Unit Testing for Qualitative Spatial and Temporal Reasoning

Carl Schultz[1], **Robert Amor**[1], **Hans Guesgen**[2]

[1] Department of Computer Science, The University of Auckland
Private Bag 92019, Auckland, New Zealand
[2] Institute of Information Sciences and Technology, Massey University
Private Bag 11222, Palmerston North, New Zealand

Abstract

Commonsense reasoning, in particular qualitative spatial and temporal reasoning (QSTR), provides flexible and intuitive methods for reasoning about vague and uncertain information including spatial orientation, topology and proximity. Despite a number of theoretical advances in QSTR, there are relatively few applications that employ these methods. The central problem is a significant lack of application-level standards and validation methods for supporting developers in adapting and integrating QSTR with their domain specific qualitative spatial and temporal models. To address this we present a significantly novel methodology for QSTR application validation, inspired by research in software engineering. In this paper we focus on unit testing, and adapt the software engineering strategy of defining boundary cases. We present two critical boundary concepts, a methodology for isolating the units under testing from other parts of the model, and methods to assist the designer in integrating our critical boundary unit testing approach with a broader validation plan.

Introduction

Intuitive commonsense reasoning is a prominent area in artificial intelligence research (Hayes 1979; Kuipers 1994). An important subdiscipline that specialises in qualitative spatial and temporal reasoning (QSTR) has also gained a lot of attention in the AI community (Cohn and Renz 2007). Over the last 25 years, AI researchers have developed a variety of sophisticated QSTR methods that provide flexible querying and reasoning for vague and uncertain spatial and temporal data. A seminal example of QSTR is Allen's interval calculus (Allen 1983) which defines a set of thirteen atomic relations between time intervals, and an algorithm for reasoning about networks of temporal relations, e.g. (where • is a composition operator)

t_1 before t_2 • t_2 contains t_3 = t_1 before t_3
t_1 overlaps t_2 • t_2 during t_3 =
t_1 (overlaps, or during, or starts) t_3

Despite significant theoretical advances, there is a distinct absence of applications that make use of these techniques (Dylla et al. 2006). The central problem is the significant lack of support for adapting and integrating existing QSTR methods with other domain specific qualitative spatial and temporal models. Because there are no application-level standards, metrics or validation methods, QSTR systems are developed in a very ad hoc manner and it is impossible to make an effective assessment of a QSTR system's validity; in fact, this is a systemic issue that has led to erroneous QSTR formalisms, e.g. refer to page 666, first column in (Wölfl et al. 2007). This problem of validation clearly presents a significant barrier for the development of QSTR applications.

Researchers in QSTR typically apply general first-order theorem provers (and higher) for system validation (Wölfl et al. 2007). However, the use of theorem provers for application level validation is not practical in general. Firstly, applying theorem provers can be very manually intensive, and even expert logicians in the QSTR research field find the task non-trivial (e.g. refer to page 292 and Section 6.2 in (Cohn et al. 1997)). Secondly, they require axioms for the logic which in many cases will not be available, making theorem provers impossible to use. For example, particularly during the early stages of application design, software developers may need to rapidly encode informal qualitative domain knowledge with the intention of refining the logic later if necessary. Thus, a thorough axiomatisation would not be necessary or appropriate.

We present a significantly different methodology for QSTR application validation, inspired by research in software engineering. In this paper we focus on unit testing, and adapt the software engineering strategy of defining boundary cases. Boundary scenarios partition the infinite space of testable scenarios into a manageable, finite number of equivalence classes, thus facilitating practical unit testing. An application designer can then use boundary case unit testing in conjunction with other QSTR specific validation approaches to achieve the necessary level of confidence that the QSTR application is fit for purpose. The following section provides an overview of the main concepts in QSTR and unit testing. We then present our unit testing critical boundaries with a worked example. In the subsequent section we identify an important class of unit tests that exercise the core components of a constraint that operate in isolation from the rest of the model. Finally we present methods for integrating our critical boundary unit testing approach with a broader validation plan.

Foundations of QSTR and Unit Testing

QSTR systems model, infer, and check the consistency of object relations in a scenario. Unit testing isolates and tests the system's atomic components with the aim of covering all scenarios by recognising that different scenarios can fall into the same logically equivalent category. Thus, designers achieve full coverage by exercising unit tests for at least one prototypical scenario from each category. This is analogous to standard software unit testing that covers all decisions and control paths in a unit of code (e.g. an object's method in object-oriented software) and reduces the test space with boundary checking, equivalence class partitioning and cause-effect graphs. Once unit correctness has been established, integration tests ensure that the designer's intention holds when the atomic components are combined and incorporated into a more complex model (Burnstein 2003).

A QSTR scenario consists of a finite set of objects O, and qualitative relationships between those objects. Formally, an object $o \in O$ is an atom and a qualitative relation $r \in R$ is a set of object tuples of fixed arity d, $r \subseteq O^d$. A QSTR system consists of relation types and constraints between relations that are evaluated in the scenario.

Often parts of the scenario are indefinite or unknown, and reasoning is used to resolve this ambiguity. Ambiguity about a scenario is encoded by explicitly representing *definitely holds*, *definitely not holds* and *indefinite*. A further case *not applicable* can also occur when qualities require preconditions. So for any relation type r in a QSTR system, the scenario model requires four mutually exclusive sets r^+ (definitely holds), r^- (definitely not holds), $r^?$ (indefinite), $r^{\sim}$ (not applicable) such that

$$O^d = r^+ \,\Delta\, r^- \,\Delta\, r^? \,\Delta\, r^{\sim}$$

for integer relation arity $d \geq 1$ where Δ is symmetric difference (mutual exclusion). For our purposes, we do not need to discuss the semantics of these scenarios in further detail, and instead refer the reader to (Ligozat et al. 2004).

The structure of a QSTR system is defined by constraints between the qualitative relations of the form

$$X \,\delta\, Y,$$

where $\delta \in \{\subset, \subseteq, \not\subset, \neq, =, \ldots\}$[1] is a set comparison and X, Y are set expressions to be evaluated in a scenario. Set expressions are either sets, or the result of set operations. If a scenario does not satisfy a constraint then reasoning attempts to move the offending tuples out of indefinite sets and into one of the definite sets (i.e. . If offending tuples are not indefinite then reasoning has identified a contradiction and the scenario is inconsistent.

[1] Two non-empty sets can take one of 5 mutually exclusive relationships: = , ⊂, ⊃, partial overlap, disjoint. If one or both sets are empty, there are 3 additional mutually exclusive relations: $=_\varnothing$ (both empty), $\subset_\varnothing$ (LHS empty), $\supset_\varnothing$ (RHS empty). Constraints can take any disjunction of these 8 mutually exclusive relations, e.g. $\subseteq$ is $\subset \vee =$. Thus there are $2^8 = 256$ possible comparisons.

An example of a scenario is

$O=\{kitchen, lounge, study\}$
$adjacent^+ = \{(lounge, study), (lounge, kitchen)\}$
$adjacent^? = \{(lounge, lounge), (study, lounge), \ldots\}$
$adjacent^- = \{\}$.

The *adjacent* relation can be defined as symmetric using the constraint

$$\{(x,y) \mid (y,x) \in adjacent^+\} \subset adjacent^+.$$

The LHS of the constraint as evaluated in the scenario is $\{(study,lounge), (kitchen,lounge)\}$. The RHS as evaluated in the scenario does not contain these tuples as required by the proper subset relation, and so reasoning moves the offending tuples out of $adjacent^?$ and into $adjacent^+$ thus satisfying symmetry.

A test consists of a set expression, a scenario instance, and the subset of objects expected from evaluating the set expression in the scenario instance. A test passes if the actual set equals the expected set, and fails otherwise. The problem of unit validation is to produce a set of tests that exercise classes of scenarios, chosen as appropriate indicators of whether the tested unit is functioning correctly.

Critical Boundaries for Unit Testing

Unit testing aims to validate an infinite number of scenarios by defining equivalence classes and ensuring at least one scenario from each class is tested. We define the testable units of a QSTR system as set expressions within constraints. Based on an analysis of set expressions, we establish two critical scenario boundaries that form equivalence classes.

Set expressions form new sets by collecting elements from existing sets according to specified conditions. The conditions used to build a set can be divided into two categories. For each tested element (or tuple of elements) a condition either allows the element into the new set, or rejects the element. It follows that each condition from a set expression either

- **triggers** the constraint (the constraint will not apply if these conditions are never satisfied)
- **overrides** the constraint (even if the constraint has been triggered, if these conditions hold then they will stop the constraint from applying), or
- is **independent** (has no effect).

For a constraint to apply, it must have one or more triggers that are not overridden. We use this insight to define equivalence classes based on the following simple structure for analysing QSTR constraints

```
select tuple :
          exists tuple :      trigger conditions
and not exists tuple :      override conditions
```

As a programming construct this format has a number of desirable properties. Most importantly, it is a modular construct that facilitates uniform testing by structuring the set expressions which function as units in a QSTR system. Moreover, the construct is orthogonal as conditions can themselves contain nested selections, and it supports both bottom-up and top-down design as nested selections can be either well-developed units or unspecified stubs. It must be noted that this logical structure is implementation independent, e.g. QSTR systems are traditionally implemented as constraint satisfaction problems, but the unit validation test plan for exercising the logic of the constraint can still be derived from the underlying model using our construct.

Given a set expression, our unit validation process model produces a collection of unit tests that sufficiently exercise the logic of the set expression:

1. Merge decision tables to remove invalid scenario classes (for node, arc, or *k*-path consistency).
2. Enumerate valid relation states.
3. Identify relevant scenario classes for testing (by employing critical scenario boundaries).
4. Generate prototype scenario test instances.
5. Assign the expected test results.

Step 1 is dimensionality reduction by eliminating contradictory scenarios, and is briefly discussed in a later section. Steps 2 and 4 are trivial. Step 3 is the main focus of this paper, and is discussed in the following sections. Step 5 requires the designer to specify the correct test result. Note that Steps 1 to 4 can be automated thus reducing development time, for example, in a software tool.

Scenario Equivalence Classes

We will now derive two critical boundary concepts based on equivalence assumptions (the reader may like to work through the example in the next section simultaneously). The conditions for tuple selection can be listed in two decision tables, one for the *exists* clause, and one for the *not exists* clause. Every scenario can be expressed as zero or more rows from each table, and conversely, an infinite number of scenarios exist for each combination of rows. The first equivalence assumption is that scenarios resulting in the same combination of rows in the decision tables are equivalent, and thus at least one scenario for each row combination must be tested. This places a critical boundary between a qualitative relation being empty $\neg\exists x \cdot x \in r^+$, and having one or more objects $\exists x \cdot x \in r^+$. It thus prescribes at least one test for each combination of qualitative relation states in the conditions of a set expression. This yields a finite critical test set size of 2^n where n is the number of rows.

A further useful distinction is the interaction of different condition classes, where the effect of the interaction of rows is different to the effect of the rows taken individually. Thus, the second equivalence assumption is that, if a set of scenarios each result in a single row of the same condition class (such as *trigger*) then any scenario that results in some combination of these rows is equivalent to the set of individual scenarios. For example, after testing all *trigger* rows individually we can ignore pairwise (and higher) comparisons of *trigger* rows. This is because, if two conditions correctly select an element when individually exercised (each satisfying some Boolean *trigger* condition $\exists x \cdot x \in r^+$), then there is no logical way that they can erroneously prevent that same element being selected when both conditions are true in the same scenario. Note that if two *triggers* are intended to be mutually exclusive then an *override* condition is used. By focusing on this important class of tests we further reduce the critical test set size without decreasing the effective test coverage. The critical test set size is now

$$1 + (t+o+i) + (t \cdot i + t \cdot o + o \cdot i) + (t \cdot o \cdot i) \qquad (1)$$

where t, i, o are the number of *trigger*, *independent* and *override* rows respectively, with an additional scenario that corresponds to zero rows.

This defines complete boundary case unit testing for a set expression and thus provides an ideal target for logic validation.

Worked Example

Consider the following application-specific constraint for determining apparent room colour temperature (Schultz et al. 2008), with a selection of cases illustrated in Figure 1:

"If a room has at least one warm light, and does not have lights of any other temperature, then the room has a warm colour temperature"

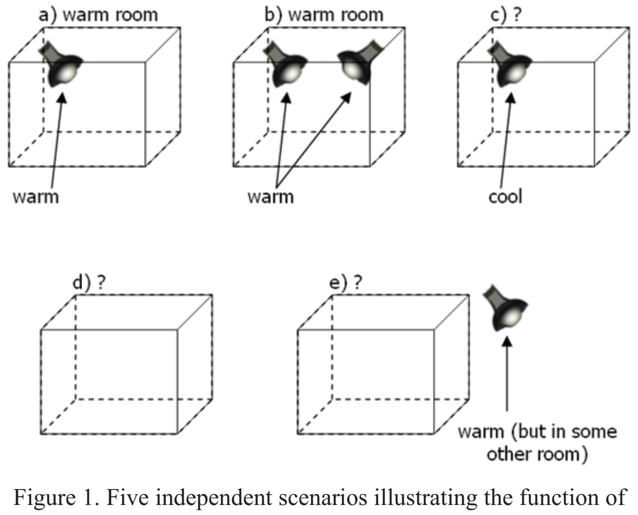

Figure 1. Five independent scenarios illustrating the function of the apparent room colour temperature constraint. Each scenario contains a room (represented by a cube) and light sources.

This constraint may then be integrated into an existing QSTR system that reasons about spatial arrangements of light sources and surfaces. One formal encoding of this constraint might be

$$\{x \mid (\exists y.\ light^{+}(y) \wedge in^{+}(y,x) \wedge warm^{+}(y)) \wedge \neg(\exists y'.\ light^{+}(y') \wedge in^{+}(y',x) \wedge warm^{-}(y'))\} \subseteq warm^{+}. \quad (2)$$

Table 1 shows the decision tables for the LHS set expression that enumerate the different relation states and the condition class of each state combination.

TABLE 1: Decision tables for conditions in the *exists* clause (upper) and *not exists* clause (lower) of the LHS set expression in constraint (2).

$y \in light$	$(y,x) \in in$	$y \in warm$	*cond. class*
holds	holds	holds	*trigger*
holds	holds	not holds	*independent*
…	…	…	…
not holds	not holds	not holds	*independent*

$y' \in light$	$(y',x) \in in$	$y' \in warm$	*cond. class*
holds	holds	holds	*independent*
holds	holds	not holds	*override*
…	…	…	…
not holds	not holds	not holds	*independent*

Not all rows between the decision tables are independent, for example, if there exists y such that $y \in warm^{+}$ then there must also exist y' such that $y' \in warm^{+}$. Thus, the decision tables are merged (Step 1) to achieve node consistent unit validation, based on the shared qualitative relations *light*, *in* and *warm*. The dimensionality is now low enough to represent the table in matrix form as shown in Table 2 (Step 2). We use $\hat{y}$ to represent both y and y' and refer to this as the constraint matrix.

TABLE 2: Three-dimensional constraint matrix for the LHS set expression in constraint (2).

$\hat{y} \in$light: holds		
$(\hat{y},x) \in in$ / $\hat{y} \in warm$	holds	not holds
holds	1) *trigger*	2) *independent*
not holds	3) *override*	4) *independent*

$\hat{y} \in$light: not holds		
$(\hat{y},x) \in in$ / $\hat{y} \in warm$	holds	not holds
holds	1) *independent*	2) *independent*
not holds	3) *independent*	4) *independent*

Every scenario can be expressed as some combination of zero or more cells in this matrix. For example, Figure 1a is cell 1, Figure 1c is cell 3 and Figure 1d is the case with zero cells.

Our first critical boundary prescribes at least one test for each combination of cells in the constraint matrix (Step 3). Applying the equation 2^n for n=8 gives a unit test set size of 256. Our second critical boundary only considers the interaction between different condition classes. Applying equation (1) for t=1, o=1, i=6 gives a unit test set size of 28, which is an order of magnitude smaller.

Deriving Constraint Intent for Effective Unit Testing

There are two problems with the given critical boundaries. Firstly, covering all the resulting unit tests may not be practical as the number of possible tests explodes with an increase in the number of conditions. Let n be the number of decision table rows, $n=t+o+i$. In the worst case[2] $t=o=i=n/3$, and the test set size from substituting into equation (1) is in the order of n^3,

$$1 + n + \frac{n^2}{3} + \frac{n^3}{27}.$$

Secondly, covering the huge number of unit tests with equal attention may obscure tests that are more logically relevant. A bias is required to determine which tests are more important than others.

To address this, we observe that constraint domains often have a number of qualitative relations that primarily support the interaction between constraints and other external modelling requirements. However, unit testing focuses on testing the constraint in isolation of other parts of the model (Burnstein 2003). Thus, our critical boundaries define a test set that is a superset of the target unit test set. For example, the domain of constraint (2) contains the relations *warm*, *light* and *in*, giving 28 unit tests after merging. However, if (2) was the only constraint in the model, then the whole system could collapse into a simple Boolean test,

$$\text{room\_is\_warm} = \exists y.warm^{+}(y) \wedge \neg\exists y'.warm^{-}(y').$$

This system only models one room in a scenario, and every defined object is a light source located in that room, yielding 5 unit tests (3 after merging). While this simplification might appear arbitrary, reasoning will arrive at the same conclusion as the more complex system about which rooms are *warm*. Any other simplification that only implicitly models *warm* will not always provide the same inference results.

Hence, the task is to identify the important components in a set expression that capture the designer's purpose or the intent of the constraint. It is desirable to have an objective definition of constraint intent, so that unit tests that exercise interesting boundary cases centred on the important components can be created in a systematic way. The problem is that a designer's intent can be both vague and subjective, thus eluding any formal, objective definition.

We propose that the intent of a constraint is reflected in the tension between the *trigger* and *override* conditions, thus interesting conditions are those that separate *triggers* and *overrides*. We refer to this concept as the principle of

[2] Equation (1) is maximised when the product of the variables is maximised. The product of two numbers that have a difference of $2j$ is $(i+j)(i-j)=i^2-j^2$. This is maximised when j=0; it follows that the product of any numbers (that must sum to n) is maximised when they are equal.

trigger-override separation. A condition implements this tension if it supports *triggers*, while not supporting *overrides*, and vice versa. A condition does not implement this tension if it supports both *triggers* and *overrides*. This is because a change in its truth value will cause the *trigger* and *override* conditions to change in step with each other, and thus will have no net effect on triggering or overriding the constraint. One very useful result of this definition is that shared conditions can be easily and objectively identified, allowing unit tests that exercise constraint intent to be discovered and generated automatically.

Following from the worked example in the previous section, we reformat the expression by placing the shared conditions in a *given* clause

$$\begin{array}{rl} \text{select R :} & \\ \text{exists Y :} & warm^{+}(\text{Y}) \\ \text{and not exists Y':} & warm^{-}(\text{Y}') \\ \text{given :} & light^{+}(\boldsymbol{Y}) \wedge in^{+}(\boldsymbol{Y},\text{R}) \ . \end{array}$$

Capitalised words are variables following Prolog syntax conventions. The semantics of this expression are equivalent to the LHS set expression in (2). That is, the scope of variables declared in either the *exists* or *not exists* clause is limited to that clause, in this case Y and Y'. The special *given* clause is evaluated in conjunction with the other two clauses where ***Y*** is interpreted as any Y variable with zero or more primes. Note that $light^{+}(\boldsymbol{Y})$ is not equivalent to $light^{+}(\text{Y}) \wedge light^{+}(\text{Y}')$ because Y' is outside the scope of the *exists* clause and Y is outside the scope of the *not exists* clause. Applying the principle of *trigger-override* separation, the refined test set contains 4 unit tests that exercise the set expression in isolation of other model components.

While constraints necessarily have *trigger* conditions, they may not have *overrides*, in which case our principle of *trigger-override* separation as a definition of constraint intent does not apply. Determining an objective definition that can apply to constraints without overrides remains an open problem and in these cases designers either must formally specify the important components or exhaustively test the constraint domain.

Incorporating Critical Boundaries into a Unit Testing Plan

Unit testing using critical boundaries is only one technique for validation and must be used as part of a suite of tools to determine that a QSTR application is fit for purpose. In this section we present methods that build on the results of set expression unit testing so that critical boundary analysis can be incorporated into a broader validation plan. The first method applies set expression results to constraint-level testing, and the second method eliminates invalid scenarios to refine unit testing.

It must be noted that our critical boundaries are relative to the set expression being tested, and not relative to the model or the requirements specification. If a condition is not present in the set expression then it will not be considered to provide a logically distinct class of scenarios and will not be tested. Thus, the given critical boundaries must be used in conjunction with black-box tests derived from the requirements specification and white-box testing methods such as test coverage and mutation testing (Schultz et al. 2008).

Unit Testing Constraints

Once the set expressions on the LHS and RHS of a constraint have been unit tested, the next step is to test the constraint directly by ensuring that the constraint's set comparator δ is valid for all logically distinct scenarios. The relevant scenarios are the cross product of the LHS and RHS set expression equivalence classes, and for each scenario the constraint will either be satisfied or contradicted. A robust validation target is to ensure that the constraint behaves correctly for all of these logically distinct scenarios. For example, constraint (2) has 28 distinct LHS scenarios and 2 distinct RHS scenarios, giving a cross product of 56 logically distinct scenarios. If the designer agrees with the constraint being satisfied or contradicted in each of these 56 scenarios, then the constraint is valid with respect to the critical boundaries of the set expressions.

Merging Decision Tables

Rows between the decision tables may not be independent, meaning that some combinations of rows represent inconsistent or invalid scenarios. These invalid classes of scenarios can be excluded from testing by merging the decision tables (Step 1 in the process model). This can dramatically reduce the size of the test set, thus making unit testing more practical and effective by limiting the focus to relevant scenarios. The three possible reasons for rejecting scenarios are

- the principle of contradiction $\forall P.\neg(P \wedge \neg P)$ where P is a proposition, e.g. this caused the decision table merge in the worked example for constraint (2),
- a second constraint between two relations, e.g. two relations might be defined as mutually exclusive such as *warm* and *cool* in some other constraint, and
- a series of k constraints that ultimately restrict two relations.

These three cases define classes of local consistency analogous to those in constraint satisfaction, namely node, arc and k-path consistency. For arc and k-path consistency to be valid, each constraint in the path must also be valid, hence only node consistency is strictly set expression unit testing. However, if a collection of constraints are taken as an atomic module then the designer can apply these local consistency criteria to merge particular decision tables and benefit from the reduced test set size with no loss in validation quality. Future work is focused on developing and analysing algorithms for automating this step, and incorporating other standard dimensionality reduction techniques.

Conclusions

This paper introduces a methodology for validating qualitative spatial and temporal reasoning (QSTR) applications, with the aim of making QSTR systems more accessible for practical applications. In particular, we present a methodology for unit testing based on critical QSTR boundary scenarios. The units of a QSTR system are defined as set expressions within constraints. To allow unit testing to cover the range of expressible scenarios we separate set expression conditions into three classes depending on whether the condition collects an element (trigger), rejects an element (override) or ignores an element (independent). The result is that all scenarios are represented by some combination of zero or more of these conditions.

We use the three condition classes as a basis for defining critical boundaries for unit testing. Our first critical boundary distinguishes between a qualitative relation being empty and non-empty, and thus prescribes at least one test for each combination of these qualitative relation states in the conditions of a set expression. Our second, more refined critical boundary ignores the interaction of conditions of the same class. That is, it emphasises scenarios where the effect of the interaction of conditions is different to the effect of the conditions taken individually. This improves the effectiveness of unit testing by refining the test set without incurring a loss in the quality of test coverage.

We identify an important class of unit tests that exercise the designer's intent in a constraint (that is, the core components of a constraint that operate in isolation from the rest of the model) by proposing that constraint intent is reflected in the separation between trigger and override conditions. This objective definition allows the class of unit tests that exercise constraint intent to be discovered and generated automatically.

Finally, we present methods to assist the designer in integrating our critical boundary unit testing approach with a broader validation plan. The first method applies set expression results to constraint-level testing, and the second method eliminates invalid scenarios to focus and refine unit testing.

Acknowledgements

This work has been funded by the Bright Future Top Achiever Doctoral Scholarship (Tertiary Education Commission, New Zealand).

References

Allen, J. F. 1983. *Maintaining knowledge about temporal intervals*. Communications of the ACM, 26, 11, pp. 832-843: ACM Press.

Burnstein I. 2003. *Practical software testing: a process oriented approach*. Springer, New York.

Cohn, A. G., Bennett, B., Gooday, J., and Gotts, N. M. 1997. *Qualitative spatial representation and reasoning with the region connection calculus*. GeoInformatica, 1, 3, pp. 275–316: Springer, Berlin.

Cohn, A. G., and Renz, J. 2007. *Qualitative spatial reasoning*. In van Harmelen F, Lifschitz V, Porter B (eds): Handbook of Knowledge Representation: Elsevier Science.

Dylla, F., Frommberger, L., Wallgrün, J. O., and Wolter, D. 2006. *SparQ: A Toolbox for Qualitative Spatial Representation and Reasoning*. In Proceedings of the Workshop on Qualitative Constraint Calculi: Application and Integration.

Hayes, P. 1979. *The Naive Physics Manifesto*. In Michie, D. (ed.) Expert Systems in the Microelectronic Age. Edinburgh: Edinburgh University Press.

Kuipers, B. 1994. *Qualitative reasoning*. MIT Press.

Ligozat, G., Mitra, D., and Condotta, J. 2004. *Spatial and temporal reasoning: beyond Allen's calculus*. AI Communications 17, 4, pp. 223–233.

Schultz, C. P. L., Amor, R., Lobb, B., and Guesgen, H. W. 2009. *Qualitative design support for engineering and architecture*. Advanced Engineering Informatics, 23, 1, pp. 68-80: Elsevier.

Wölfl, S., Mossakowski, T., and Schröder, L. 2007. *Qualitative constraint calculi: Heterogeneous verification of composition tables*. In Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2007), pp. 665-670. AAAI Press 2007.