computer science, artificial intelligence, geography, philosophy, psychology, and linguistics. So far, a multitude of formal calculi over sets of spatial relations (like 'overlaps', 'left-of', 'north-of') have been proposed, focusing on different aspects of space (mereotopology, orientation, distance, etc.) and dealing with different kinds of objects (points, line segments, extended objects, etc.). SparQ aims at making these qualitative spatial calculi and the developed reasoning techniques available in a single homogeneous framework that is released under the GPL license for freely available software and can easily be included into applications. Our aim is providing a common toolbox spanning across all techniques of qualitative reasoning, thereby providing a universal toolbox to the user. Currently, we provide techniques for

- specifying qualitative formalisms and analyzing them
- interfacing the continuos domain with qualitative representations
- manipulating symbolical propositions
- reasoning with symbolical propositions, in particular checking consistency of qualitative information
- interfacing with other reasoning tools

SparQ is designed for the application designer or researcher working in a field other than qualitative reasoning, offering access to reasoning techniques in an easy-to-use manner. If you are new to qualitative spatial reasoning, see the following chapter for an introduction to this field and the services it can offer to your field.

SparQ is also designed for the researcher working on qualitative spatial and temporal reasoning (QSR). It provides an implementation toolbox of key techniques in QSR, making experimental analysis easier. Furthermore, SparQ offers an easy format to specify new calculi. This eases distribution of new calculi and enables researches to more easily compare different calculi, for example in an application context. In this manner SparQ is a community effort: it provides a rich repertoire of qualitative calculi to application designers. We would be happy to include your calculus! Last but not least, SparQ offers tools for analyzing QSR calculi, thereby supporting the calculus designer.

This document provides answers to four topic areas:

- installation of SparQ
- brief introduction to the field of QSR
- reference of SparQ commands and calculi specification syntax
- brief description of provided calculi

For questions or feedback, please get in contact with us an e-mail to the address below. We are always interested in suggestions for improvement and in hearing about your experience with SparQ.

The SparQ team
`qshape@sfbtr8.uni-bremen.de`

License

1. Installing SparQ

SparQ is built using several standard tools available for a variety of operating systems. SparQ is written for POSIX systems. Its functionality is continuously tested on Linux, Solaris, and Mac OS X, but it should work on any Unix system.

1.1. Requirements

SparQ is currently not available in binary versions. For installation some freely available standard tools are required. Besides its calculi specifications as plain text, SparQ comprises a set of C libraries and a main program written in Lisp. For compilation we rely on availability of these tools:

- Steel Bank Common Lisp (SBCL)[1], version 0.9.10 or higher
- gcc and g++, version 2.95 or higher
- GNU libtool, version 1.4.3 or higher[2]
- LaTeX for typesetting this manual

1.2. Building the Executable

To build a running version of SparQ, unpack the source code package, enter the newly created SparQ directory (called `sparq-<version>`) and run

```
$ ./configure
```

followed by

```
$ make
```

The executable will be installed within the SparQ directory. Please note that you have to recompile SparQ if you move the directory to another place.

If you encounter any problems during the build process, please contact the authors.

[1] `http://sbcl.sourceforge.net/`

[2] On Mac OS X a newer libtool version may be required, otherwise a manual patch may be necessary. See INSTALL notes for details.

2. Reasoning with Qualitative Spatial Relations

In this section, we provide a brief introduction on qualitative spatial reasoning and explain the most important terms required when dealing with qualitative spatial calculi in SparQ. For more in-depth introductions to the field, we refer to Cohn and Hazarika (2001), Cohn (1997), Ladkin and Reinefeld (1992), Ladkin and Maddux (1994), Düntsch (2005), and the references provided for particular calculi in Appendix A.

2.1. What is a Qualitative Spatial Calculus?

A qualitative calculus consists of a set of relations between objects from a certain domain and operations defined on these relations. Let us start with an easy example: the spatial version of the Point Algebra (PA) (Vilain et al., 1989). Imagine, we are being told about a boat race on a river by a friend on the phone[1]. We can model the river as an oriented line and the boats of the 5 participants A,B,C,D,E as points moving along the line (see Fig. 2.1). Thus, our domain (the set of spatial objects considered) is the set of all 1D points.

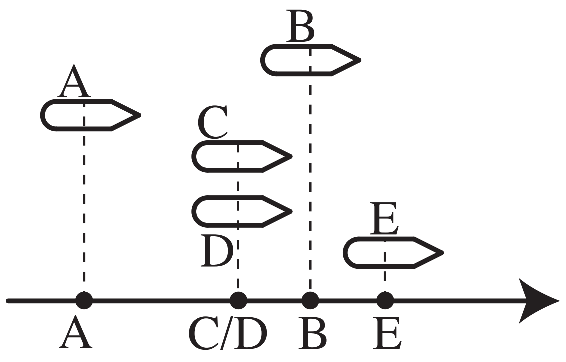


Figure 2.1.: A possible situation in a boat race which can be modeled by 1D points on an oriented line and be described by qualitative relations from the Point Algebra.

We now distinguish three relations between objects from our domain. A boat can be *ahead* of another boat, *behind* it, or on the *same* level. These relations can be used to

[1]This example has been borrowed from Ligozat (2005).

formulate knowledge about the current situation in the race. For instance, our friend tells us the following:

1. A is *behind* B
2. E is *ahead* of B
3. A is *behind* C
4. D is on the *same* level as C
5. A is *ahead* of D

From this information we are able to conclude that our friend must have made an error, probably confusing the names of the participants: We know that A is *behind* C (sentence 3) and D is *behind* A (conversion of sentence 5). From composing these two facts it follows that C and D cannot be on the *same* level which contradicts sentence 4.

On the other hand, only taking the first three sentences into account, we can conclude that E is also *ahead* of A by composing the facts A is *behind* B (sentence 1) and B is *behind* E (conversion of sentence 2). However, this information is not sufficient to derive the exact relation between C and E, as C can either be *ahead*, *behind* or on the *same* level as E.

The calculus, in this case the PA, defines a set of base relations (*ahead*, *behind*, and *same*) and provides the elementary reasoning steps in the form of operations defined over the base relations. In our small example, the applied operations were conversion, which given the operation between x and y returns the relation between y and x (thus the converse of *ahead* is *behind*), and composition which takes the relations holding between X & Y and Y & Z and returns the relation holding between X & Z (e.g. composition of *ahead* and *ahead* is *ahead*).

Often the result of operations like the composition operation is not a single base relation but the union of more than one. For instance, knowing that X is *ahead* of Y and Y is *behind* Z yields the union of *ahead*, *behind*, and *same*. Because of this, the set of relations considered in a spatial calculus is not just the set of base relations, but the set of all unions of base relations including the empty set and the union of all base relations (the universal relation). All operations of the calculus are then defined for all unions of base relations: For example, we can apply conversion to the information that X is either *ahead* or at the *same* level as Y to infer that Y is either *behind* or at the *same* level as X.

2.2. Constraint Networks, Consistency, and Consistent Scenarios

A spatial configuration of a finite set of objects from the domain as given by sentences 1–5 can be described as a *constraint network* as shown in Fig. 2.2. It consists of a variable for each object represented by the nodes of the network and edges labeled with relations from the considered calculus denoted as sets of base relations. For instance, sentence 1 is represented by the edge going from A to B labeled with {*behind*}. If no edge connects two nodes, this corresponds to an edge labeled with the universal relation U (the union of all base relations expressing complete ignorance), which is usually omitted.

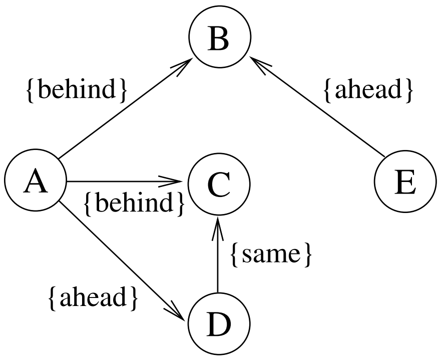


Figure 2.2.: The situation described by sentences 1–5 as a constraint network.

As we have seen in our example, the information given in a constraint network can be inconsistent. This means, no objects from the domain can be assigned to the variables so that all the constraints given by the spatial relations annotated to the edges are satisfied. If, on the other hand, such an assignment can be found, the constraint network is said to be *consistent* or *satisfiable* or *realizable*. Determining whether a constraint network is consistent is a fundamental problem of qualitative spatial reasoning. Special techniques for determining consistency based on the operations of the calculus (especially the composition operations) have been developed. However, it is important to note that the soundness of these methods depends on the properties of the calculus at hand and are often still subject of ongoing investigations. For more details on this issue, we refer to Renz and Ligozat (2005) and the literature on individual calculi listed in Appendix A.

A constraint network in which every constraint between two variables is a base relation is called *atomic* or a *scenario*. This means all spatial relations between two objects are completely determined with respect to the employed calculus and the remaining question is if the network is consistent or not. However, if a constraint network contains relations that are not base relations like in Fig. 2.3(a), we might also be interested in finding a scenario that is a *refinement* of the original network (meaning it has been derived by removing individual base relations from the sets annotated to the edges) and that is consistent. Fig. 2.3(b) shows such a *consistent scenario* for the network in Fig. 2.3(a). If such a consistent scenario can be found, we also know that the original network is consistent. Otherwise, we know it is inconsistent. Of course, it is possible that more than one consistent scenario exists for a given constraint network and we might be interested in finding only one or all of these. An alternative consistent scenario is depicted in Fig. 2.3(c).

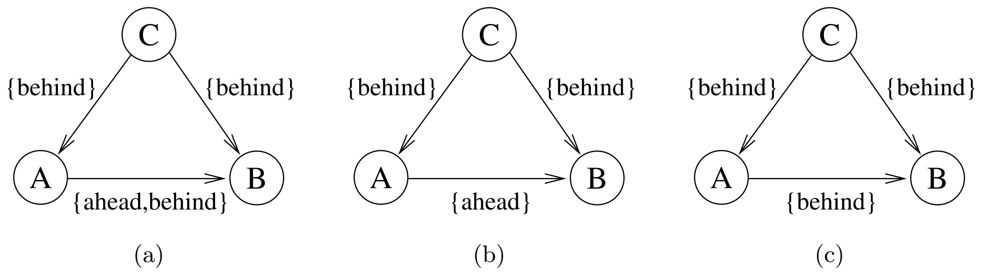


Figure 2.3.: A non-atomic constraint network (a) with possible consistent scenarios (b) and (c).

The problems of determining consistency and finding consistent scenarios are subsumed under the term *constraint-based reasoning* throughout this text.

2.3. Qualitative Constraint Calculi

After giving a rather intuitive introduction to qualitative spatial calculi, we want to give a more formal definition of a spatial calculus and especially the operations a calculus needs to define.

Definition 1 (qualitative calculus, base relations, arity)**.** A *qualitative calculus* $< \mathcal{B}, \mathcal{D}^n >$ defines a finite, non-empty set $\mathcal{B}$ of n-ary qualitative *base relations* over some domain $\mathcal{D}$, i.e. $\mathcal{B} \subseteq 2^{\mathcal{D}^n}$. We call n the arity of the calculus.

For the purpose of constraint reasoning one usually requires that the set of base relations partitions $\mathcal{D}^n$.

Definition 2 (JEPD)**.** A qualitative calculus $< \mathcal{B}, \mathcal{D}^n >$ is called *jointly exhaustive*, if the base relations cover $\mathcal{D}^n$, i.e. $\bigcup_{B \in \mathcal{B}} B = \mathcal{D}^n$. The calculus is called *pairwise disjoint* if no two base relations overlap, i.e. $\forall B, B' \in \mathcal{B} : B \cap B' = \emptyset$. Jointly exhaustive and pairwise disjoint calculi are commonly referred to as JEPD calculi.

In qualitative reasoning we consider a simple formal language that only allows relating (unqualified) objects by qualitative relations. Here, a infix notation is commonly used. For example, $A\,r\,B$ stands for $(A, B) \in r$. Uncertainty can be modeled by using unions of base relations, e.g., to express that some objects may either stand in some relation r or s can be expressed by the relation $r \cup s$, usually denoted $\{r, s\}$[2].

Definition 3 (general relations)**.** In a qualitative calculus $< \mathcal{B}, \mathcal{D}^n >$ a *general relation* $B = \{B_{i_1}, B_{i_2}, \ldots, B_{i_m}\}$, where $B_{i_1}, B_{i_2}, \ldots, B_{i_2} \in \mathcal{B}$ represents the relation $\bigcup_{j=1,2,\ldots,n} B_{i_j}$ obtained by uniting base relations. We will denote set of general relations obtained from a set of base relations $\mathcal{B}$ by $\mathcal{R}_{\mathcal{B}}$. Two special general relations are the empty relation $\varnothing$ and the universal relation $U = \bigcup_{B \in \mathcal{B}} B$.

In this context the JEPD property is important in two regards:

1. It offers a normal form of representing knowledge
2. The empty relation corresponds to unsatisfiability

The latter is particular important for reasoning: the empty relation cannot be part of any consistent scene description. Thus, deriving that no relation other than the empty relation can hold between two objects, means that the scene description is contradictory.

2.3.1. Operations

Let us now turn to the operations a calculus needs to define. There are three groups of operations:

- Set-theoretic operations on the level of general relations.
- Operations that represent a change of perspective
- Operations to integrate distinct propositions

2.3.2. Set-theoretic Operations

The set-theoretic operations on the level of general relations can be defined independent of the calculus at hand. The following table lists the standard operations and their corresponding SparQ operation names (R and S stand for general relations):

operation	**SparQ names**	**definition**
union	`union`	$R \cup S = \{ x \mid x \in R \vee x \in S \}$
intersection	`intersection, isec`	$R \cap S = \{ x \mid x \in R \wedge x \in S \}$
complement	`complement, cmpl`	$\overline{R} = U \setminus R = \{ x \mid x \in U \wedge x \notin R \}$

[2]The notation using sets like $\{r, s\}$ can be misleading as it syntactically identifies sets of relations with their unions. However, this notation is the established form.

2.3.3. Operations that Change Perspective

In the case of a binary (2-ary) calculus a change of perspective means that given we know the relation $A\,r\,B$, we can infer the relation r' such that $B\,r'\,A$.

Definition 4 (converse)**.** In a binary calculus $<\mathcal{B},\mathcal{D}^2>$ the unary *converse* operation $\breve{}$ is defined as follows:

$$r\breve{} = \{(A,B) \mid (A,B) \in \mathcal{D}^2 \wedge (B,A) \in r\}$$

Obviously, there are more ways to change perspectives in a general n-ary calculus for $n > 2$. Currently, only ternary (3-ary) calculi are also important to QSR. For ternary calculi 5 unary operations are commonly considered that are defined analogously to the converse in binary calculi. The following table gives a complete overview:

operation	**SparQ names**	**effect**
binary calculi:		
converse	`converse, cnv`	$ArB \quad \rightsquigarrow B r\breve{} A$
ternary calculi:		
inverse	`inv, inverse`	A,B r C $\rightsquigarrow$ B,A inv(r) C
shortcut	`sc, shortcut`	A,B r C $\rightsquigarrow$ A,C sc(r) B
shortcut inverse	`sci, shortcuti`	A,B r C $\rightsquigarrow$ C,A sci(r) B
homing	`hm, homing`	A,B r C $\rightsquigarrow$ B,C hm(r) A
homing inverse	`hmi, homingi`	A,B r C $\rightsquigarrow$ C,B hmi(r) A

These operations may further be generalized for n-ary calculi. Given the unavailability of calculi with arities higher than 3, SparQ currently implements the operations for binary and ternary calculi only.

2.3.4. Operations that Integrate

Admittingly the heading of this paragraph is misleading in that there is only one kind operation defined that integrates relations, the *composition* operations, most notably the *binary composition* $\circ$.

Definition 5 (binary composition in binary calculi)**.** In a binary calculus $<\mathcal{B},\mathcal{D}^2>$ the *composition* operation $\circ$ is defined as binary operator:

$$r \circ s := \{(A,C) \in \mathcal{D}^2 \mid \exists B \in \mathcal{D} \;:\; (A,B) \in r \wedge (B,C) \in s\}$$

Definition 6 (binary composition in ternary calculi)**.** In a ternary calculus $<\mathcal{B},\mathcal{D}^3>$ the *composition* operation $\circ$ is defined as binary operator:

$$r \circ s := \{(A,B,D) \in \mathcal{D}^3 \mid \exists C \in \mathcal{D} \;:\; (A,B,C) \in r \wedge (B,C,D) \in s\}$$

Other ways of composing two ternary relations can be expressed as a combination of the unary permutation operations and the composition (Scivos and Nebel, 2001) and thus do not have to be defined separately and are also not accessible individually in SparQ.

Besides the definition of ternary composition employed in SparQ and by many others, for example Freksa (1992a), ternary composition has also been defined as ternary operator, more specifically a n-ary operator in an n-ary calculus Condotta et al. (2006).

Definition 7 (n-ary composition)**.** In a n-ary calculus $<\mathcal{B}, \mathcal{D}^n>$ the *n-ary composition* operation $\bullet$ is defined as follows:

$$\begin{aligned}\bullet(r_1, r_2, \ldots, r_n) \;:=\; & \{(A_1 A_2 \ldots A_n) \in \mathcal{D}^n \mid \exists B \in \mathcal{D} : (A_1, A_2, \ldots, A_{n-1}, B) \in r_1 \wedge \\ & (A_1, A_2, \ldots, A_{n-2}, B, A_n) \in r_2 \wedge \ldots \wedge (B, A_2, A_3 \ldots, A_n) \in r_n\}\end{aligned}$$

2.3.5. Weak vs. Strong Operations

By definition of the operations it is not clear that for example the converse of a qualitative (base) relation itself is a (base) relation too. In fact, this is not the case for some calculi. It may even be the case that no finite set of relations exists that describes the results of the operations. Take for example the aforementioned point calculus Vilain et al. (1989) over the domain of natural numbers, i.e., $<\{\dot{<}, \dot{=}, \dot{>}\}, \mathbb{N}^2>$. Here, the composition $\dot{<} \circ \dot{<}$ stands for the relation "larger by at least 2" which cannot be described as a union of base relations provided by the point calculus. Extending the set of relation by the respective relation—lets call it $\dot{<}_1$—would only shift the problem since we would be facing a similar problem considering the composition $\dot{<}_1 \circ \dot{<}$.

The framework of qualitative reasoning requires us to restrict ourselves to a finite set of relations, the general relations. So when we cannot express the true relations obtained by applying some operation we must use some form of approximation. An upper approximation of the true operation is utilized to accomplish this, i.e., an approximation that fully contains the true relation. Such upper approximations of operations are called *weak operations* as opposed to the true or *strong operations*. Figure 2.4 gives an illustration.

In the case of the weak composition in a binary calculus $<\mathcal{B}, \mathcal{D}^2>$ it is defined as:

$$r \circ^\star s := \{B \in \mathcal{B} \mid B \cap (r \circ s) \neq \varnothing\}$$

Note that the use of an upper approximation of some operation still guarantees like in the case of strong operations that the empty relation can only be the result of contradicting information. However, the use of a weak operation can lead to situations in which a set of not agreeable statements is not detected as such.

Given that from a syntactical point of view the difference between weak and strong operations is not identifiable, SparQ does not differentiate strong and weak operations syntactically.

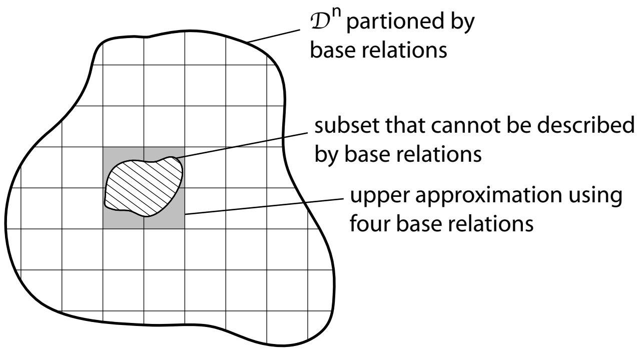

$\mathcal{D}^n$ partioned by base relations
subset that cannot be described by base relations
upper approximation using four base relations

Figure 2.4.: Illustration of a relation, i.e., a subset of $\mathcal{D}^n$, represented as an upper approximation

2.4. Checking Consistency

Determining consistency of a constraint network in which the constraints are given as qualitative spatial relations from a particular calculus, is a particular instance of a *constraint satisfaction problem* (CSP). Unfortunately, the domains of our variables are typically infinite (e.g. the set of all points in the plane) and thus backtracking over all the values of the domain cannot be used to determine consistency.

The techniques developed for relational constraint problems are instead based on weaker forms of consistency called *local consistencies* which can be tested or enforced based on the operations of the calculus and which are under particular conditions sufficient to decide consistency.

One important form of local consistency is *path-consistency* which (in binary CSPs) means that for every triple of variables each consistent evaluation of the first two variables can be extended to the third variable in such a way that all constraints are satisfied. In the best case, path-consistency decides consistency for a given calculus. This means, that if we can make the network path-consistent by possibly removing some base relations from the constraints without ending up with the empty relation, we know that the original network is consistent. If this cannot be achieved, the network has to be inconsistent. Unfortunately, it is usually not the case that path-consistency decides consistency.

However, sometimes path-consistency is sufficient to decide consistency at least for a subset $\mathcal{S}$ of the relations from $\mathcal{R}$, for instance the set of base relations. On the one hand, this means that whenever our constraint networks only contains labels which are base relations, we again can use path-consistency as a criterion to decide consistency. On the other hand, if the subset $\mathcal{S}$ exhaustively splits $\mathcal{R}$ (which means that every relation from $\mathcal{R}$ can be expressed as a union of relations from $\mathcal{S}$), this at least allows to formulate a backtracking algorithm to determine consistency by recursively splitting the constraints and using path-consistency as a decision procedure for the resulting CSPs with constraints from $\mathcal{S}$ (Ladkin and Reinefeld, 1992).

To enforce path-consistency, syntactic procedures called *algebraic closure algorithms* have been developed that are based on the operations of the calculus (the composition operation in particular) and work in $O(n^3)$ time for binary calculi and $O(n^4)$ for ternary calculi where n is the number of variables. But again, we have to note that these syntactic procedures do not necessarily yield the correct results with respect to path-consistency as defined above. Whether algebraic closure coincides with path-consistency needs be investigated for each calculus individually and we again refer to the literature listed in the individual calculus descriptions in Appendix A.

3. Using SparQ

SparQ consists of a set of modules that logically structure the different services provided, which will be explained below. The general architecture is visualized in Fig. 3.1. The dashed parts are extensions planned for the future (see section 4.2).

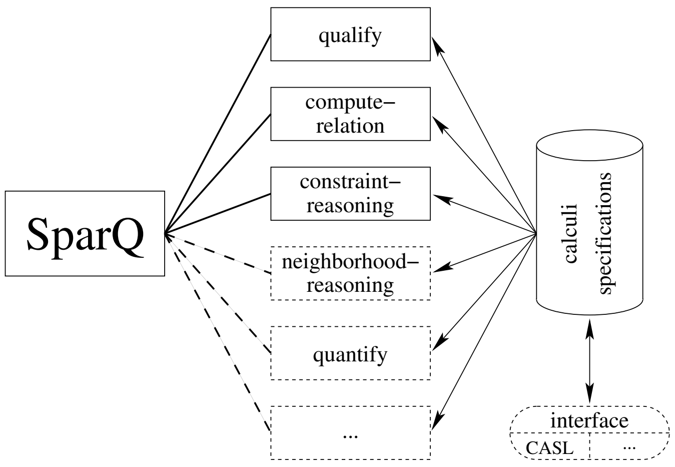


Figure 3.1.: Module architecture of the SparQ toolbox.

The general syntax for using SparQ is

```
$ ./sparq ¡module¿ ¡calculus¿ ¡module specific parameters¿
```

where `module` designates the particular module to use, `calculus` refers to the qualitative calculus to use, and the remainder of the command line give command specific arguments which will be explained in the following. SparQ can also be used in interactive mode (see section 3.11) — the general syntax is the same though.
Example:

```
$ ./sparq compute-relation rcc-8 composition dc ntpp
```

computes the composition of the rcc-8 relations DC and $NTPP$. SparQ prints the result `(dc ec po tpp ntpp)` to the shell which stands for $\{DC, EC, PO, TPP, NTPP\}$.
SparQ provides the following modules:

qualify — transforms a quantitative geometric description of a spatial configuration into a qualitative description based on one of the supported spatial calculi

quantify — complement to qualify, computes an exemplary geometric model

compute-relation — applies the operations defined in the calculi specifications (intersection, union, complement, converse, composition, etc.) to a set of spatial relations

constraint-reasoning — performs computations on constraint networks

neighborhood-reasoning — performs computations on constraint networks based on conceptual neighborhoods

algebraic-reasoning — geometric reasoning using algebraic geometry

analyze-calculus — relation-algebraic analysis of calculus structures

We will take a closer look at each of these three modules in the next sections.

3.1. Command Line Options

Implemented switches:

`-v` verbose mode primarily used for debugging purposes

`-i, --interactive` interactive mode (see section 3.11)

`-p <port>, --port` in interactive mode listen for connection on TCP/IP port rather using input from the shell

3.2. General Syntax

SparQ is case-insensitive, so the notation *leftOf* and *Leftof* denote the same identifier. When printing, SparQ uses small letters for relations and capital letters for objects. There are some characters that must not be used in specifiers for either relations or objects, in particular parentheses (`(`, `)`), punctuation (`.`, `,`, `;`, `:`), and `#` are not allowed. Nearly all other character sequences may be used—if in doubt, refer to the ANSI Common Lisp standard on symbols or just try out.

3.2.1. Calculi Identifier

SparQ commands require specifying a calculus. For each calculus implemented in SparQ an identifier has been defined (see the appendix for details on the calculi). Alternatively, calculi may be specified by giving the path name of the calculus definition file.

Calculi may have specific parameters, for example the granularity parameter in $\mathcal{OPRA}_m$. These parameters are appended with a '`-`' after the calculus' base identifier. `opra-3` for example refers to $\mathcal{OPRA}_3$.

calculus identifier(s)	**calculus**	**section**	**page**
`allen, aia, ia`	Allen's interval algebra (Allen, 1983)	A.1	54
`block-algebra, ba`	2D block-algebra (Güsgen, 1989)	A.2	55
`cardir`	Cardinal direction calculus (Ligozat, 1998)	A.3	56
`depcalc, dep`	Dependency calculus (Ragni and Scivos, 2005)	A.5	59
`dipole-coarse, dra-24`	Dipole calculus (Moratz et al., 2000)	A.11	67
`double-cross, dcc`	Double cross calculus (Freksa, 1992a) using the original tuple naming scheme	A.7	61
`alternative-double-cross, adcc`	Double cross calculus (Freksa, 1992a) using the alternative single number naming scheme	A.7	61
`flipflop, ffc, ff`	FlipFlop calculus (Ligozat, 1993)	A.8	63
`geomori, ori, align`	Geometric Orientation calculus	A.10	66
`point-calculus, pc, point-algebra, pa`	Point algebra (Vilain et al., 1989)	A.13	71
`rcc-5`	Region connection calculus (RCC-5) (Randell et al., 1992)	A.4	58
`rcc-8`	Region connection calculus (RCC-8) (Randell et al., 1992)	A.4	57
`reldistcalculus`	Exemplary calculus from this manual	3.13.1	36
`single-cross, scc`	Single cross calculus (Freksa, 1992a)	A.6	60
`opra-`	Oriented point reasoning algebra ($\mathcal{OPRA}_m$)(Moratz, 2006)	A.12	69

`qtc-b11`	Qualitative trajectory calculus in 1D with distance (van de Weghe, 2004)	A.14	72
`qtc-b12`	Qualitative trajectory calculus in 1D with velocity (van de Weghe, 2004)	A.14	73
`qtc-b21`	Qualitative trajectory calculus in 2D with distance (van de Weghe, 2004)	A.14	74
`qtc-b22`	Qualitative trajectory calculus in 2D with distance and velocity (van de Weghe, 2004)	A.14	74
`qtc-c21`	Qualitative trajectory calculus in 2D with distance and side (van de Weghe, 2004)	A.14	75
`qtc-c22`	Qualitative trajectory calculus in 2D with distance, side, and velocity (van de Weghe, 2004)	A.14	76

3.2.2. Denoting Relations

Relations are denoted using their name, as a disjunction using `(`, `)`, or by wild cards `*` and `?`.
Example using the RCC-8 calculus:

- `nttp`, `Ntpp` both stand for the relation `nttp`
- `(nttp eq po)` stands for the relation `nttp` ∪ `eq` ∪ `po`
- `p?` stands for all relations with a two-letter name starting with 'p', i.e. `po` ∪ `pp` in case of the RCC-8 calculus
- `*` stands for the universal relation

Please be aware, that if you pass arguments to SparQ via the command line, the shell will perform some replacements, in particular if you are using parentheses or `*`. You need to wrap quotes around your arguments, i.e. use `"(po eq)"` instead of `(po eq)`, to avoid unwanted replacements.

3.2.3. Denoting Configurations

Configurations are static scene descriptions that interrelate named objects using qualitative relations of a particular calculus. Named objects are related by enclosing object identifiers and relation by parentheses, e.g. `(A po B)`, configurations are specified using a list (enclosed in parentheses, no comma-separation), e.g. `((A (po eq) B) (B eq C))`.

3.3. Qualify

The purpose of the qualify module is to turn a quantitative geometric scene description into a qualitative scene description composed of base relations from a particular calculus. The calculus is specified via the calculus identifier that is passed with the call to SparQ. Qualification is required for applications in which we want to perform qualitative computations over objects whose geometric parameters are known.

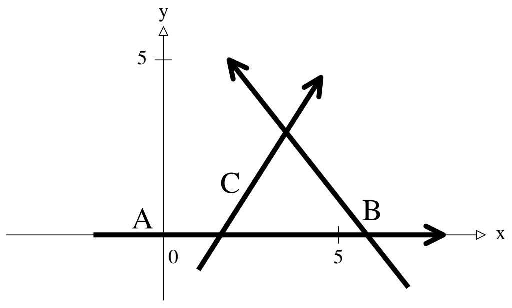


Figure 3.2.: An example configuration of three dipoles.

The qualify module reads a quantitative scene description and generates a qualitative description. A quantitative scene description is a space-separated list of base object descriptions enclosed in parentheses. Each base object description is a tuple consisting of an object identifier and object parameters that depend on the type of the object. For instance, let us say we are working with dipoles which are oriented line segments. The object description of a dipole is of the form '(name x_s y_s x_e y_e)', where name is the identifier of this particular dipole object and the rest are the coordinates of start and end point of the dipole. Let us consider the example in Fig. 3.2 which shows three dipoles A, B, and C. The quantitative scene description for this situation would be:

```
( (A -2 0 8 0) (B 7 -2 2 5) (C 1 -1 4.5 4.5) )
```

Note: Coordinates may be specified as either integers (2; -3; ...), floats (3.2234; -1e-07), or rational numbers (13/7; -4/2). Using rational numbers can help avoiding effects of rounding errors. Depending on the basis entity (i.e., the domain of the qualitative calculus), different values need to be supplied. This table gives an overview:

basis entity	**format**	**semantics**
1d-point	`(id x)`	Real-valued position of the point
interval	`(id s e)`	Real-valued (closed) interval
2d-point	`(id x y)`	Real-valued coordinates
2d-oriented-point	`(id x y dx dy)`	Real-valued coordinates and direction of the orientation
dipole	`(id xs ys xe ye)`	Directed line segment connecting the 2d points (xs ys) and (xe ye)
2d-box	`(id x1 y1 x2 y2)`	Axis-aligned rectangle with bottom left point (x1,y1) and top right point (x2,y2)
polygon	`(id x1 y1 ... xn yn)`	Polygon with vertices (x1,y1), (x2,y2), ... (xn,yn)

The qualify module has one module specific parameter *mode* that needs to be specified. It controls which relations are included into the qualitative scene description; there are two settings:

`all` The relation between every object and every other object will be included. In the case of a binary calculus SparQ prints out a configuration containing n^2 relations, if n is the total number of objects in the scene description.

`first2all` If mode is set to `first2all` only the relations between the first and all other objects are computed in the binary case or between the first two objects and all other objects in the ternary case.

The resulting qualitative scene description is a space-separated list of relation tuples enclosed in parentheses. A relation tuple consists of an object identifier followed by a relation name and another object identifier, meaning that the first object stands in this particular relation with the second object. The command to produce the qualitative scene description followed by the result is[1]:

```
$ ./sparq qualify dra-24 all "((A -2 0 8 0) (B 7 -2 2 5) (C 1 -1 4.5 4.5))"
¿ ((A rllr B) (A rllr C) (B lrrl C))
```

If we had chosen ‘first2all’ as mode parameter the relation between B and C would not have been included in the qualitative scene description.

3.4. Quantify

The `quantify` command complements the `qualify` in the sense that it computes an exemplary geometric scene from a constraint network.

[1]In all the examples, input lines start with ‘$’. Output of SparQ is marked with ‘>’.

```
$ ./sparq quantify allen "((a b b) (a fi c))"
¿ ((C 1 2) (B 5 6) (A 3 4))
```

3.5. Compute-relation

The compute-relation module allows to compute with the operations defined in the calculus specification. The module specific parameters are the operation that should be conducted and one or more input relations depending on the arity of the operation. Let us say we want to compute the converse of the *llrl* dipole relation. The corresponding call to SparQ and the result are:

```
$ ./sparq compute-relation dra-24 converse llrl
¿ (rlll)
```

The result is always a list of relations as operations often yield a disjunction of base relations. In this case, however, the disjunction only contains a single relation. The composition of two relations requires one more relation as parameter because it is a binary operation, e.g.:

```
$ ./sparq compute-relation dra-24 composition llrr rllr
¿ (lrrr llrr rlrr slsr lllr rllr rlll ells llll lrll)
```

Here the result is a disjunction of 10 base relations. It is also possible to have disjunctions of base relations as input parameters. For instance, the following call computes the intersection of two disjunctions:

```
$ ./sparq compute-relation dra-24 intersection "(rrrr rrll rllr)" "(llll rrll)"
¿ (rrll)
```

Note that you need to put relation specifications in quotes when giving them as arguments on the command line. SparQ can also process nested computations if individual parts are enclosed in parentheses, e.g., computing the complement of the converse of a relation can be done as follows:

```
$ ./sparq compute-relation dra-24 "(complement (converse (rr??)))"
¿ (ells errs eses lere llll lllr llrl lrll lrrl lsel rele rlll rllr rrll rrlr rrrl rser sese slsr srsl)
```

The following operations are currently implemented (see section 2.3.1):

spatial:

`composition, comp`	$(r, s) \mapsto r \circ s$
`ternary-composition, tcomp`(3)	$(r, s, t) \mapsto \bullet(r, s, t)$
`n-ary-composition, ncomp`	$(r_1, \dots, r_n) \mapsto \bullet(r_1, \dots, r_n)$
`converse, cnv`(2)	$r \mapsto r^{\smile}$
`homing, hm`(3)	$r \mapsto hm(r)$
`homingi, hmi`(3)	$r \mapsto inv(hm(r))$
`inverse, inv`(3)	$r \mapsto inv(r)$
`shortcut, sc`(3)	$r \mapsto sc(r)$
`shortcuti, sci`(3)	$r \mapsto inv(sc(r))$
calculi-theoretic:	
`closure`	$r_1 r_2 \dots r_n \mapsto Cl(r_1, r_2, \dots, r_n)$ Cl denotes the minimal set of relations that is closed under composition, converse, and intersection
`base-closure`	$Cl(br_1, br_2, \dots, br_n)$ Computes closure of the set of base relations
`test-properties`	Tests whether a calculus meets relation algebra axioms
set-theoretic:	
`complement, cmpl`	$r \mapsto r^C$
`minus`	$rs \mapsto r \backslash s$
`union`	$rs \mapsto r \cup s$
`intersection, isec`	$rs \mapsto r \cap s$

Commands marked (2)are valid for binary calculi only, commands marked (3)are valid for ternary constraints only.

Note: The commands `closure` and `base-closure` are currently defined exclusively for binary calculi. The closure of a set of relations may be very large and computation may, consequently, take very long.

3.6. Constraint-reasoning

The constraint-reasoning module reads a description of a constraint network—which is a qualitative scene description that may include disjunctions and may be inconsistent and/or underspecified—and performs an operation on it, e.g., a particular kind of consistency check:

```
constraint-reasoning <calculus> <operation> <constraint-network> ...
```

Currently, operations are implemented for checking consistency of a constraint network and propagating constraints (‘check-consistency’, ‘algebraic-closure’, ‘ternary-closure’, ‘scenario-consistency’), for manipulating constraint networks (‘refine’, ‘extend’, ‘update’) and for correspondence between two networks (‘match’, ‘best-match’).

3.6.1. Constraint-based reasoning

Action ‘algebraic-closure’ causes the module to enforce path-consistency on the constraint network using a variant of Mackworth’s AC-3 algorithm (Mackworth, 1977). As a result the constraint network obtained is returned. If during constraint propagation an inconsistency is discovered, the inconsistency is reported and not netwrok is returned. In case of ternary calculus the canonical extension of the AC-3 algorithm as described in Dylla and Moratz (2004) is used. For ternary calculi there also exist the option to use the more natural ternary composition operation (if defined for the respective calculus) instead of the binary composition operation. Enforcing path-consistency using ternary composition operation is invoked using ‘ternary-closure’.

Examples

We could for instance check if the scene description generated by the qualify module in Section 3.3 is algebraically closed—which of course it is. To make it slightly more interesting, we add the base relation *ells* to the constraint between A and C resulting in a constraint network that is not algebraically closed:

```
$ ./sparq constraint-reasoning dra-24 algebraic-closure
"((A rllr B) (A (ells rllr) C) (B lrrl C))"
¿ Modified network.
¿ ( (B (lrrl) C) (A (rllr) C) (A (rllr) B) )
```

The result is an algebraically closed constraint network in which *ells* has been removed. The output ‘Modified network’ indicates that the original network was not algebraically closed and had to be changed. Otherwise, the result would have started with ‘Unmodified network’. In the next example we remove the relation *rllr* from the disjunction between A and C. This results in a constraint network for which algebraic closure detects an inconsistency which means it is not globally consistent.

```
$ ./sparq constraint-reasoning dra-24 algebraic-closure
"((A rllr B) (A ells C) (B lrrl C))"
¿ Not consistent.
¿ ((B (lrrl) C) (A () C) (A (rllr) B))
```

SparQ correctly determines that the network is inconsistent and returns the constraint network in the state in which the inconsistency showed up (indicated by the empty relation () between A and C).

In a last example for algebraic-closure we use the ternary double cross calculus:

```
$ ./sparq constraint-reasoning dcc algebraic-closure
"((A B (7_3 6_3) C) (B C (7_3 6_3 5_3) D) (A B (3_6 3_7) D))"
¿ Not consistent.
¿ ((A B (3_6 3_7) D)(A B () C)(B C (5_3 6_3 7_3) D)(D C (0_4 1_5 2_5 3_5 3_6 3_7 4_4
5_3 6_3 7_3 b_4) A))
```

If 'scenario-consistency' is provided as argument, the constraint-reasoning module checks if an algebraically closed scenario exists for the given network. It uses a backtracking algorithm to generate all possible scenarios and checks them via algebraic closure as described above. Knowledge about tractable sets, if defined for the calculus at hand, are exploited too. A second module specific parameter determines what is returned as the result of the search:

return — This parameter determines what is returned in case of a constraint network for which path-consistent scenarios can be found. It can take the values 'first' which returns the first path-consistent scenario, 'all' which returns all path-consistent scenarios, and 'interactive' which returns one solution and queries whether the search shall be continued. Finally, 'check' instructs SparQ to only check existence of an algebraically closed scenario which, in some cases, can be much faster than actually computing a solution.

While computing scenario-consistency, algebraic closure is also used as a forward-checking method during the search to make it more efficient. For certain calculi, the existence of an algebraically closed scenario implies consistency. However, this again has to be investigated for each calculus (cmp. Section 2.4).

In the following example, we use 'first' as additional parameter so that only the first solution found is returned:

```
$ ./sparq constraint-reasoning dra-24 scenario-consistency first
"((A rele C) (A ells B) (C errs B) (D srsl C) (A rser D) (D rrrl B))"
¿ ((B (rlrr) D) (C (slsr) D) (C (errs) B) (A (rser) D) (A (ells) B) (A (rele) C))
```

In case of an inconsistent constraint network, SparQ returns 'Not consistent.' as in the following example:

```
$ ./sparq constraint-reasoning dra-24 scenario-consistency first
"((A rele C) (A ells B) (C errs B) (D srsl C) (A rser D) (D rllr B))"
¿ Not consistent.
```

For calculi which define tractable subsets the consistent scenarios may be printed in a condensed form by giving disjunctions of base relations such that all combinations determine a consistent scenario, e.g.

```
$ ./sparq constraint-reasoning rcc8 scenario-consistency all
"((a po b) (b ntpp c))"
¿ ((B (ntpp) C)(A (ntpp po tpp) C)(A (po) B))
¿ 3 scenarios found, no further scenarios exist.
```

In this example, any of the relations *ntpp*, *po*, *tpp* holding between *A* and *C* describes a consistent scenario, thus there are three scenarios in total.

3.6.2. Manipulating constraint networks

Constraint network manipulation is realized by operations which compute the conjunction of two constraint networks. With qualitative calculi it is natural to assume that a constraint network that does not define a constraint between two objects (or which does not even involve the objects) implicitly declares the universal relation as constraint. This allows manipulation with three simple operations.

The action 'refine' returns the conjunction of two constraint networks. Analogously, 'extend' returns the disjunction:

```
$ ./sparq constraint-reasoning dra-24 refine "((A (rele errs) B))" "((A errs B))"
¿ ((A errs B))
```

```
$ ./sparq constraint-reasoning dra-24 extend "((A rele B))" "((A errs B))"
¿ ((A (rele errs) B))
```

Finally, operation 'update' allows constraints to be overwritten:

```
$ ./sparq constraint-reasoning dra-24 update "((A rele B) (B eses C))" "((B llrr C))"
¿ ((B (LLRR) C) (A (RELE) B))
```

Overview of commands operating on constraints:

consistency checking:	
`check-consistency`	decides consistency using calculus-specific consistency check
`algebraic-closure`, `a-closure`, `path-consistency`	Enforces path-consistency — since this is a purely syntactical operation on the level of qualitative relations the term "algebraic closure" is more adequate. However, since "path-consistency" is widely used in this meaning, this name is supported too.

`scenario-consistency`	Computes algebraically closed networks containing base-relations only
`ternary-closure`	Computes algebraically closed networks using ternary composition with ternary calculi
`match`	Computes possible correspondence (isomorphy and joint consistence)
manipulating constraint networks:	
`refine`	Merges two networks by intersecting corresponding constraints
`extend`	Merges two networks by uniting corresponding constraints
`update`	Merges two networks by overwriting corresponding constraints

3.7. Algebraic reasoning

SparQ includes a module for reasoning about real-valued domain using techniques of algebraic geometry. Spatial reasoning problems get posed as algebraic problems of solving systems of equations. The techniques implemented in SparQ are based on Gröbner bases (see e.g., Cox et al. (1998)). The main service offered by the algebraic reasoning module is to provide a consistency checking mechanism for constraint networsk that is based on the relation semantics only. This means, algebraic reasoning can be used to compute or to verify operation tables such as e.g., used for specifying the composition operation. Furthermore, algebraic reasoning can be used to analyze certain calculi properties.

3.7.1. Consistency checking

The algebraic reasoning module provided consistency analysis of constraint networks using the same syntax as constraint based consistency analysis (see Sec. 3.6). In contrast to constraint based reasoning, algebraic reasoning only makes use of an algebraic relation specification.

```
$ ./sparq a-reasoning ¡calculus¿ consistency ¡network¿
```

Networks are denoted as defined in the context of the constraint based reasoning module. Possible results:

`SATISFIABLE` The network is proven to be satisfiable.

`NOT SATISFIABLE.` The network is proven to be unsatisfiable.

`CANNOT DECIDE.` Neither one of the above proofs succeeded.

Currently, the implementation of the algebraic reasoning module aims at proving inconsistency of constraint networks, thus the answer “satisfiable” appears only rarely. Making algebraic reasoning a more powerful tool is ongoing research.
Example:

```
$ sparq -v a-reasoning ff consistency ”((A B l C) (B C l D) (A B f D))”
```

By turning on the verbose mode (command line option “ `-v`”), the proof generated by SparQ is printed to the console. Algebraic reasoning as implemented in SparQ is based on analyzing Gröbner bases. Computing Gröbner bases is a computationally very expensive process and a time limit is implemented into SparQ to prevent hangs. Currently, the time limit is hard wired to approx. 17 minutes. To abort an ongoing computation, use Control-C to stop SparQ. Currently, the time limit is a compile-time option, hence, to increase time limit, your need to change the source parameter declaration `*timeout*` in the beginning of `Source/polysolver.lisp`. Note that the unit size is ms here.

Due to the computational demands of algebraic reasoning we advice users to utilize this method only for analyzing small constraint networks. The most useful application of this module is for computing calculus operations.

Example: computing calculus operations

Suppose, you are about to introduce a new qualitative calculus and have designed the set of base relations. In order to do any constraint based reasoning with this new calculus, permutation operations (converse, inverse, shortcut, etc.) and the composition operation need to be defined. Determining composition operation tables is a demanding process: all 3-consistent constraint networks involving $k + 1$ base objects need to be computed whereby k denotes the arity of the calculus. In such situations algebraic reasoning can be effectively applied to rule out inconsistent networks. Consider again the example from above, analyzing the constraint network

```
((A B l C) (B C l D) (A B f D))
```

in context of the FlipFlop base relations (see A.8) SparQ will come up with the reply that the given network is not consistent. Therefore, the relation “f” (front) is not a possible result of the composition of “l” and “l” (left). To automate operation analysis an additional command is implemented.

3.7.2. Operation analysis

This command applies algebraic reasoning to check operations as explained in the previous section.

```
$ sparq a-reasoning ¡calculus¿ analyze-operation ¡operation¿
```

The command iterates over all base relations to analyze the given operation (e.g., composition, shortcut, etc.) and prints out a table summarizing the results. The table for an unary operation looks like this:

```
progress. r        analysis
......... r1        Verified.
......... r2        CANNOT INCLUDE: (r1)
......... r3        could not prove non-membership of : (r1 r2)
......... r4        could not prove membership of : (r2)
......... r5        ALSO INCLUDES: (r1)
```

The dots in the progress column are printed to show the progress of the verification, the column labeled "r" lists the relation examined. The column analysis lists the result which is one of the following:

`Verified` The entry of the operation table as given in the calculus specification has been verified.

`CANNOT INCLUDE (r1 r2 ...)` The reported base relations are listed in the operation table but they cannot be the result of applying the operation. This is an error in the operation table - remove the conflicting relations.

`ALSO INCLUDES: (r1 r2 ...)` The reported base relations have been proved to be missing in the operation table - add them.

`could not prove membership of: (r1 r2 ...)` The reported base relations are listed in the operation table, but no proof could be generated to show that these relations can be the result of applying the operation to r. This does not indicate an error in the operation table.

`could not prove non-membership of: (r1 r2 ...)` The reported base relations are not listed in the operation table, but no proof could be generated to show that these relations cannot be the result of applying the operation to r. This does not indicate an error in the operation table.

3.7.3. Qualification

When an algebraic calculus specification is provided, it can be used in qualification too. This means, a scenario can be qualified without supplied an designated qualification function.

```
$ ./sparq a-reasoning ¡calculus¿ qualify ¡option¿ ¡scenario¿
```

Syntax and options are the same as for the qualification module described in Sec. 3.3.

3.8. Analyzing Calculi

SparQ involves a set of commands that aim to aid researchers to analyze qualitative calculi. These commands are summarized in the module `analyze-calculus` and are named as follows:

`test-algebra` Using this command one can check whether a calculus definition meets the axioms that define a relation algebra in the sense of Tarski. The command is also helpful to identify potential errors in operation tables of newly added calculi—if SparQ only reports very few violations of an axiom, these violations may be caused my en erroneous entry in an operation table

`test-property` This command allows specific properties or axioms to be tested for a specific calculus. Essentially, SparQ checks a single statement as usable with the `compute-relation`-command with quantified variables. We illustrate this by an example, checking whether `rrc5` has an idempotent converse operation:

```
$ ./sparq analyze-calculus rcc5 test-property ”(forall r baserel) (equals r (converse (converse
r)))”
```

Here, `r` is a variable ranging over all base relations of the RCC-5 calculus. Note that the quotes are necessary to prevent a Unix shell from interpreting parenthesis expressions. It is possible to use quantifiers `exists` and `forall`, ranging over either base relations (`baserel`) or all general relations (`rel`). Special operators for comparing relations that are likely used with this command are `equal, covers, coverseq` (either `equal` or `covers` hold), `coveredBy, coveredByEq`.

`algebra-stats` Operation tables of calculi often differ in their information content, e.g., how often a universal relation occurs as the result of applying a composition operation. This command determines the information content of applying k steps of composition, an additional parameter determines the time in seconds to spend on analyzing the calculus.

3.9. Neighborhood-Based Reasoning

The aim of this module is to provide tools for reasoning based on the notion of conceptual neighborhood (Freksa, 1992b) that can be used for addressing spatial change over time as well as constraint relaxation.

TBD

	neighborhood-reasoning ¡CALCULUS¿ ¡NEIGHBORHOOD¿ ...
`similarity OP <CSP> <CSP>`	computes similarity of two constraint-networks, using OP as distance accumulation operation
`neighbors <REL>`	gives the conceptual neighbors of a relation
`merge OP OP <CSP> <CSP`	merges two constraint networks, using OP as distance accumulation operation
`relax <RELATION>`	coarsens relation by including all conceptual neighbors of a relation
`neighborhood-distance <REL> <REL>`	distance in the neighborhood graph

Table 3.4.: Summary of commands for neighborhood-based reasoning

3.10. Interfaces

Currently, SparQ provides only means for exporting calculi specification to formats used in other reasoners, the syntax is as follows:

```
$ ./sparq export ¡calculus¿ ¡type¿ ¡filename¿
```

where type is one of

`qat` QAT is a toolkit for qualitative spatial and temporal reasoning written in Java (Condotta et al., 2006) that uses an xml format for calculi specifications. Currently, only binary calculi can be exported into this format. For more information see: `http://www.cril.univ-artois.fr/~saade/QAT`

`gqr` GQR is a generic constraint reasoner for binary calculi, i.e., it provides an alternative implementation of SparQ's constraint-reasoning module but limited to binary calculi. For more information see: `https://sfbtr8.informatik.uni-freiburg.de/R4LogoSpace/Resources/GQR`

By invoking the export command, SparQ creates a file `<filename>` (two in the case of GQR export) in SparQ's main directory.

3.11. Interactive Mode

SparQ can be started in interactive mode to process commands repeatedly. This greatly reduces overhead of loading the program or calculi definitions. Interactive mode is activated by the command line option `-i` (or `--interactive`). The command syntax is identical to the standard mode of operation, there are some additional commands though:

command	**description**
`quit`	exits SparQ
`help`	prints short help message
`load-calculus CALC`	loads a specified calculus into memory
`*`	used as calculus specifier in commands, `*` stands for the calculus recently loaded into memory. This avoids overhead of reloading a calculus
`let VAR = EXP`	binds expression/result of command to variable
`let (VAR1 ... VARN) = EXP`	bind multiple variables to commands providing multiple return values
`print $VAR`	print value bound to variable

3.11.1. Variables

In the interactive mode, SparQ provides simple variables to reduce typing effort – or network traffic, if SparQ is interfaced using sockets. Variables are set using the `let` command; `print` allows values to be printed out. As some commands return multiple values, `let` allows multiple variables to be bound at once, the special name `_` (underscore sign) can be used to ignore a value. When used in commands, variables are marked by a leading dollar sign. Note that names are case insensitive. Here is an example on using variables:

```
SparQ¿ let myScene = qualify pc all ((a 1) (b 2) (c 1/4))
¿ ((A ¡ B) (A ¿ C) (B ¿ C))
SparQ¿ let myNeWScene = constraint-reasoning pc extend $myscene ((a ¡ d))
¿ ((A (¡) B) (A (¿) C) (A (¡) D) (B (¿) C))
SparQ¿ let (_ closedNet) = constraint-reasoning pc a-closure $myNewScene
Modified network.                                        ;; ignore first return value
((B (¿) C)(D (¿) C)(D (¡ = ¿) B)(A (¿) C)(A (¡) B)(A (¡) D))
SparQ¿ constraint-reasoning pc scenario-consistency first $myNewScen
((D (¿) A)(C (¡) A)(C (¡) D)(B (¿) A)(B (¿) D)(B (¿) C))
SparQ¿ print $myScene
¿ ((A ¡ B) (A ¿ C) (B ¿ C))
```

3.12. Including SparQ Into own Applications

In interactive mode, SparQ can be used as server that can easily be integrated into own applications. We have chosen a client/server approach as it allows for straightforward integration independent of the programming language used for implementing the application.

When run in server mode, SparQ uses a TCP/IP connection as input/output and interacts with the client via simple plain-text line-based communication. This means the client sends commands just like if using SparQ in interactive mode, and can then read the results from the TCP/IP stream.

SparQ is started in server mode by providing the command line option `--interactive` (`-i`), optionally followed by `--port` (`-p`) to specify the port.

```
$ ./sparq –interactive –port 4443
```

If no port is given, SparQ interacts with standard-input and standard-output, i.e., it can be used interactively from the shell.

```
import socket,sys,time
CRLF = "\r\n"    # Define line endings

def readline():
  "Read a line from the server.  Strip trailing CR and/or LF."
  input = sockfile.readline()
  if not input:
    raise EOFError
  if input[-2:] == CRLF:  # strip line endings
    input = input[:-2]
  elif input[-1:] in CRLF:
    input = input[:-1]
  if len(input) == 0:
    return readline()
  if input[0] == ";":     # ignore comments
    return readline()
  else:
    return input

def sendline(line):  # send a line to SparQ
  sock.send(line + CRLF) # unbuffered write

def removePrompt( line ):  # remove "sparq>" prompt
    return line[line.find('>')+7:]

# create a socket and connect
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('localhost', 4443))
sockfile = sock.makefile('rw')

# qualify a geometrical scenario with DRA-24
sendline('qualify dra-24 first2all ((A 4 6 9 0.5) (B -5 5 0 2) (C -4 5 ↘
    →6 0))')
scene = removePrompt( readline() )
```

```
print scene

# add an additional network ((B 4_1 C))
sendline("constraint-reasoning dra-24 refine " + scene + ' ((B eses C))')
scene2 = removePrompt( readline(  ) )
print scene2

# check the new scenario for consistency
sendline('constraint-reasoning dra-24 algebraic-closure ' + scene2)
consistent = removePrompt( readline() )
print consistent
if consistent != "Not consistent.": # ...read resulting CSP
    net = readline()
    print net

sendline("quit")
sock.close()
```

An example of client/server communication with SparQ is given in Listing ?? which shows a small Python script that opens a connection to the server and performs some simple computations (qualification, adding another relation, enforcing algebraic closure). It produces the following output:

```
> ((A rrll B) (A rrll C))
¿ ((A rrll B) (A rrll C) (B eses C))
¿ Not consistent.
¿ ((B (eses) C) (A (rrll) B) (A () B))
```

Special care may be given to line endings, depending on the operating system. The example code defines a function to strip line endings from the result lines. Furthermore, comment lines beginning with a semicolon have to be ignored. Also note that every first return line after a command comes with a preceeding `sparq` prompt, so the first six characters are stripped of the result. This behavior is due to compatibility reasons and will definitely change in later versions of SparQ (downward compatibility, however, will be granted).

3.13. Adding new Calculi

For most calculi it should be rather easy to include them into SparQ. Adding a new calculus consists of two steps:

1. Provide a calculus specification and store it in SparQ's subdirectory `Calculi`
2. Register your calculus in the calculus registry `Calculi/calculus-registry.lisp`

SparQ also allows loading calculi from an arbitrary file (by supplying the path name as calculus argument), so calculi are not required to be registered permanently. Example:

```
$ ./sparq compute-relation /path/to/my/calculus.lisp converse someRel
```

3.13.1. Calculus Specification

Let us start by giving an example for a simple calculus for reasoning about distances between three point objects that distinguishes only the three relations 'closer', 'farther', and 'same'. Following the intuition, $A\,B$ `closer` C holds if and only if A and B are closer to one another than A is to C. Farther and same are defined analogously. Listing 3.1 shows the specification which is done in Lisp-like syntax.

The arity of the calculus, the base relations, the identity relation and the different operations have to be specified, using lists enclosed in parentheses (e.g. when an operation returns a disjunction of base relations). In this example, the shortcut operation applied to 'same' yields 'same' and composing 'closer' and 'same' results in the universal relation written as the disjunction of all base relations. It is not required to specify the inverse shortcut and inverse homing operations (cmp. Section 2.3.1) as these can be computed by applying the other operations (e.g., inverse of shortcut yields inverse shortcut). It is, principally, possible to leave more operations unspecified. However, this may mean that certain computations cannot be performed for this calculus.

In addition to the calculus specification, one could provide the implementation of a qualifier function which for a n-ary calculus takes n geometric objects of the corresponding base type as input and returns the relation holding between these objects. The qualifier function encapsulates the methods for computing the qualitative relations from quantitative geometric descriptions. If it is not provided, the qualify module will not work for this calculus.

For some calculi, it is not possible to provide operations in form of simple tables as in the example. For instance, $\mathcal{OPRA}_m$ has an additional parameter that specifies the granularity and influences the number of base relations. Thus, the operations can only be provided in procedural form, meaning the result of the operations are computed from the input relations when they are required. For these cases, SparQ allows to provide the operations as implemented functions and uses a caching mechanism to store often required results.

3.13.2. Specification Reference

Any specification must be in the form

```
(def-calculus <NAME>
  {<SLOT-AND-OPTIONS>}* )
```

Listing 3.1: Specification of a simple ternary calculus for reasoning about distances.

```
;;; RELDISTCALCULUS
;;;

(def-calculus  "Relative distance calculus (reldistcalculus)"
  :arity :ternary

  :identity-relation same

  :basis-entity :point
  :qualifier #'(lambda (p1 p2 p3)
                 (let ((d12 (point-distance2 p1 p2))
                       (d13 (point-distance2 p1 p3)))
                   (cond ((< d12 d13) 'closer)
                         ((> d12 d13) 'farther)
                         (t           'same))))

  :base-relations (same closer farther)

  :inverse-operation ((same (same closer farther))
                      (closer (same closer farther))
                      (farther (same closer farther)))

  :shortcut-operation ((same same)
                      (closer farther)
                      (farther closer))

  :homing-operation ((same (same closer farther))
                     (closer (same closer farther))
                     (farther (same closer farther)))

  :composition-operation ((same same (same closer farther))
                          (same closer (same closer farther))
                          (same farther (same closer farther))
                          (closer same (same closer farther))
                          (closer closer (same closer farther))
                          (closer farther (same closer farther))
                          (farther same (same closer farther))
                          (farther closer (same closer farther))
                          (farther farther (same closer farther))))
```

where `<NAME>` is a textual description of the calculus enclosed in double quotes. `<SLOT-AND-OPTIONS>` refers to a collection of calculus properties (in no particular order) that consists of pairs of a so-called slot specifier (always starting with a colon) and slot-specific options. Table 3.6 gives an overview. The abbreviations `SRC` and `LIB` stand for source code specification and library link which are explained thereafter.

3.13.3. Operation Specification

Calculi comprise the definition of operations like converse or composition for which SparQ provides three principles ways of specification:

1. Tabular form (suits most standard calculi)
2. Lisp source code
3. Reference to C function in external library

The tabular form is straight-forward. In the case of a unary operation (such as e.g., converse) it is simply a space-separated list of lists that give a relation and the respective outcome when applying the operation. Note that the operation needs only to be defined for base relations! In the case of the binary composition operation the tabular form is a list of lists that give the result for any combination of two base relations.

The last two options of operation specification are particular relevant for defining a quantifier or defining parametrical calculi, i.e., calculi that are instantiated by some parameter(s). In SparQ, these calculi use identifiers that end with a minus “ `-`”, e.g., `opra-`.

Lisp source operation specification

Definitions may be supplied as Lisp function which then will be compiled by SparQ. Lisp functions need to be denoted as lambda expressions. Here is an (slightly silly) example for specifying base relations by a Lisp function:

```
:base-relations #'(lambda () (list 'closer 'farther 'same))
```

As can be observed, relations are simply returned as lists of relations and symbols are used to represent base relations. When specifying an operation that requires an input parameter to the function (e.g., converse, composition), then these are the arguments passed to the provided function. If a parametric calculus is defined, then the parameter will be accessible in by the globally visible parameter `calculi:*calculus-parameter*`.

Altogether, the exemplary specification from above is equivalent to

```
:base-relations (closer farther same)
```

slot	arguments	description
`:arity`	`:binary`|`:ternary`	arity of the calculus
`:base-relations`	`(r {r-i}*)` | `LIB` | `SRC`	list of base relations
`:identity-relation`	`r`	relation r that holds in $A\,r\,A$
`:consistency`	`method`	method for deciding consistency, one of `:a-closure`, `:n-ary-closure`, `:scenario-consistency`, `:a-consistency`, `:n-ary-scenario-consistency`
`:converse-operation`	`({(r1 r1-cnv)}*)` | `LIB` | `SRC`	lists of relations and their corresponding converse relation, reference to external library function, or Lisp code
`:inverse-operation`	`({(r1 r1-inv)}*)` | `LIB` | `SRC`	lists of relations and their corresponding inverse relation, reference to external library function, or Lisp code
`:shortcut-operation`	`({(r1 r1-sc)}*)` | `LIB` | `SRC`	lists of relations and their corresponding shortcut relation, reference to external library function, or Lisp code
`:homing-operation`	`({(r1 r1-hm)}*)` | `LIB` | `SRC`	lists of relations and their corresponding homing relation, reference to external library function, or Lisp code
`:composition-operation`	`({(r1 r2 r1r2-cmp)}*)` | `LIB` | `SRC`	lists of two relations and their composition, reference to external library function, or Lisp code
`:n-ary-composition-operation`	`({(r1 r2 ... rn r-cmp)}*)` | `LIB` | `SRC`	lists of n relations and their composition, reference to external library function, or Lisp code
`:basis-entity`	`:1d-point` | `:interval` | `:2d-point` | `:dipole` | `:2d-oriented-point` | `:2d-box`	type of the basis objects, domain of the calculus
`:parametric?`	`t` | `nil`	Boolean whether the calculus depends on a parameter or not
`:qualifier`	`:algebraic` | `SRC` | `LIB`	qualifier specification (`:algebraic` is not yet fully functional)

Table 3.6.: Overview of the slots in calculi definitions and their options

Note: Though you must use lambda expressions to specify lisp functions, you may define additional functions or parameters in the calculus definition. As the Lisp programmer would expect, the calculus definition is processed by the Lisp compiler and `def-calculus` is a (quite complex) compiler macro.

Lisp source qualifier specification

Providing a lisp source definition for qualifiers is similar to specifying operations. In the case of a binary calculus you need to provide a 2-argument function, in the case of a ternary calculus a 3-argument function. These arguments are exactly those of the command qualify but without the object identifier, i.e., a qualifier for the point-based calculus (e.g., cardinal directions) requested by the command "qualify cardir ((A 2 3) (B 1 3) (C -3 2))" will lead to calls passing the lists `(2 3)`, `(1 3)`, or `(-3 2)` as arguments. Supplied functions are not required to do any error checking, as this is taken care of already.

External operation libraries

All operations may be specified by giving a reference to an external library by writing `(external-lib LIBNAME C_FUNC)`. Hereby, `LIBNAME` must be a string (delimited by double quotes) referring to a shared library inside SparQ's subdirectory `Lib/bin`, `C_FUNC` (a string too) gives the name of the corresponding C function to call. In principal, any programming language can be used as long as it allows for building a shared library that provides functions that follow the C calling convention. The signature of the C function for unary operations must be

```
const char* C_FUNC( const char* param, const char* relation)
```

When called by SparQ, `C_FUNC` is passed the currently active calculus parameters (ignore, if your calculus does not use parameters), i.e., all characters that are appended to the last "-" in the calculus name. As second argument, the relation is passed. The function needs to return the result in the same form SparQ uses as print form for relations. In case of returning disjunctions, this means a space-separated list enclosed in parentheses.

When defining a binary operation (composition), the signature looks as follows:

```
const char* C_FUNC( const char* param, const char* r1, const char* r2)
```

Obviously, two relations need to be passed to the composition operation.

External qualifier libraries

Similar as with operation specification, external libraries can be used for defining a qualifier module for SparQ. The signature of the external functions is as follows:

```
const char* C_FUNC( const char* param, double P1, double P2, ...)
```

Put differently, SparQ passes (besides the calculus parameter) all spatial information as double precision floats to the function. The amount of parameters depends on:

- the arity of the calculus
- the dimension of the calculus' basis entities

Let a denote the arity of a calculus (either $a = 2$ or $a = 3$) and let d denote the dimension of the basis entity, then $a \cdot d$ double parameters are passed. Dimensions of currently supported basis entities are as follows:

basis entity	d
1d-point	1
interval	2
2d-point	2
dipole	4
2d-oriented-point	4
2d-box (axis aligned rectangle)	4
polygon	2× number of vertices

Limitations: Passing coordinates as fixed precision floating point numbers can introduce difficulties that arise due to rounding in computer arithmetics. If no special care is taken then it can easily happen that the relation between A and B obtained by qualification is not the converse of the relation obtained by qualifying the relation between B and A.

To avoid this issue SparQ can use precise (rational) arithmetics and qualifiers implemented in Lisp source code take advantage of this too. For external C functions this is not possible though.

3.13.4. Algebraic Relation Specification

An algebraic calculus specification builds the basis for algebraic reasoning. Base relations need to be specified by systems of polynomial equations over a real-valued domain $\mathbb{R}^n$. Such specification is possible to a wide range of spatio-temporal calculi, but it is, for example, not possible to specify relations in an arbitrary topological space as considered as domain of the RCC calculus family.

In a first step, the base objects of a qualitative calculus need to be represented algebraically. Objects involved in a constrained network (e.g., represented by variables A, B, C, etc.) are represented by tuples of real-valued variables. Currently, SparQ supports the following base entities:

basis entity	algebraic representation	variable representation of A, B, C
1d-point	x	`ax; bx; cx`
interval	a, b	`a1, a2; b1,b2`
2d-point	x, y	`ax, ay; bx, by; cx, cy`
2d-oriented-point	x, y, dx, dy	`ax, ay, adx, ady; bx,by, bdx, bdy; cx,cy,cdx,cdy`
dipole	sx, sy, ex, ey	`sax, say, eax, eay; sbx, aby, ...`
2d-box	x_1, y_1, x_2, y_2	`ablx, ably, atrx, atry, bblx, bbly, ...` (bl: bottom left, tr: top right)

In case of representing oriented points (see Sec. A.12), the variables dx, dy give a direction vector. In contrast, specification of a dipole (see Sec. A.11) is based on the start point sx, sy and the end point ex, ey. Note that the algebraic representation of basis entities corresponds to the form of specifying scenarios to processed by the quantifier module.

The variable representation is used when specifying qualitative relations. To specify a qualitative relation $r(A, B)$, a set of multivariate polynomial equations need to be provided such that

$$\begin{array}{rcl} p_{r,1}(x_1, x_2, \ldots, x_k) & < & 0 \\ p_{r,2}(x_1, x_2, \ldots, x_k) & < & 0 \\ & \vdots & \\ p_{r,i}(x_1, x_2, \ldots, x_k) & = & 0 \\ p_{r,i+1}(x_1, x_2, \ldots, x_k) & = & 0 \\ & \vdots & \\ p_{r,j}(x_1, x_2, \ldots, x_k) & > & 0 \\ & \vdots & \\ p_{r,k}(x_1, x_2, \ldots, x_k) & > & 0 \end{array}$$

is satisfied if and only if $r(A, B)$ holds whereby $x_1, \ldots x_k$ stand for the variable representation introduced above.

Polynomials are denoted in a list based syntax, for example:
`(0 = (1 ((ax 1))) (-1 ((bx 1))))` stands for $0 = 1 \cdot ax^1 - 1 \cdot bx^1$.

3.14. Extending SparQ

Beyond adding new calculi, SparQ offers a simple mechanism to introduce new tools. The aim of this method is to provide means for adding additional methods, either specific to a certain calculus, or general ones. At startup, SparQ will evaluate the contents of the file `Lib/extensions.lisp` inside the SparQ directory in which tool declarations are expected. Tools declared in `Lib/extensions.lisp` are in all regards equivalent to the tools internally provided by SparQ.

Tool declaration is based on a simple macro `def-tool` in which argument syntax and the call to the actual tool code are declared. Here's the syntax:

(`def-tool` (*argument**)
  `:documentation` "*brief description* "
  [`:requires` *file*|*list-of-files*]
  code)

Tool code inside `def-tool` needs to be written in Lisp, but of course it can just be used to call some external non-Lisp code.

- Please refer to the SBCL manual on how to invoke external C/C++ code residing in shared libraries—it's easy!
- The SBCL manual also explains all details of invoking external programs and obtaining the output

SparQ involves a complex type system which shapes the way tool arguments are declared. This is covered in Section 4 and we only give some examples here sufficient declaring simple tools:

```
(def-tool ("translate" (c (calculus allen)) (csp constraint-network c) ↘
   ↪"point-caluclus")
  :documentation "converts CSP with Allen relations to point-calculus"
  :requires      "translators/my-allen->pc-translator.lisp"
  (do-the-work csp))
```

This command would declare a tool "translate" that would take as input two parameters, `c` which is a calculus and `csp`, which is a constraint-network. More precisely, `c` must be a calculus of type "allen" which, is satisfied by Allen's interval algebra. Additionally, the constraint-network must be defined over the calculus `c`, i.e., it must only involve Allen relations. If matching parameters are supplied, the function `do-the-work` is invoked which is assumed to reside in the file specified by `:requires`. With command declarations, overloading a tool similar to object-oriented programming is possible.

```
(def-tool ("list-directory" (which symbol))
  :documentation "retrieves contents of a directory"
  (with-output-to-string (dir)
    (run-program "/bin/ls" (list (symbol-name which)) :output dir)))
```

This tool shows a simple way of invoking an external program (here: the Unix ls tool) and retrieving the output generated. The only argument to this command, `which`, is declared to be of type `symbol`, the Lisp type corresponding to identifiers in other programming languages.[2]

[2]Lisp programmers may wonder about case-(in)sensitivity here: SparQ uses a case-sensitive read-table for parsing user input. Later, case information is stripped away for identifiers. Thus, symbol-name as used here will yield a string preserving cases.

4. Internals

This section provides some information about the internals of SparQ together with planned extensions. SparQ is under current development in the project R3-[Q-Shape]. If you have any questions, additions, or recommendations we'd be glad getting into contact.

4.1. Implementation Details

(to be supplemented)

4.1.1. SparQ type systen

SparQ takes a conflicting approach to data types. On the one hand side, there exists a sophisticated type system that is used to declare tools and to select appropriate methods, but on the other hand side SparQ aims at stripping away any type information to facilitate fast bit bang operations when it comes to reasoning. In what follows, we explain the sophisticated type system as this is exhibited through the extension facility of SparQ. Keep in mind that internal functions are often designed to operate on other types of data though.

The type system used in SparQ comprises the full type system of Lisp, both object-oriented types/classes and regular types. The most important use of types is declaring new commands for SparQ. We start by a practical example:

```
(def-tool ("check-matches" (c (calculus rcc8)) (csp constraint-network c) (timeout real) (option (member first all)) (p (sparq:list-of simple-polygon)))
    :documentation "Checks whether a csp with RCC-8 relations matches quantitative input data given as polygonals"
    :requires      "some/external/code.lisp"
    (do-the-check c csp timeout option p))
```

This imaginary tool could be invoked as follows:

```
SparQ> check-matches rcc8 ((A (po eq) B) (A ntpp C)) all ((A (1.0 2.3
       29.2 4.0 23.0 1.0)) (B (1.0 1.0 2.0 2.0 3.1 2.0)) (B (-1.0 1.0
       2.0 3.0 3.4 4.0)))
```

Our tool declaration makes use of several facets of the type system:

`"check-matches"` Though not precisely a type specifier, string arguments are introduced as short-hand notations for types that only consists of a single object (the string given) and for which no argument will be defined.

`(calculus rcc8)` Denotes a SparQ built-in type specialized to `rcc8` (see next section on implementation details). This roughly corresponds to Lisp's type specializers such as, for example `integer` vs. `(integer 0 10)`.

`constraint-network c` declares a type `constraint-network` which additionally depends on the argument `c` in the command's lambda-list. The argument `c` of type `(calculus rcc8)` will be supplied to the parser dedicated to the type `constraint- network`—see next section for details. SparQ automatically determines a suitable order in which to parse arguments.

`real` a simple real number (which may be either an integer or fractional number type)

`(member first all)` the argument `option` is declared as a standard Lisp type using the type constructor `member` to declare a type that exclusively consists of the two symbols `first` and `all`. Similarly, all other Lisp type constructors can be used.

`(sparq:list-of simple-polygon)` combines a SparQ-specific list type specialized to the type `simple-polygon`, i.e., all members of the list are declared to be of the type `simple-polygon`.

SparQ primitive class

New types (in particularly those used in the user/tool interface) are declared as classes inheriting from the class `sparq:primitive`, which is defined in the file "commands". There are some methods defined for the purpose of interfacing with the user:

`parse-primitive` — method dedicated to parsing user input

`initialize-primitive` — method for post-parsing object initialization. This method exists to enable delaying costly object initialization until all parsing has been completed successfully

`describe-primitive` — writing a textual description to be used with SparQ help command

In order to develop a new type, declare an appropriate class and specialize the parse-primitive method using an EQL-specializer.

```
(defclass my-type (sparq:primitive)
 ;; slot-definitions
)
```

```
(defmethod sparq:parse-primitive ((x (eql 'my-type)) expression &rest extra)
 ;; dedicated parsing code goes here
 ;; return (cons :FAIL "error-msg") if parsing fails
)

(defmethod sparq:initialize-primitive ((x my-type))
  ;; any additional initialization code would go here
)

(defmethod sparq:describe-primitive ((x my-type) stream)
  ;; print type description to stream
)

(defmethod print-object ((x my-type) stream)
  ;; specialized printing code
)
```

The method `parse-primitive` needs a closer look: in order to parse dependent types and type parameters such as, for example, `(calculus allen)`, SparQ will invoke the parse-primitive method of the appropriate type (here: `calculus`) and supply the specifiers as `&rest` parameters. It is the duty of `parse-primitive` to check that any additional specializers are satisfied.

In case of dependent types, additional information will also be supplied as `&rest/&key` parameter. Consider the following tool declaration:

```
(def-tool ("constraint-reasoning" (c calculus) "refine" (cn constraint-network c) (cn2 constraint-network c))
 ...)
```

Here, the constraint-networks `cn` and `cn2` both depend on the calculus `c`. When parsing the arguments `cn` and `cn2`, SparQ will supply the calculus parameter `c` as `&key` argument to `parse-primitive` EQL-specialized to `constraint-network`; parsing then roughly looks like this:

```
(defmethod parse-primitive ((x (eql 'constraint-network)) expression &key calculus)
 ;; step 1: check that "expression" is a proper CSP
 ;; step 2: check that only relations from "calculus" are used
 ...)
```

Please bear in mind that deriving new classes from primitive is only required when defining complex data types—the Lisp type system is usually powerful enough to declare what you need and providing you the data in a friendly, list-based format. In this case, no parsing etc. needs to be implemented.

4.2. Outlook — Planned Extensions

Besides extending the set of supported qualitative calculi, the following extensions are currently planned for the future:

tool integration — we are planning to provide further interfaces to exchange calculus specifications and reasoning with other reasoners

Further goals are to continue the optimization of the algorithms employed in SparQ, for instance by applying maximal tractable subsets for the constraint reasoning part, and to include new results from the QSR community as they become available, in particular with respect to constraint reasoning techniques for calculi for which the standard algebraic closure algorithms are insufficient to decide consistency.

Again, please feel free to contact us if you have any ideas or wishes concerning the extension or improvement of SparQ.

Bibliography

J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, pages 832–843, November 1983.

P. Balbiani, J. Condotta, and L. Fariñas del Cerro. Tractability results in the block algebra. *J. Log. Comput.*, 12(5):885–909, 2002.

A G Cohn, B Bennett, J M Gooday, and N Gotts. RCC: A calculus for region based qualitative spatial reasoning. *GeoInformatica*, 1:275–316, 1997.

Anthony G. Cohn. Qualitative spatial representation and reasoning techniques. In Gerhard Brewka, Christopher Habel, and Bernhard Nebel, editors, *KI-97: Advances in Artificial Intelligence, 21st Annual German Conference on Artificial Intelligence, Freiburg, Germany, September 9-12, 1997, Proceedings*, volume 1303 of *Lecture Notes in Computer Science*, pages 1–30, Berlin, 1997. Springer.

Anthony G. Cohn and Shyamanta M. Hazarika. Qualitative spatial representation and reasoning: An overview. *Fundamenta Informaticae*, 46(1-2):1–29, 2001.

Jean-Francois Condotta, Gérard Ligozat, and Mahmoud Saade. A generic toolkit for n-ary qualitative temporal and spatial calculi. In *Proceedings of the 13th International Symposium on Temporal Representation and Reasoning (TIME'06)*, Budapest, Hungary, 2006.

D. A. Cox, J. B. Little, and D. O'Shea. *Using Algebraic Geometry*, volume 185 of *Graduate Texts in Mathematics.* Springer, NY, 1998.

Ivo Düntsch. Relation algebras and their application in temporal and spatial reasoning. *Artificial Intelligence Review*, 23(4):315–357, 2005.

Frank Dylla. *An Agent Control Perspective on Qualitative Spatial Reasoning*, volume 320 of *DISKI.* Akademische Verlagsgesellschaft Aka GmbH (IOS Press), Heidelberg, Germany, 2008. ISBN 978-3-89838-320-2. Doctoral thesis (Universitty of Bremen).

Frank Dylla and Jae Hee Lee. A combined calculus on orientation with composition based on geometric properties. In *ECAI-10*, 2010.

Frank Dylla and Reinhard Moratz. Empirical complexity issues of practical qualitative spatial reasoning about relative position. In *Workshop on Spatial and Temporal Reasoning at ECAI 2004*, Valencia, Spain, August 2004.

Frank Dylla and Reinhard Moratz. Exploiting qualitative spatial neighborhoods in the situation calculus. In Freksa et al. (2005), pages 304–322.

Andrew Frank. Qualitative spatial reasoning about cardinal directions. In *Proceedings of the American Congress on Surveying and Mapping (ACSM-ASPRS)*, pages 148–167, Baltimore, Maryland, USA, 1991.

Christian Freksa. Using orientation information for qualitative spatial reasoning. In A. U. Frank, I. Campari, and U. Formentini, editors, *Theories and methods of spatio-temporal reasoning in geographic space*, pages 162–178. Springer, Berlin, 1992a.

Christian Freksa. Temporal reasoning based on semi-intervals. *Artificial Intelligence*, 1 (54):199–227, 1992b.

Christian Freksa, Markus Knauff, Bernd Krieg-Brückner, Bernhard Nebel, and Thomas Barkowsky, editors. *Spatial Cognition IV. Reasoning, Action, Interaction: International Conference Spatial Cognition 2004*, volume 3343. Springer, Berlin, Heidelberg, 2005.

Hans W. Güsgen. Spatial reasoning based on Allen's temporal logic. Technical Report TR-89-049, International Computer Science Institute, Berkeley, 1989.

Amar Isli and Anthony G. Cohn. An algebra for cyclic ordering of 2d orientations. In *AAAI/IAAI*, pages 643–649, Madison, WI, 1998.

Amar Isli and Anthony G. Cohn. A new approach to cyclic ordering of 2d orientations using ternary relation algebras. *Artificial Intelligence.*, 122(1-2):137–187, 2000.

P. Ladkin and R. Maddux. On binary constraint problems. *Journal of the Association for Computing Machinery*, 41(3):435–469, 1994.

Peter Ladkin and Alexander Reinefeld. Effective solution of qualitative constraint problems. *Artificial Intelligence*, 57:105–124, 1992.

G. Ligozat. Reasoning about cardinal directions. *Journal of Visual Languages and Computing*, 9:23–44, 1998.

Gerard Ligozat. Qualitative triangulation for spatial reasoning. In Andrew U. Frank and Irene Campari, editors, *Spatial Information Theory: A Theoretical Basis for GIS, (COSIT'93), Marciana Marina, Elba Island, Italy*, volume 716 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 1993. ISBN 3-540-57207-4.

Gérard Ligozat. Categorical methods in qualitative reasoning: The case for weak representations. In *Spatial Information Theory: Cognitive and Computational Foundations, Proceedings of COSIT'05*, 2005.

A. K Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

Reinhard Moratz. Representing relative direction as binary relation of oriented points. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006)*, Riva del Garda, Italy, August 2006.

Reinhard Moratz, Jochen Renz, and Diedrich Wolter. Qualitative spatial reasoning about line segments. In W. Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI)*, Berlin, Germany, 2000. IOS Press.

Reinhard Moratz, Frank Dylla, and Lutz Frommberger. A relative orientation algebra with adjustable granularity. In *Proceedings of the Workshop on Agents in Real-Time and Dynamic Environments (IJCAI 05)*, Edinburgh, Scotland, July 2005.

Marco Ragni and Alexander Scivos. Dependency calculus reasoning in a general point relation algebra. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pages 1577–1578. Professional Book Center, 2005.

David A. Randell, Zhan Cui, and Anthony Cohn. A spatial logic based on regions and connection. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR'92)*, pages 165–176. Morgan Kaufmann, San Mateo, CA, 1992.

Jochen Renz and Gérard Ligozat. Weak composition for qualitative spatial and temporal reasoning. In Peter van Beek, editor, *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP 2005)*, volume 3709 of *LNCS*, pages 534–548. Springer, October 2005.

Christoph Schlieder. Reasoning about ordering. In *Proceedings of COSIT'95*, volume 988 of *Lecture Notes in Computer Science*, pages 341–349. Springer, Berlin, Heidelberg, 1995.

Alexander Scivos and Bernhard Nebel. Double-crossing: Decidability and computational complexity of a qualitative calculus for navigation. In *Proceedings of COSIT'01*, Berlin, 2001. Springer.

Alexander Scivos and Bernhard Nebel. The finest of its class: The practical natural point-based ternary calculus $\mathcal{LR}$ for qualitative spatial reasoning. In Freksa et al. (2005), pages 283–303.

Nico van de Weghe. *Representing and Reasoning about Moving Objects: A Qualitative Approach.* PhD thesis, Ghent University, 2004.

M. B. Vilain, H. A. Kautz, and P. G. van Beek. Constraint propagation algorithms for temporal reasoning: A revised report. In *Readings in Qualitative Reasoning about Physical Systems.* Morgan Kaufmann, San Mateo, CA, 1989.

A. Implemented Calculi

In this section, we briefly describe the spatial calculi currently supported by SparQ. Some calculi are actually calculi families for which a set of *calculus parameters* needs to be specified in order to obtain a particular calculus instance. For instance, for the $\mathcal{OPRA}_m$ calculus the granularity (number of partitioning lines) has to be specified as a calculus parameter.

Each calculi description in this section starts with a box summarizing the main characteristics of the considered calculus. The meaning of the entries in the box are explained below:

short name — the name used in SparQ to refer to this calculus

calculus parameters — the parameters that need to be specified whenever using this calculus

arity — the arity of the relations of this calculus (binary or ternary)

entity type — the spatial entities related in this calculus (2D points, oriented 2D points, line segments dipoles, etc.)

description — a short description of the calculus

base relations — a naming scheme or list of the base relations of the calculus

references — references to literature about this calculus

remarks — special remarks concerning the calculus

A.1. Allen's Interval Algebra (IA)

Allen's Interval Algebra overview

calculus identifier	allen, aia, ia
calculus parameters	none
arity	binary
entity type	intervals (defined by a start and end-point) on a unidirectional time line
description	describes the mereotopological relation between two intervals
base relations	b (before), bi (before inverse), m (meets), mi (meets inverse), o (overlaps), oi (overlaps inverse), s (starts), si (starts inverse), d (during), di (during inverse), f (finishes), fi (finishes inverse), eq (equals)
references	Allen (1983)

Allen's interval algebra (IA) (Allen, 1983) relates pairs of time intervals. Time interval x is represented as a tuple of a starting point x_s and end point x_e with $x_s < x_e$ using real numbers, e.g., `(life-of-Bach 1685 1750)`. Altogether 13 base relations are distinguished by comparing the start and end points of the intervals.

relation	term	example	definition
`b`	x before y	xxx	$x_e < y_s$
`bi`	y after x	yyy	
`m`	x meets y	xxxx	$x_e = y_s$
`mi`	y met-by x	yyyy	
`o`	x overlaps y	xxxxx	$x_s < y_s < x_e \wedge$
`oi`	y overlapped-by x	yyyyy	$x_e < y_e$
`d`	x during y	xxx	$x_s > y_s \wedge$
`di`	y includes x	yyyyyyy	$x_e < y_e$
`s`	x starts y	xxx	$x_s = y_s \wedge$
`si`	y started-by x	yyyyyyy	$x_e < y_e$
`f`	x finishes y	xxx	$x_e = y_e \wedge$
`fi`	y finished-by x	yyyyyyy	$x_s > y_s$
`eq`	x equals y	xxxxxxx	$x_s = y_s \wedge$
		yyyyyyy	$x_e = y_e$

A.2. Block-Algebra (BA)

Block Algebra overview

calculus identifier	block-algebra,ba
calculus parameter	none
arity	binary
entity tape	axis-aligned boxes, $(x_{\min}, y_{\min}, x_{\max}, y_{\max})$
description	describes spatial arrangement of boxes by independently projecting to x- and y-axis intervals and applying Allen's Interval Algebra (see Section sec:allen) to both dimensions (see Figure fig:block-algebra)
base relations	relations r_s, whereby r, s are Allen relations which yields $13 \cdot 13 = 169$ base relations
references	Güsgen (1989), Balbiani et al. (2002)

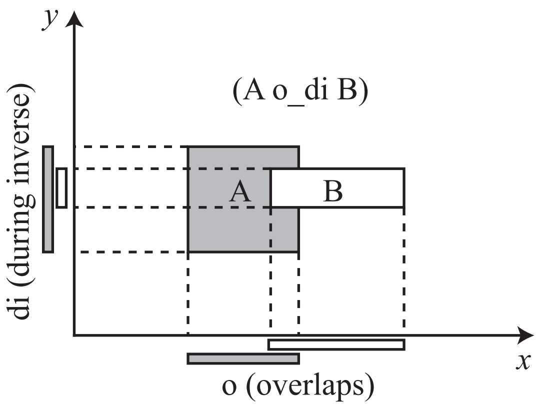


Figure A.1.: Example relation o_di in the block algebra

A.3. Cardinal Direction Calculus

Cardinal Direction Calculus overview

calculus identifier	cardir
calculus parameters	none
arity	binary
entity type	2D points
description	describes the orientation of two point regarding and absolute orientation
base relations	N, NE, E, SE, S, SW, W, NW, EQ
references	Frank (1991), Ligozat (1998)

Frank (1991) introduced the cardinal direction calculus.[1] The euclidian plane $\mathcal{P}$ is partitioned into regions with respect to a reference point R and a global *west-east/south-north* reference frame. Any point $P \in \mathcal{P}$ belongs to one of the nine basic relations: **N**orth, **N**orth**E**ast, **E**ast, **S**outh**E**ast, **S**outh, **S**outh**W**est, **W**est, **N**orth**W**est, or **Eq**ual. The model is depicted in Figure A.2.

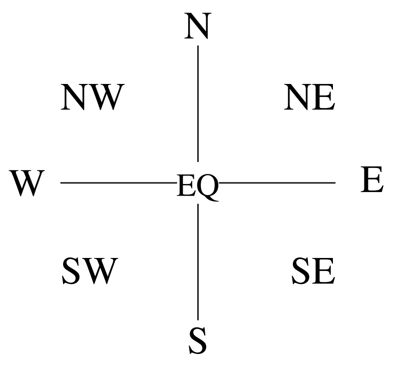


Figure A.2.: Base relations of the cardinal direction calculus.

[1]Frank introduced two different variants, called projection-based and cone-based. This calculus definition implements the projection-based variant.

A.4. The Region Connection Calculus family (RCC)

The calculi from the RCC family (RCC-8 and RCC-5) allow mereotopological reasoning (reasoning about connection and part-of relationships) about simple regions in the plane. Other domains involving regions can also be considered in the context of RCC, e.g. 3D regions, or non-simple regions in the plane, which can affect the correctness of the constraint-based reasoning algorithms. Since so far no qualifier for RCC is available in SparQ, the exact domain is actually still not determined. However, we will assume the case of simple regions in the plane in the following.

RCC-8

Region Connection Calculus 8 (RCC-8) overview

calculus identifier	rcc-8
calculus parameters	none
arity	binary
entity type	simple regions in the plane
description	describes the mereotopological relation between two regions
base relations	dc (disconnected), ec (externally connected), po (partially overlapping), eq (equal), tpp (tangential proper part), ntpp (non-tangential proper part), tppi (tangential proper part inverse), ntppi (non-tangential proper part inverse)
references	Randell et al. (1992), Cohn et al. (1997)
remarks	no qualifier is available for this calculus yet

RCC-8 is the more fine-grained variant of RCC calculi. It distinguishes the eight base relations dc (disconnected), ec (externally connected), po (partially overlapping), eq (equal), tpp (tangential proper part), ntpp (non-tangential proper part), tppi (tangential proper part inverse), and nttpi (non-tangential proper part inverse) which are illustrated in Fig. A.3.

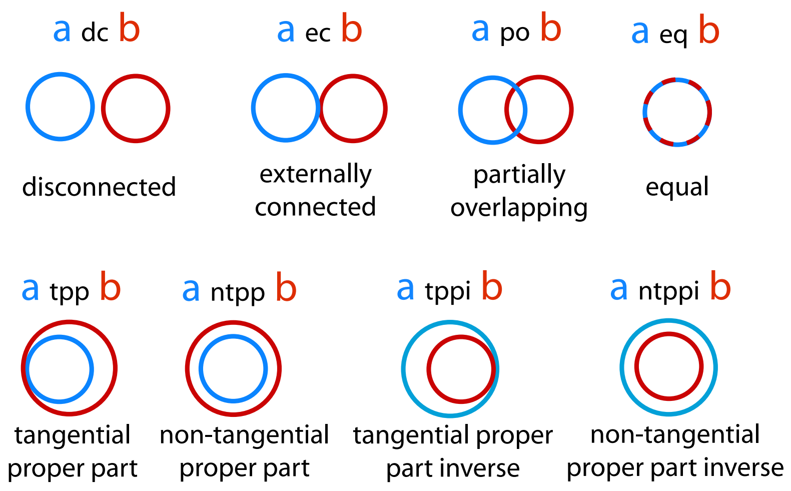


Figure A.3.: The RCC-8 base relations.

RCC-5

Region Connection Calculus 5 (RCC-5) overview

calculus identifier	rcc-5
calculus parameters	none
arity	binary
entity type	simple regions in the plane
description	describes the mereotopological relation between two regions
base relations	dr (discrete from), po (partially overlapping), eq (equal), pp (proper part), ppi (proper part inverse)
references	Cohn et al. (1997)
remarks	no qualifier is available for this calculus yet

RCC-5 is a coarser version of RCC-8. The RCC-8 relations dc and ec are combined into one relation called dr. Similarly, ntpp and tpp are combined into pp and ntppi and tppi into ppi.

A.5. Dependency Calculus

Dependency Calculus overview

calculus identifier	depcalc, dep
calculus parameters	none
arity	binary
entity type	-
description	describes the order between nodes in a network
base relations	<, =, >, ^, ~
references	Ragni and Scivos (2005), **?**
remarks	no qualifier is available for this calculus yet

The Dependency Calculus (DC) represents pairs of points regarding their dependencies in a partial ordered structure. Therefore, it meets all requirements to describe dependencies in networks. If x, y are points in a partial order $\langle T, \leq \rangle$ the base relations are defined as follows (Ragni and Scivos, 2005):

$$\begin{aligned}
x < y &\quad \text{iff} \quad x \leq y \text{ and not } y \leq x.\\
x = y &\quad \text{iff} \quad x \leq y \text{ and } y \leq x.\\
x > y &\quad \text{iff} \quad y \leq x \text{ and not } x \leq y.\\
x \;\hat{}\; y &\quad \text{iff} \quad \exists z\; z \leq y \wedge z \leq x \text{ and neither } x \leq y \text{ nor } y \leq x.\\
x \sim y &\quad \text{iff} \quad \text{neither } \exists z\; z \leq y \wedge z \leq x \text{ nor } x \leq y \text{ nor } y \leq x.
\end{aligned}$$

A.6. Singlecross Calculus (SCC)

Singlecross calculus (SCC) overview

calculus identifier	scc
calculus parameters	none
arity	ternary
entity type	2D points
description	relates the referent c relative to the line between origin a and relatum b and the orthogonal line thru b, resulting in 11 base relations
base relations	0..7 and b (for b=c), dou, tri
references	Freksa (1992a)

The single cross calculus is a ternary calculus that describes the direction of a point C (the referent) with respect to a point B (the relatum) as seen from a third point A (the origin). It has originally been proposed in Freksa (1992a). The plane is partitioned into regions by the line going through A and B and the perpendicular at B. This results in eight possible directions for C as illustrated in Fig. A.4. We denote these base relations by numbers from 0 to 7 instead of using linguistic prepositions, e.g. 2 instead of *left*, as originally done in Freksa (1992a). Relations 0, 2, 4, 6 are linear ones, while relations 1, 3, 5, 7 are planar. In addition, three special relations exist for the cases $A \neq B = C$ (**b**), $A = B \neq C$ (**dou**), and $A = B = C$ (**tri**). A single cross relation rel_{SCC} is written as $A, B\ rel_{SCC}\ C$, e.g. $A, B\ \mathbf{4}\ C$ or $A, B\ \mathbf{dou}\ C$. The relation depicted in Fig. A.4 is the relation $A, B\ \mathbf{5}\ C$.

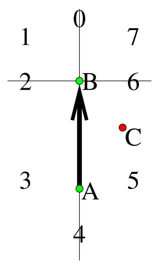


Figure A.4.: The Single Cross Reference System

A.7. Doublecross Calculus (DCC)

Doublecross calculus (DCC) overview

calculus identifier	dcc, double-cross
calculus parameters	none
arity	ternary
entity type	2D points
description	relates the referent C relative to the line between origin A and relatum B and the orthogonal lines through A and B, resulting in 17 base relations
base relations	0_4, 1_5, 2_5, 3_5, 3_6, 3_7, 4_0, 5_1, 5_2, 5_3, 6_3, 7_3, 4_4, b_4, 4_a, dou, tri
references	Freksa (1992a)

The double cross calculus (Freksa, 1992a) can be seen as an extension of the single cross calculus adding another perpendicular, this time at A (see Fig. A.5 (right)). It can also be interpreted as the combination of two single cross relations, the first describing the position of C with respect to B as seen from A and the second with respect to A as seen from B (cf. Fig. A.5 (left)). The resulting partition distinguishes 13 relations (7 linear and 6 planar) denoted by tuples derived from the two underlying SCC reference frames and four special cases, $A = C \neq B$ (**4_a**), $A \neq B = C$ (**b_4**), $A = B \neq C$ (**dou**), and $A = B = C$ (**tri**), resulting in 17 base relations overall. In Fig. A.5 (right) the relation A, B **5_3** C is depicted.

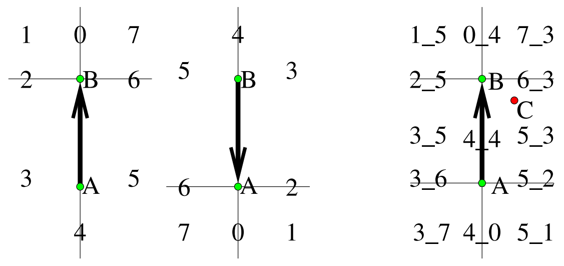


Figure A.5.: The two Single Cross reference frames resulting in the overall Double Cross Calculus reference frame

Alternative Doublecross Calculus (DCC) overview

calculus identifier	adcc, alternative-double-cross
calculus parameters	none
arity	ternary
entity type	2D points
description	relates the referent C relative to the line between origin A and relatum B and the orthogonal lines through A and B, resulting in 17 base relations
base relations	0-12, a, b, dou, tri
references	Freksa (1992a)

In the literature also a single numbered notation can be found. We refer to this nomenclatur as the Alternative Doublecross Calculus. Apart from relations b_4, 4_a the mapping between tuple notation and alternative notation is given in Figure A.6. b_4 corresponds to **b** and 4_a to **a**. **dou** and **tri** are defined as above.

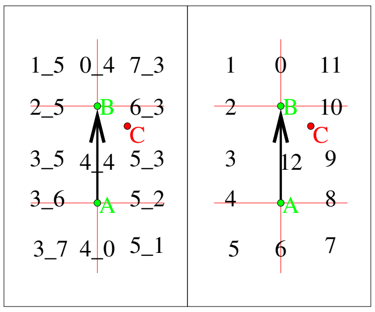


Figure A.6.: Schematic overview of Doublecross base relation names in tuple notation and the alternative notation.

A.8. FlipFlop Calculus With $\mathcal{LR}$ Refinement

FlipFlop calculus (FFC) overview

calculus identifier	ffc, ff, flipflop
calculus parameters	none
arity	ternary
entity type	2D points
description	relates the referent C relative to the line segment starting at origin A and ending at relatum B resulting in nine base relations
base relations	l (left), r (right), f (front), b (back), i (inside), s (start), e (end), dou, tri
references	Ligozat (1993), Scivos and Nebel (2005)
remark	SparQ uses the $\mathcal{LR}$ refinement in its implementation of the FFC

The FlipFlop calculus proposed in Ligozat (1993) describes the position of a point C (the referent) in the plane with respect to two other points A (the origin) and B (the relatum) as illustrated in Fig. A.7. It can for instance be used to describe the spatial relation of C to B as seen from A. For configurations with $A \neq B$ the following base relations are distinguished: C can be to the **l**eft or to the **r**ight of the oriented line going through A and B, or C can be placed on the line resulting in one of the five relations **i**nside, **f**ront, **b**ack, **s**tart ($C = A$) or **e**nd ($C = B$) (cp. Fig. A.7). Relations for the case where A and B coincide were not included in Ligozat's original definition (Ligozat, 1993). This was done with the $\mathcal{LR}$ refinement (Scivos and Nebel, 2005) that introduces the relations **dou** ($A = B \neq C$) and **tri** ($A = B = C$) as additional relations, resulting in 9 base relations overall. A $\mathcal{LR}$ relation $rel_{\mathcal{LR}}$ is written as $A, B\ rel_{\mathcal{LR}}\ C$, e.g. A, B **r** C as depicted in Fig. A.7.

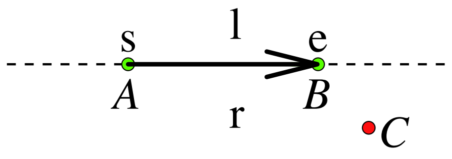


Figure A.7.: The reference frame for the $\mathcal{LR}$ calculus, an enhanced version of the FlipFlop Calculus

A.9. Cycord Family

Cycord calculi (Isli and Cohn, 1998) are based on oriented line segments, either defined by a direction directly, or start and end point (cf. dipoles in A.11). The relations between two line segments X and Y can take one of the four values: e (equal alignment), l (oriented left, i.e. the angle α between X and Y is $\alpha \in (0, \pi)$, o (opposite, i.e. $\alpha = \pi$, or r (oriented right, i.e. $\alpha \in (\pi, 2\pi)$. The binary case, where two oriented line segments are related, is equivalent to the alignment calculus (Section A.10). In the ternary case three line segments are considered and the relation consists of a three tuple (r_{XY}, r_{XZ}, r_{YZ}) with r_{XY} denoting the binary relation between X and Y, r_{XZ} between X and Z, and $r_{YZ})$ between Y and Z. Out of the 64 possible only 24 are valid, i.e. only 24 are consistent in terms of the binary Cycord definition.

Binary CyCord

Binary Cycord calculus overview

calculus identifier	cycord2, cc2
calculus parameters	none
arity	binary
entity type	dipoles in the plane (oriented line segments)
description	relates two dipoles regarding their relative orientation (alignment)
base relations	e, l, o, r
references	Isli and Cohn (1998), Isli and Cohn (2000)

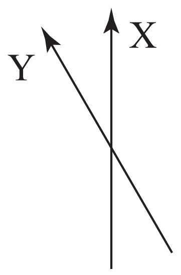


Figure A.8.: A binary Cycord relation: l

Ternary CyCord

Ternary Cycord calculus overview

calculus identifier	cycord3, cc3
calculus parameters	none
arity	ternary
entity type	dipoles in the plane (oriented line segments)
description	relates two dipoles regarding their relative orientation (alignment)
base relations	eee, ell, eoo, err, lel, lll, llo, llr, lor, lre, lrl, lrr, oeo, olr, ooe, orl, rer, rle, rll, rlr, rol, rrl, rro, rrr
references	Isli and Cohn (1998), Isli and Cohn (2000)

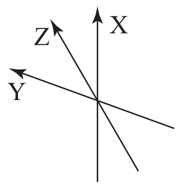


Figure A.9.: A ternary Cycord relation: llr

A.10. The Geometric Orientation (Alignment) Calculus

The Geometric Orientation (Alignment) Calculus overview

calculus identifier	geomori, ori, align
calculus parameters	none
arity	binary
entity type	dipole
description	describes the alignment of two oriented line segments
base relations	P, $+$, O, $-$
references	Dylla (2008), Dylla and Lee (2010)
remarks	no qualifier is available for this calculus yet; conceptually equivalent to Binary Cycord (see Appendix A.9)

The Geometric Orientation Calculus, also called Geometric Alignment Calculus, relates the alignment of two oriented line segments. The alignment is derived by shifting both points of the second dipole such that the starting points of dipole A and B coincide. Dipole B may point in the same direction as A (parallel), in opposite direction as A (opposite-parallel), somewhere to the left of dipole A (mathematically positive), or somewhere to the right of dipole A (mathematically negative). An example with two dipoles which are aligned positively is depicted in Figure A.10. The alignment of dipoles is part of the development of the fine grained Dipole Relation Algebra with paralellism in (Dylla and Moratz, 2005).

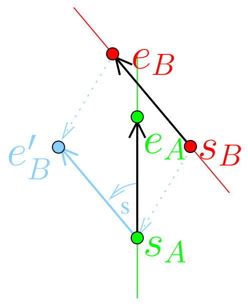


Figure A.10.: Example of two dipoles which are aligned positively.

A.11. Dipole Calculus Family

A dipole is an oriented line segment as e.g. determined by a start and an end point. We will write $\vec{d}_{AB}$ for a dipole defined by start point A and end point B. The idea of using dipoles was first introduced by Schlieder (1995) and extended resulting in the coarse-grained Dipole Relation Algebra $\mathcal{DRA}_c$ (Moratz et al., 2000). Later, a fine-grained version of the dipole calculus ($\mathcal{DRA}_f$) has been proposed (Dylla and Moratz, 2005) and which has further been extended to $\mathcal{DRA}_{fp}$ (Dylla and Moratz, 2005). In SparQ, currently only the coarse-grained version $\mathcal{DRA}_c$ is available.

Coarse-grained Dipole Relation Algebra ($\mathcal{DRA}_c$)

Coarse-grained dipole calculus ($\mathcal{DRA}_c$) overview

calculus identifier	dra-24, dipole-coarse
calculus parameters	none
arity	binary
entity type	dipoles in the plane (oriented line segments)
description	relates two dipoles using the FlipFlop relations between the start and end point of one dipole and the other dipole
base relations	4-symbol words where each symbol can be either l (left), r (right), s (start), or e (end) (not all combinations are possible)
references	Moratz et al. (2000)

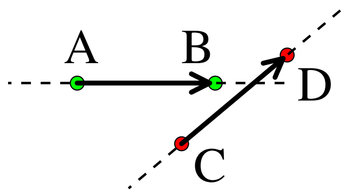


Figure A.11.: A dipole configuration: $\vec{d}_{AB}$ *rlll* $\vec{d}_{CD}$ in the coarse-grained dipole relation algebra ($\mathcal{DRA}_c$).

The coarse-grained dipole calculus variant ($\mathcal{DRA}_c$) describes the orientation relation between two dipoles $\vec{d}_{AB}$ and $\vec{d}_{CD}$ with the preliminary of A, B, C, and D being in general position, i.e. no three disjoint points are collinear. Each base relation is a 4-

tuple (r_1, r_2, r_3, r_4) of FlipFlop relations relating a point from one of the dipoles with the other dipole. r_1 describes the relation of C with respect to the dipole $\vec{d}_{AB}$, r_2 of D with respect to $\vec{d}_{AB}$, r_3 of A with respect to $\vec{d}_{CD}$, and r_4 of B with respect to $\vec{d}_{CD}$. The distinguished FlipFlop relations are **l**eft, **r**ight, **s**tart, and **e**nd (see Fig. A.7). Dipole relations are usually written without commas and parentheses, e.g. *rrll*. Thus, the example in Fig. A.11 shows the relation $\vec{d}_{AB}$ *rlll* $\vec{d}_{CD}$. Since the underlying points for a $\mathcal{DRA}_c$ relation need to be in general position, r_i can only take the values **l**eft, **r**ight, **s**tart, or **e**nd resulting in 24 base relations.

A.12. Oriented Point Reasoning Algebra With Granularity m ($\mathcal{OPRA}_m$)

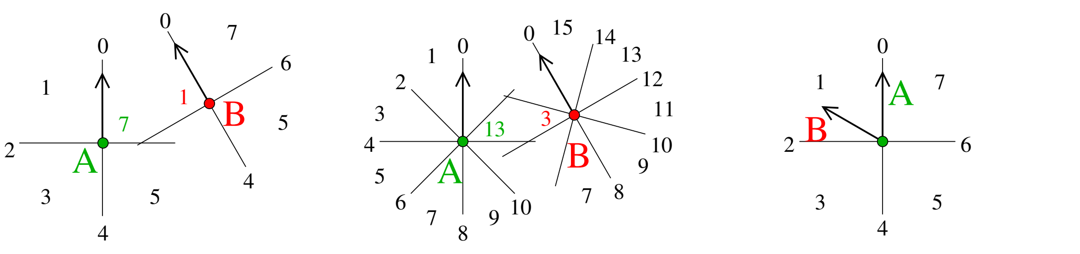
Oriented Point Relation Algebra ($\mathcal{OPRA}_m$) overview

calculus identifier	opra-
calculus parameters	granularity - number of partitioning lines (= number of planar relations / 2), must be > 0
arity	binary
entity type	oriented 2D points
description	relates two oriented points a and b with respect to granularity m
base relations	$[i,j]$ with $i,j \in \{0,..,4m{-}1\}$, if a and b have different positions; $[i]$ with $i \in \{0,..,4m-1\}$ if they have the same position
references	Moratz et al. (2005), Moratz (2006)

The domain of the Oriented Point Relation Algebra ($\mathcal{OPRA}_m$) (Moratz et al., 2005, Moratz, 2006) is the set of oriented points (points in the plane with an additional direction parameter). The calculus relates two oriented points with respect to their relative orientation towards each other. An oriented point $\vec{O}$ can be described by its Cartesian coordinates $x_O, y_O \in \mathbb{R}$ and a direction $\phi_{\vec{O}} \in [0, 2\pi]$ with respect to an absolute reference direction and thus $D = \mathbb{R}^2 \times [0, 2\pi]$.

The $\mathcal{OPRA}_m$ calculus is suited for dealing with objects that have an intrinsic front or move in a particular direction and can be abstracted as points. The exact set of base relations distinguished in $\mathcal{OPRA}_m$ depends on the granularity parameter $m \in \mathbb{N}$. For each of the two related oriented points, m lines are used to partition the plane into $2m$ planar and $2m$ linear regions. Fig. A.12 shows the partitions for the cases $m = 2$ (a) and $m = 4$ (b). The orientation of the two points is depicted by the arrows starting at $\vec{A}$ and $\vec{B}$, respectively. The regions are numbered from 0 to $4m - 1$; region 0 always coincides with the orientation of the point. An $\mathcal{OPRA}_m$ base relation $rel_{\mathcal{OPRA}_m}$ consists of a pair (i, j) where i is the number of the region of $\vec{A}$ which contains $\vec{B}$, while j is the number of the region of $\vec{B}$ that contains $\vec{A}$. These relations are usually written as $\vec{A}\ {}_m\angle_i^j\ \vec{B}$ with $i, j \in \mathcal{Z}_{4m}$[2]. Thus, the examples in Fig. A.12 depict the relations $\vec{A}\ {}_2\angle_7^1\ \vec{B}$ and $\vec{A}\ {}_4\angle_{13}^3\ \vec{B}$. Additional base relations called *same relations* describe situations in which the positions of both oriented points coincide. In these cases, the relation is determined

[2] $\mathcal{Z}_{4m}$ defines a cyclic group with $4m$ elements.



(a) $m = 2$: $\vec{A}\ {}_2\angle^1_7\ \vec{B}$ (b) $m = 4$: $\vec{A}\ {}_4\angle^3_{13}\ \vec{B}$ (c) case where $\vec{A}$ and $\vec{B}$ coincide: $\vec{A}\ {}_2\angle 1\ \vec{B}$

Figure A.12.: Two oriented points related at different granularities.

by the number s of the region of $\vec{A}$ into which the orientation arrow of $\vec{B}$ falls (as illustrated in Fig. A.12(c)). These relations are written as $\vec{A}\ {}_2\angle s\ \vec{B}$ ($\vec{A}\ {}_2\angle 1\ \vec{B}$ in the example).

The complete set $\mathcal{R}$ of $\mathcal{OPRA}_m$ relations is the power set of the base relations described above.

A.13. Point Calculus (Point Algebra)

Point Calculus (Point Algebra) overview

calculus identifier	point-calculus, pc, point-algebra, pa
calculus parameters	none
arity	binary
entity type	1D points
description	describes the order between two 1D points (values)
base relations	<, =, >
references	Vilain et al. (1989)

The Point Calculus (PC) (Vilain et al., 1989) relates pairs of 1D points, represented by real-valued numbers. Pairs of values are categorized using the three base relations less than (<), equal (=), or greater than (>).

A.14. Qualitative Trajectory Calculus Family

van de Weghe (2004) developed a family of trajectory calculi on the basis of relative trajectories of two moving objects. He investigates representation where he combined subsets of the three different features: change in distance, change to the side, and relative velocity. He also investigated differences in representations based on one dimensional (1D) and two dimensional (2D) entities. The most basic calculus is QTC_{B11} dealing with change in distance in 1D, enhanced with velocity in QTC_{B12}. The extensions to 2D entities is given in QTC_{B21} and QTC_{B22}. QTC_{C21} (QTC_{C22}) extends QTC_{B21} (QTC_{B22}) by relative velocity.

QTC in 1D With Distance

Qualitative Trajectory Calculus in 1D (QTC-B11)

calculus identifier	qtc-b11
calculus parameters	none
arity	binary
entity type	interval (1D trajectory positions at two different time points)
description	describes the relative orientation between two trajectory segments
base relations[3]	++, +-, +O, -+, - -, -O, O+, O-, OO
references	van de Weghe (2004)
remarks	no qualifier is available for this calculus yet

QTC_{B11} represents the relative distance change of two moving objects A and B at timepoints t_i and t_{i+1} . Intuitively, the first character denotes whether A moves towards the starting position of B ($-$), moves away (+, or the distance stays the same (O). With $dist(x, y)$ denoting the distance between two positions and A_i denotes the position of object A at time point t_i moving towards means $dist(A_{i+1}, B_i) < dist(A_i, B_i)$, moving away means $dist(A_{i+1}, B_i) > dist(A_i, B_i)$, and equidistant means $dist(A_{i+1}, B_i) = dist(A_i, B_i)$ The second character represents the change in distance regarding B wrt. A. This results in nine base relations.

[3]For avoiding the necessity to quote every single relation such that leading zeros are not ignored we realized the implementation with O's instead of zeros.

QTC in 1D With Distance and Velocity

Qualitative Trajectory Calculus in 1D with velocity (QTC-B12)

calculus identifier	qtc-b12
calculus parameters	none
arity	binary
entity type	-
description	describes the relative orientation between two trajectory segments
base relations	+++, ++-, ++O, +-+, +- -, +-O, +O+, -++, -+-, -+O, - -+, - - -, - -O, -O+, O+-, O- -, OOO
references	van de Weghe (2004)
remarks	no qualifier is available for this calculus yet

The first two characters of QTC_{B12} represent the same as QTC_{B11}. The third character represents the relative velocitiy between A and B, i.e. whether object A is slower than B ($-$), is faster (+), or both have the same velocity (O). Because the conditions of the three characters interfere in 1D only 17 out of 27 potential relations are feasible.

QTC in 2D With Distance

Qualitative Trajectory Calculus in 2D (QTC-B21)

calculus identifier	qtc-b21
calculus parameters	none
arity	binary
entity type	dipole (2D trajectory positions at two different time points)
description	describes the relative orientation between two trajectory segments
base relations	++, +-, +O, -+, - -, -O, O+, O-, OO
references	van de Weghe (2004)
remarks	no qualifier is available for this calculus yet

QTC_{B21} is similar to QTC_{B11} except dealing with trajectories in 2D instead of only 1D.

QTC in 2D With Distance and Velocity

Qualitative Trajectory Calculus in 2D with velocity (QTC-B22)

calculus identifier	qtc-b22
calculus parameters	none
arity	binary
entity type	-
description	describes the relative orientation between two trajectory segments
base relations	$\{+, O, -\} \times \{+, O, -\} \times \{+, O, -\}$
references	van de Weghe (2004)
remarks	no qualifier is available for this calculus yet

QTC_{B22} is similar to QTC_{B12} except dealing with trajectories in 2D instead of only

1D. In contrast to QTC_{B12} in 2D all 27 potential relations are feasible.

QTC in 2D With Distance and Side

Qualitative Trajectory Calculus in 2D (QTC-C21)

calculus identifier	qtc-c21
calculus parameters	none
arity	binary
entity type	dipole (2D trajectory positions at two different time points)
description	describes the relative orientation between two trajectory segments
base relations	$\{+, O, -\} \times \{+, O, -\} \times \{+, O, -\} \times \{+, O, -\}$
references	van de Weghe (2004)
remarks	no qualifier is available for this calculus yet

QTC_{C21} relations are given by a four character tuple. The first two characters represent the same as a QTC_{B21} relation. The third character denotes whether A moves to the left (−), to the right (+), or on the reference line (O) spanned between A and B at t_i. The fourth character represents the change of B wrt. to this reference line. This results in $3^4 = 81$ base relations.

QTC in 2D With Distance, Side, and Velocity

Qualitative Trajectory Calculus in 2D with velocity (QTC-C22)

calculus identifier	qtc-c22
calculus parameters	none
arity	binary
entity type	-
description	describes the relative orientation between two trajectory segments
base relations	$\{+, O, -\} \times \{+, O, -\} \times \{+, O, -\} \times \{+, O, -\} \times \{+, O, -\}$
references	van de Weghe (2004)
remarks	no qualifier is available for this calculus yet

QTC_{C22} relations are given by a five character tuple. The first four directly map onto QTC_{C21} relations. The fifth character represents the relative velocity between objects A and B. $-$ denotes A being slower than B , $+$ is faster, and O if both move at the same speed.

B. Quick Reference

B.1. Command Summary

`compute-relation c op arg-1 {arg-i}`

Applies operation `op` of calculus `c` to arguments. Operations:

spatial:	
`composition, comp`	$rs \mapsto r \circ s$
`converse, cnv`$^{(2)}$	$r \mapsto r^{\smile}$
`homing, hm`$^{(3)}$	$r \mapsto hm(r)$
`homingi, hmi`$^{(3)}$	$r \mapsto inv(hm(r))$
`inverse, inv`$^{(3)}$	$r \mapsto inv(r)$
`shortcut, sc`$^{(3)}$	$r \mapsto sc(r)$
`shortcuti, sci`$^{(3)}$	$r \mapsto inv(sc(r))$
calculi-theoretic:	
`closure`	$r_1 r_2 \ldots r_n \mapsto Cl(r_1, r_2, \ldots, r_n)$ Cl denotes the minimal set of relations that is closed under composition, converse, and intersection
`base-closure`	$Cl(br_1, br_2, \ldots, br_n)$ Computes closure of the set of base relations
set-theoretic:	
`complement, cmpl`	$r \mapsto r^C$
`minus`	$rs \mapsto r \backslash s$
`union`	$rs \mapsto r \cup s$
`intersection, isec`	$rs \mapsto r \cap s$

For complex operations a Lisp-style prefix syntax can be used, i.e., nested expression enclosed by parantheses, e.g., `(composition rel1 (inverse (homing rel2)))` computes rel1$\circ$ inv(hom(rel2))

`analyze-calculus c op`

Determines relation-algebraic properties

`c` calculus

`op` operation:

`analyze-calculus c test-algebra`	checks which axioms are met by the algebraic structure of the calculus
`analyze-calculus c test-property prop`	checks whether axiom `prop` is satisfied by the calculus

`qualify c opt scene`

Determines qualitative configuration from quantitative scene description.

`c` calculus

`opt` either `first2all` or `all`, sets which objects to relate

`scene` list of lists containing objects id and coordinates, e.g. `((Point-A 12.2 34.8) (Point-B -2.3 28.8))`

`constraint-reasoning c op conf1 [conf2]`

Performs operation `op` on qualitative configuration with respect to calculus `c`:

consistency checking:	
`algebraic-closure`, `a-closure`, `path-consistency`	Enforces path-consistency — since this is a purely syntactical operation on the level of qualitative relations the term “algebraic closure” is more adequate. However, since “path-consistency” is widely used in this meaning, this name is supported too.
`scenario-consistency`	Computes consistent networks containing base-relations only
`ternary-closure`	Computes algebraically closed networks using ternary composition with ternary calculi
manipulating constraint networks:	
`refine`	Merges two networks by intersecting corresponding constraints

`extend`	Merges two networks by uniting corresponding constraints
`update`	Merges two networks by overwriting corresponding constraints

`a-reasoning c cmd args`

Algebraic reasoning commands (see Sec. 3.7 starting on page 28), `c` designates calculus to use, command `cmd` and arguments are `args`:

command	**arguments**	**decription**
`consistency`	constraint-nework	tests network for satisfiability, answers: `SATISFIABLE.` network proven to be consistent; `NOT SATISFIABLE.` network proven to be inconsistent; `CANNOT DECIDE.` neither of the above
`analyze-operation`	operation	Verifies operation table; operation is e.g., `composition`, `converse`, `inverse`, `shortcut`, …
`qualify`	scene opt	qualification, arguments as for the qualify model (see above) but based purely on the algebraic specification

`export c format`

Exports calculus definition of `c` in `format`:

`qat` XML specification for QAT toolkit

`gqr` specification for GQR constraint reasoner

B.2. Interactive Mode

Using SparQin the interactive mode (i.e., invoking it with the “-i” command line option) some additional commands are available:

command	description
`quit`	exits SparQ
`help`	prints short help message
`load-calculus CALC`	loads a specified calculus into memory
`*`	used as calculus specifier in commands, `*` stands for the calculus recently loaded into memory. This avoids overhead of reloading a calculus

B.3. List of Calculi

calculus identifier(s)	calculus	section	page
`allen, aia, ia`	Allen's interval algebra (Allen, 1983)	A.1	54
`block-algebra, ba`	2D block algebra (Güsgen, 1989)	A.2	55
`cardir`	Cardinal direction calculus (Ligozat, 1998)	A.3	56
`depcalc, dep`	Dependency calculus (Ragni and Scivos, 2005)	A.5	59
`dipole-coarse, dra-24`	Dipole calculus (Moratz et al., 2000)	A.11	67
`double-cross, dcc`	Double cross calculus (Freksa, 1992a) using the original tuple naming scheme	A.7	61
`alternative-double-cross, adcc`	Double cross calculus (Freksa, 1992a) using the alternative single number naming scheme	A.7	61
`flipflop, ffc, ff`	FlipFlop calculus (Ligozat, 1993)	A.8	63
`geomori, ori, align`	Geometric Orientation calculus	A.10	66
`point-calculus, pc, point-algebra, pa`	Point algebra (Vilain et al., 1989)	A.13	71

`rcc-5`	Region connection calculus (RCC-5) (Randell et al., 1992)	A.4	58
`rcc-8`	Region connection calculus (RCC-8) (Randell et al., 1992)	A.4	57
`reldistcalculus`	Exemplary calculus from this manual	3.13.1	36
`single-cross, scc`	Single cross calculus (Freksa, 1992a)	A.6	60
`opra-`	Oriented point reasoning algebra ($\mathcal{OPRA}_m$)(Moratz, 2006)	A.12	69
`qtc-b11`	Qualitative trajectory calculus in 1D with distance (van de Weghe, 2004)	A.14	72
`qtc-b12`	Qualitative trajectory calculus in 1D with velocity (van de Weghe, 2004)	A.14	73
`qtc-b21`	Qualitative trajectory calculus in 2D with distance (van de Weghe, 2004)	A.14	74
`qtc-b22`	Qualitative trajectory calculus in 2D with distance and velocity (van de Weghe, 2004)	A.14	74
`qtc-c21`	Qualitative trajectory calculus in 2D with distance and side (van de Weghe, 2004)	A.14	75
`qtc-c22`	Qualitative trajectory calculus in 2D with distance, side, and velocity (van de Weghe, 2004)	A.14	76