GQR – A Fast Reasoner for Binary Qualitative Constraint Calculi

Zeno Gantner
Information Systems and Machine Learning Lab,
University of Hildesheim,
Samelsonplatz 1, 31141 Hildesheim, Germany
gantner@ismll.uni-hildesheim.de

Matthias Westphal and **Stefan Wölfl**
Department of Computer Science,
University of Freiburg,
Georges-Köhler-Allee, 79110 Freiburg, Germany
{westpham, woelfl}@informatik.uni-freiburg.de

Abstract

GQR (Generic Qualitative Reasoner) is a solver for binary qualitative constraint networks. GQR takes a calculus description and one or more constraint networks as input, and tries to solve the networks using the path consistency method and (heuristic) backtracking. In contrast to specialized reasoners, it offers reasoning services for different qualitative calculi, which means that these calculi are not hard-coded into the reasoner. Currently, GQR supports arbitrary binary constraint calculi developed for spatial and temporal reasoning, such as calculi from the RCC family, the intersection calculi, Allen's interval algebra, cardinal direction calculi, and calculi from the OPRA family. New calculi can be added to the system by specifications in a simple text format or in an XML file format. The tool is designed and implemented with genericity and extensibility in mind, while preserving efficiency and scalability. The user can choose between different data structures and heuristics, and new ones can be easily added to the object-oriented framework. GQR is free software distributed under the terms of the GNU General Public License.

Introduction

Qualitative constraint calculi are representation formalisms for efficient reasoning about continuous aspects of the world. Contrary to numerical or quantitative formalisms, which often rely on undecidable formal systems, qualitative calculi provide an abstraction layer over metrical data, which can be applied for developing efficient reasoning methods. In the past 25 years a huge list of such calculi has been discussed in the literature (see, e.g., Cohn and Hazarika 2001). Examples include the point algebra (Vilain and Kautz 1986) and Allen's interval algebra (Allen 1983), the various region connection calculi (Randell, Cui, and Cohn 1992; Düntsch, Wang, and McCloskey 1999), the intersection calculi (Egenhofer 1991; Egenhofer and Franzosa 1991), cardinal direction calculi (Frank 1991; Skiadopoulos and Koubarakis 2004), the double cross calculus (Freksa 1992), the OPRA calculi (Moratz, Dylla, and Frommberger 2005), and many more. Qualitative calculi have been applied in the modeling of natural languages, for representing spatial or temporal aspects in human-machine interaction (e.g., Moratz et al. 2003), in high-level vision systems (e.g., Bennett et al. 2004), and in high-level agent control applications (e.g., Dylla and Moratz 2004; Dylla et al. 2007). Qualitative calculi can also be used in query rewriting and data integrity checks in Geographic Information Systems (GIS) or as an add-on to planning systems to restrict possible state transitions by temporal or spatial constraints.

GQR (Generic Qualitative Reasoner), developed at the University of Freiburg, is a solver for binary qualitative constraint networks. GQR takes a calculus description and one or more constraint networks as input, and tries to solve the networks using the path consistency method and (heuristic) backtracking. In contrast to specialized reasoners, it offers reasoning services for different qualitative calculi, which means that these calculi are not hard-coded into the reasoner. Currently, GQR supports arbitrary binary constraint calculi; new calculi can be added to the system by specifications in a simple text format or in an XML file format.

The remainder of this article is structured as follows. In the next section, we will discuss related approaches to generic qualitative reasoning. Then, we will elaborate on how qualitative reasoning with GQR works on a conceptual level. After that, we give some details on the way the reasoner has been actually implemented. Next, we compare GQR's runtime behaviour to that of calculus-specific solvers. We conclude the article with an outlook on future enhancements to the system.

Related Work

When the development of GQR started, there were only reasoners for particular calculi (van Beek and Manchak 1996; Nebel 1996; Renz and Nebel 1998). In order to reason in a new calculus, one had to develop a new program, modify an existing one for a similar calculus, or encode it in a more generic logic — usually first-order logic, but also propositional modal logic (Bennett 1994) — for which solvers exist.

In the meanwhile there exist similar research efforts. The qualitative algebra toolkit (QAT) (Condotta, Saade, and Ligozat 2006) is a Java implementation of tools and libraries for qualitative calculi and constraint networks developed at the Université d'Artois. It features XML-based descriptions of *n*-ary calculi, a tool to define a calculus as a Cartesian product of existing algebras, as well as methods for generating random constraint networks. An included solver provides various reasoning algorithms for the consistency and minimal network problem, such as backtracking search using different heuristics and local constraint propagation methods.

SparQ (Dylla et al. 2006; Wallgrün et al. 2006) is developed at the University of Bremen. It provides support for binary and ternary calculi, transformations from quantitative descriptions into qualitative representations, and constraint based reasoning methods, such as the path consistency algorithm and backtracking search. SparQ is written in LISP with libraries implemented in C.

In contrast to those tools, the main focus in the design of GQR has been to implement a *fast* and *extensible* generic solver, which preserves the efficiency of calculus-specific solvers as much as possible.

Qualitative Reasoning with GQR

Reasoning in GQR is based on a purely syntactical definition of qualitative calculi (Ligozat and Renz 2004; Wölfl, Mossakowski, and Schröder 2007). A *(binary) qualitative calculus* is defined by a non-empty finite set B of symbols (elements of B are referred to as *base relations*), a unary function $\smile : B \longrightarrow B$ (assigning to each base relation its converse), a binary function $\circ : B \times B \longrightarrow 2^B$ (assigning to each pair of base relations their composition), and a distinguished element $\mathrm{id} \in B$ (the *identity relation*) such that some minimal requirements are met (e. g., $(a^\smile)^\smile = a$, $\mathrm{id} \circ a = a \circ \mathrm{id} = a$; etc.). Given a qualitative calculus in this sense, the set 2^B is a Boolean algebra (its elements are referred to as *relations*[1]). Moreover, a non-associative relation algebra is defined if the functions $\smile$ and $\circ$ are extended to functions $\smile : 2^B \longrightarrow 2^B$ and $\circ : 2^B \times 2^B \longrightarrow 2^B$ via the settings: $r^\smile := \{ b^\smile : b \in r \}$ and $r \circ r' := \bigcup_{b \in r, b' \in r'} b \circ b'$.

Reasoning problems in qualitative calculi are usually formulated as so-called constraint satisfaction problems. The instances of these problems can be described as *constraint networks*, which are directed finite graphs, where each edge is labeled by a relation of the calculus (representing the *constraint relation* between the connected nodes). In GQR these graphs are represented as *adjacency* matrices, where each entry is a bit vector used to encode the calculus relation between two nodes in the graph. The constraint satisfaction problem, then, is to determine for a constraint network, whether there exists an assignment to its variables/nodes in a given domain D such that all constraints of the network become true (based on a fixed interpretation of the relational symbols on this domain). Further typical reasoning tasks are to check that some constraint is entailed by a constraint network and to compute an equivalent minimal constraint network. All these reasoning tasks can be shown to be equivalent under polynomial Turing reductions.

A crucial aspect in qualitative reasoning is the fact that the underlying models usually are infinite. Hence, in order to test satisfiability of constraint networks, it is not feasible to enumerate all possible assignments to variables in a model until one finds a satisfying assignment. For this reason other techniques based on algebraic and semantic properties of the calculus must be applied for testing satisfiability. In particular, the path consistency algorithm manipulates a given constraint network by successively refining the labels $r_{x,y}$ (on the edge from node x to node y) via the operation $r_{x,y} \longleftarrow r_{x,y} \cap (r_{x,z} \circ r_{z,y})$, where z is any third variable occurring in the network. In GQR Mackworth's variant of the path consistency algorithm is implemented (Mackworth 1977; Dechter 2003), which runs in cubic time in the size of the constraint network.

[1] Sets of base relations are read disjunctively and express imprecise knowledge about the actual configuration.

Calculus	**Identifier**	$\lvert B \rvert$
point algebra	point	3
Allen's interval algebra	allen	13
RCC-5	rcc5	5
RCC-8	rcc8	8
OPRA-4	opra4	272
Cardinal directions	cd	9

Table 1: Some of the calculi defined for GQR

Since, in general, the path consistency method is not sufficient to decide consistency of constraint networks, GQR uses chronological backtracking, such trying out different instantiations of the constraints containing disjunctions of base relations (cf. Allen 1983; Ladkin and Reinefeld 1997; Nebel 1996; Renz and Nebel 1998; van Beek and Manchak 1996). Moreover, by using known tractable subclasses of a calculus (i.e., sets of relations closed under intersection and composition, for which the path consistency method decides consistency), one can speed up the reasoning time: instead of splitting a constraint during backtracking into base relations, one can split it into relations belonging to a tractable subclass, which reduces the branching factor of the search tree considerably (Nebel 1996).

Both the path consistency method and the chronological backtracking search may benefit from heuristics about which part of the constraint network is to be processed next. Currently, the weight and the cardinality heuristics (van Beek and Manchak 1996) are implemented for the path consistency method. For the backtracking search, the cardinality heuristic for variable selection (van Beek and Manchak 1996) is implemented.

GQR is written in C++. Because of its object-oriented design, users may add their own heuristics quite easily as well. New qualitative calculi are defined by writing simple text or XML files, which are then read and processed by the reasoner. Table 1 gives an overview of calculi that are already included with GQR.

Also, GQR allows for checking algebraic properties of specified calculi (as mentioned before) and it is possible to precompute (parts of) the complete composition tables of the calculus (which is reasonable for small calculi only).[2]

[2] For an overview of methods to speed up the composition operation see (Ladkin and Reinefeld 1997).

Finally, it should no go unmentionend that GQR has already been used successfully (a) in a high-level agent control system implementing rule-compliant behavior of agents in sea navigation (Dylla et al. 2007) (where reasoning in a large constraint calculus such as OPRA-4 turned out to be crucial) and (b) for the evaluation of different algorithms for application-specific customizations of qualitative calculi (Renz and Schmid 2007).

Implementation Details

In this section, we give an overview of the implementation of algorithms and data structures used in GQR.

Principal algorithms used in GQR

As pointed out in the previous section, the path consistency algorithm is one of the central methods to solve constraint networks in qualitative calculi. A naïve implementation of this procedure needs $O(n^5)$ intersections and compositions. Mackworth (1977) suggested a method that puts the paths that may be affected by changes into a queue, and processes the queue until it is empty. It runs in $O(n^3)$ time and uses $O(n^2)$ of memory. Algorithm 1 shows the method.

Algorithm 1 Path-Consistency(V,C)

```
Input: A constraint network $(V,C)$
Output: A constraint network $(V,C')$ that is a (sometimes
    even path-consistent) refinement of $(V,C)$
 1: $Q \leftarrow \{(i,j) | 1 \leq i < j \leq n\}$ // Initialize the queue
 2: while $Q$ is not empty do
 3:   select and delete an $(i,j)$ from $Q$
 4:   for $k \leftarrow 1$ to $n, k \neq i$ and $k \neq j$ do
 5:     $t \leftarrow C_{ik} \cap (C_{ij} \circ C_{jk})$
 6:     if $t \neq C_{ik}$ then
 7:       $C_{ik} \leftarrow t$
 8:       $C_{ki} \leftarrow t^{\smile}$
 9:       $Q \leftarrow Q \cup \{(i,k)\}$
10:     end if
11:     $t \leftarrow C_{kj} \cap (C_{ki} \circ C_{ij})$
12:     if $t \neq C_{kj}$ then
13:       $C_{kj} \leftarrow t$
14:       $C_{jk} \leftarrow t^{\smile}$
15:       $Q \leftarrow Q \cup \{(k,j)\}$
16:     end if
17:   end for
18: end while
19: return $(V,c)$
```

Depending on semantic properties of the calculus at hand, the output of the algorithm has to be interpreted in different ways. For many calculi discussed in the literature, the path consistency method returns a network that is (semantically) equivalent to the original one, sometimes the network is even (semantically) path-consistent (for a more detailed discussion see Ligozat and Renz 2004). For calculi with a weakly correct composition table, a network cannot be consistent if one of the edges in the network resulting from the path consistency algorithm contains the empty relation.

A main aspect in the development of new binary calculi is to identify so-called *tractable* subclasses (of the set 2^B of all relations): the consistency of networks containing only relations from such a class is decidable in polynomial time via the path consistency algorithm. For some subclasses the network resulting from the path consistency method is even minimally equivalent.

If the result of the path consistency method is a non-atomic network, which is normally the case, we must resort to trying out different instantiations of the constraints containing disjunctions of base relations (cf. Allen 1983; Ladkin and Reinefeld 1997; Nebel 1996; Renz and Nebel 1998; van Beek and Manchak 1996). Algorithm 2 works as follows: If our search ran into a dead end, we proceed by looking at the next possible instantiation of the most recently changed constraint. Such an approach is called *chronological backtracking*. Before each instantiation, the path consistency method is applied in order to prune the search tree, i. e., we avoid instantiations that will lead to inconsistencies anyway. Except for the first step, the path consistency method only has to be run for the paths that are possibly affected by the prior instantiation, which takes $O(n^2)$ intersections and compositions (this detail is not included in the listing of Algorithm 2).

Algorithm 2 Consistent(V,C)

```
Input: A constraint network $(V,C)$
Output: A Boolean value that is true if and only if $(V,C)$ is
    satisfiable
 1: Path-Consistency$(V,C)$
 2: if C contains the empty relation then
 3:   return false
 4: end if
 5: if there are non-basic edges then
 6:   Pick such an edge $e = (i,j)$
 7:   for all base relations $b$ in the label of $e$ do
 8:     $C_{ij} \leftarrow b$
 9:     if Consistent$(V,C)$ then
10:       return true
11:     end if
12:   end for
13: end if
14: return false
```

Using tractable subclasses of a calculus can speed up the backtracking search. First, a constraint network containing only relations from the subclass can be solved in polynomial time by enforcing path consistency. Second, if the network contains relations that do not belong to such a subclass, subclasses can still be used to reduce the branching factor during backtracking. Instead of splitting a constraint into its base relations, it may be split into relations belonging to the subclass. In (Nebel 1996) and (Renz and Nebel 1998), this strategy was applied to the interval algebra and to the RCC-8 calculus, respectively. Note that those subclasses are not hard-coded into the program; they can be defined as simple text files, without any programming.

Data structures for the path consistency method

Van Beek and Manchak (1996) implemented a very efficient queue data structure, which is also used in the software used by Nebel et al. (Nebel 1996; Renz and Nebel 1998) and in the work presented here. Compared to a data structure put together from generic components (parts of the C++ *Standard Template Library STL*), there is a huge performance gain, as can be seen in Fig 1. In GQR, the queue is implemented as a virtual class, allowing for easy and convenient extensions.

Computations on Relations

Relations in GQR are stored as bit vectors. There is a special bit vector class based on templates, that allows for compact representations in memory and efficient operations on the vectors. Further, the class provides an order on bit vectors, as well as a hash function. The extensive use of templates allows the reasoner to be recompiled from the same source to support any given calculus size, without compromising the speed when working with smaller calculi.

As previously mentioned, GQR allows for precomputations of composition and converse tables. It supports both full and partial precomputations, where partial precomputations are available in two different ways: Firstly, it implements Hogge's method (Ladkin and Reinefeld 1997), which splits each relations into two (nearly equally sized) parts, the higher bits and the lower bits. For compositions four tables are precomputed, containing the results for compositions of relations with only higher bits, results for those with only lower bits, and two for lower bit relations with higher bit relations and vice versa. The result of a composition with arbitrary relations can then be assembled from the tables using AND, shift, and OR operations. A similar strategy is applied to converse computations. This reduces the memory requirements significantly, since for a calculus with n base relations only six tables with at most 2^{n+1} entries have to be stored.

Secondly, GQR provides caching for both composition and converse results, using a hash table. This is also discussed in (Ladkin and Reinefeld 1997), and performs really well in nearly all situations. Since the number of entries in the hash table is not directly dependent on the number of base relations, caching is especially interesting for large calculi, where full precomputations are impossible and even Hogge's method is not directly applicable. Also, as the path-consistency algorithm is a fixed point algorithm, and therefore the actual compositions that have to be calculated are usually clustered, even small hash tables are already useful,

GQR automatically decides which method to use, according to given memory limits for composition and converses. It is also possible to add new precomputation methods for specific calculi.

Comparing GQR with a Non-Generic Reasoner

As a benchmark for the implementation of GQR, we compared the performance of GQR with that of two solvers for constraint networks in Allen's interval algebra (Nebel 1996) and RCC-8 (Renz and Nebel 1998). Those two solvers are implemented very efficiently. In fact, they are faster than GQR, because they are written in C and thus do not use certain features of C++, like virtual methods, which are convenient but affect the execution speed. Moreover, these calculus-specific reasoners can exploit certain properties of the underlying calculus (e.g., the interval algebra) to improve the reasoning performance. However, we think that, from an architectural point of view, it is worth trading some speed for flexibility and genericity.

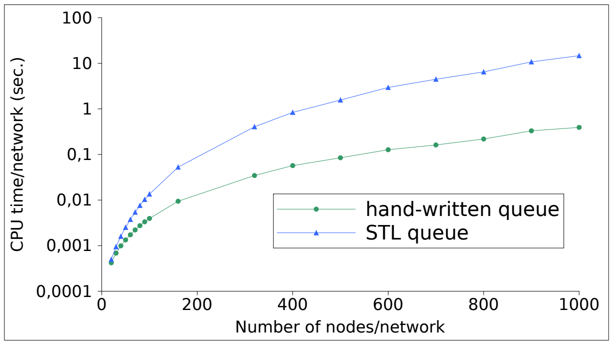

Figure 1: Performance comparison between two queue types for RCC-8

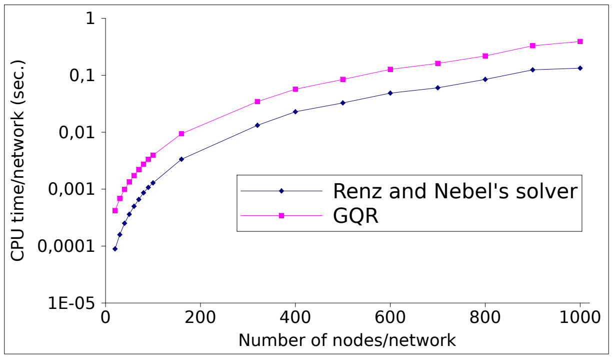

Figure 2: Performance of the path consistency method for RCC-8

The benchmark tests were carried out on a computer with an Intel Core2 processor with a CPU frequency of 2.4 GHz and 4 GB RAM. It was running Linux 2.6.18, the code was compiled with gcc/g++ 4.1.2. Only one of the CPU cores was used for the experiments. The compiler flags were `-DNDEBUG -O3` in both cases. The size of the relation data type was set to 32 bit.

Path consistency. The performance of the path consistency algorithm is crucial for the overall performance of a qualitative reasoner. Therefore, we compared GQR's path consistency implementation to that of Renz and Nebel's RCC-8 solver. The results of the path consistency benchmark are shown in Figure 2. The parameters for the constraint networks used here and in the data structure comparison (Figure 1, see the previous section on data structures)

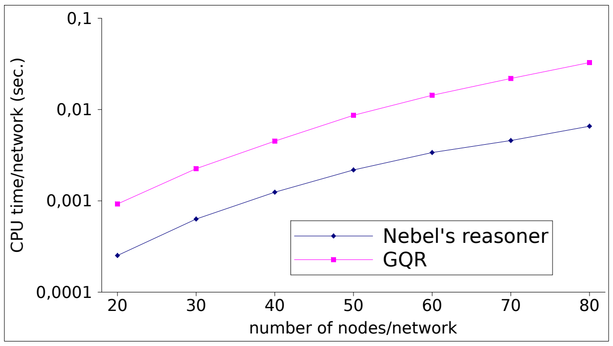

Figure 3: Performance for easy IA networks

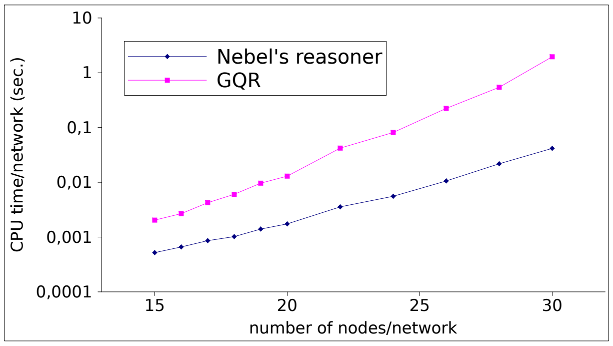

Figure 4: Performance for hard IA networks

were the same: As can be seen from the graphs, the network sizes considered were between 20 and 1000 nodes. For each size, networks with an average degree (the number of non-universal constraints of a node) of 8, 9, 9.5, 10.0, 10.5, 11, 12, and 13 were generated. The problem set for every size-degree combination consisted of between 2000 (size 20) and 50 (size 320) networks. For the sizes 500 and 1000, 10 networks were generated for every average degree. Renz and Nebel's reasoner ran three (large networks) up to four times faster than GQR. While GQR is clearly slower than the other solver, it appears to have the same scaling behavior. It is worth mentioning that the longer runtimes for small networks can be caused by the overhead of reading the constraint networks from disk, and not by the implementation of the algorithm itself.

Global Consistency. To assess the speed of the backtracking search, two different scenarios involving Allen's interval algebra were considered. First, we looked at a class of constraint networks that are relatively easy to solve. Such networks contain random relations which are representable as first-order Horn formulae, that is, the relations of the ORD-Horn subclass. The network sizes were between 10 and 80 (step size 10), and the average degrees between 3 and 12 (step size 1); for every size-degree combination, 500 networks were generated.

Second, we ran a benchmark on particularly hard instances, where the constraints were picked from a set of relations which can be represented in 3-CNF when transformed into first-order formulae. Additionally, the average degree of nodes was chosen in a way that the resulting networks belong/are close to the so-called phase transition region, in which it is about equally probable that a network is satisfiable or not. Problems in that region are known to be hard to solve (Nebel 1996). For the sizes 15 to 20, the degree was between 10.8 and 11.3, for sizes from 22 to 30, it was between 11.4 and 11.8. 500 problem instances were generated for each network size.

The results can be seen in Figure 3 and 4. While for the easy instances, the results are consistent to our path consistency results, GQR's performance on larger instances of the hard class of problems is still not satisfying. This is an issue we are currently addressing.

License and Availability

GQR is freely usable and distributable under the terms of the GNU General Public License. The software can be downloaded from `https://sfbtr8.informatik.uni-freiburg.de/R4LogoSpace/Resources/GQR`.

Conclusion and Future Work

In this article, we presented GQR, a generic reasoner for qualitative calculi. We described its underlying reasoning methods and their implementation, and presented preliminary benchmark results.

We are currently working on the next generation of GQR, which will allow for processing ternary constraint calculi as well as for handling constraint networks containing relations from different calculi. We will also implement ways to build new calculi as combinations of existing calculi and a module that automatically computes the weights used in the heuristics. Furthermore, we plan to compare the performance of GQR to that of other reasoners in more detail, especially with respect to larger calculi.

Acknowledgements

This work was partially supported by Deutsche Forschungsgemeinschaft (DFG) as part of the Transregional Collaborative Research Center *SFB/TR 8 Spatial Cognition*. We would like to thank Bernhard Nebel and Peter van Beek for allowing us to use and release their implementation of the weighted path queue used in the path consistency algorithm. We are grateful for the support by Malte Helmert, who helped us in the early stages of GQR's development, and Micha Altmeyer, who implemented Python bindings, a TCP/IP interface as well as a graphical user interface for GQR.

References

Allen, J. F. 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM* 26(11):832–843.

Bennett, B.; Magee, D. R.; Cohn, A. G.; and Hogg, D. C. 2004. Using spatio-temporal continuity constraints to enhance visual tracking of moving objects. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004*, 922–926. IOS Press.

Bennett, B. 1994. Spatial reasoning with propositional logics. In *KR'94: Principles of Knowledge Representation and Reasoning*. San Francisco, California: Morgan Kaufmann. 51–62.

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

Condotta, J.-F.; Saade, M.; and Ligozat, G. 2006. A generic toolkit for n-ary qualitative temporal and spatial calculi. In *Thirteenth International Symposium on Temporal Representation and Reasoning (TIME'06)*, 78–86. Los Alamitos, CA, USA: IEEE Computer Society.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Dylla, F., and Moratz, R. 2004. Exploiting qualitative spatial neighborhoods in the situation calculus. In Freksa, C.; Knauff, M.; Krieg-Brückner, B.; Nebel, B.; and Barkowsky, T., eds., *Spatial Cognition IV*, LNCS 3343, 304–322. Springer.

Dylla, F.; Frommberger, L.; Wallgrün, J. O.; and Wolter, D. 2006. SparQ: A toolbox for qualitative spatial representation and reasoning. In *Qualitative Constraint Calculi: Application and Integration, Workshop at KI 2006*, 79–90.

Dylla, F.; Frommberger, L.; Wallgrün, J. O.; Wolter, D.; Nebel, B.; and Wölfl, S. 2007. SailAway: Formalizing navigation rules. In *Proceedings of the Artificial and Ambient Intelligence Symposium on Spatial Reasoning and Communication, AISB'07*.

Düntsch, I.; Wang, H.; and McCloskey, S. 1999. Relation algebras in qualitative spatial reasoning. *Fundamenta Informaticae* 39(3):229–249.

Egenhofer, M. J., and Franzosa, R. D. 1991. Point set topological relations. *International Journal of Geographical Information Systems* 5:161–174.

Egenhofer, M. J. 1991. Reasoning about binary topological relations. In *Proceedings of the Second Symposium on Large Spatial Databases, SSD'91*, LNCS 525, 143–160. Springer.

Frank, A. U. 1991. Qualitative spatial reasoning with cardinal directions. In *Proceedings of the Seventh Austrian Conference on Artificial Intelligence*, Informatik-Fachberichte 287, 157–167. Springer.

Freksa, C. 1992. Using orientation information for qualitative spatial reasoning. In *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space, International Conference GIS*, LNCS 639, 162–178. Springer.

Ladkin, P. B., and Reinefeld, A. 1997. Fast algebraic methods for interval constraint problems. *Annals of Mathematics and Artificial Intelligence* 19(3-4):383–411.

Ligozat, G., and Renz, J. 2004. What is a qualitative calculus? A general framework. In *PRICAI 2004: Trends in Artificial Intelligence, 8th Pacific Rim International Conference on Artificial Intelligence, Proceedings*, LNCS 3157, 53–64. Springer.

Mackworth, A. K. 1977. Consistency in networks of relations. *Artificial Intelligence* 8(1):99–118.

Moratz, R.; Tenbrink, T.; Bateman, J. A.; and Fischer, K. 2003. Spatial knowledge representation for human-robot interaction. In *Spatial Cognition III*, LNCS 2685, 263–286. Springer.

Moratz, R.; Dylla, F.; and Frommberger, L. 2005. A relative orientation algebra with adjustable granularity. In *Proceedings of the Workshop on Agents in Real-Time and Dynamic Environments (IJCAI 05)*.

Nebel, B. 1996. Solving hard qualitative temporal reasoning problems: Evaluating the efficiency of using the ORD-Horn class. In *12th European Conference on Artificial Intelligence, Proceedings*, 38–42. John Wiley and Sons, Chichester.

Randell, D. A.; Cui, Z.; and Cohn, A. G. 1992. A spatial logic based on regions and connection. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 3rd International Conference (KR-92)*, 165–176. Morgan Kaufmann.

Renz, J., and Nebel, B. 1998. Efficient methods for qualitative spatial reasoning. In *Proceedings of the 13th European Conference on Artificial Intelligence*, 562–566. John Wiley & Sons.

Renz, J., and Schmid, F. 2007. Customizing qualitative spatial and temporal calculi. In Orgun, M. A., and Thornton, J., eds., *Australian Conference on Artificial Intelligence*, LNCS 4830, 293–304. Springer.

Skiadopoulos, S., and Koubarakis, M. 2004. Composing cardinal direction relations. *Artifical Intelligence* 152(2):143–171.

van Beek, P., and Manchak, D. W. 1996. The design and experimental analysis of algorithms for temporal reasoning. *Journal of Artificial Intelligence Research* 4:1–18.

Vilain, M. B., and Kautz, H. A. 1986. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the 5th National Conference on Artificial Intelligence*. Morgan Kaufmann. 377–382.

Wallgrün, J. O.; Frommberger, L.; Wolter, D.; Dylla, F.; and Freksa, C. 2006. Qualitative spatial representation and reasoning in the SparQ-toolbox. In Barkowsky, T.; Knauff, M.; Ligozat, G.; and Montello, D. R., eds., *Spatial Cognition V*, LNCS 4387, 39–58. Springer.

Wölfl, S.; Mossakowski, T.; and Schröder, L. 2007. Qualitative constraint calculi: Heterogeneous verification of composition tables. In Wilson, D., and Sutcliffe, G., eds., *Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference (FLAIRS 21)*, 665–670. AAAI Press.