2011

Automatic detection of inconsistencies in a conceptual graph knowledge base

Caralee Kassos

AUTOMATIC DETECTION OF INCONSISTENCIES IN A CONCEPTUAL GRAPH KNOWLEDGE BASE

by

CARALEE KASSOS

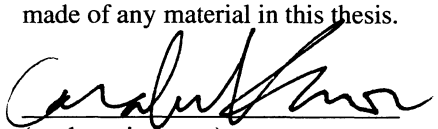
A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in
The Department of Computer Science
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2011

In presenting this thesis in partial fulfillment of the requirements for a master's degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.



(student signature)

3-14-11

(date)

THESIS APPROVAL FORM

Submitted by Caralee Kassos in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate on the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

3-14-11 Committee Chair

3/14/11

3/14/2011

Department Chair

3/14/11 College Dean

5/5/11 Graduate Dean

ABSTRACT
The School of Graduate Studies
The University of Alabama in Huntsville

Degree Master of Science College/Dept.: Science/Computer Science
Name of Candidate Caralee Kassos
Title Automatic Detection of Inconsistencies in a Conceptual Graph Knowledge Base

This thesis proposes a format for representing knowledge base constraints in conceptual graph form and an algorithm for using these constraints to validate an existing knowledge base by finding sets of nodes that are inconsistent with these constraints. The algorithm is analyzed and the performance is found to be polynomial with respect to knowledge base size and number of children each constraint node has.

Abstract Approval:

Committee Chair

Department Chair

Graduate Dean

ACKNOWLEDGMENTS

I would like to thank Dr. Harry Delugach for his guidance regarding conceptual graphs. The other members of my committee, Dr. Dan Rochowiak and Dr. Letha Etzkorn, have also been very helpful with comment and suggestions..

I would also like to thank my family for encouraging and supporting my through my work for this degree.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The purpose of this thesis is to propose a method for automatically detecting *inconsistencies* in a conceptual graph knowledge base. A knowledge base is a collection of data that is used for a specific purpose. The data in the knowledge base is used to draw inferences and perform computations. Inconsistencies in the data can lead to flawed computations. In some cases, it is possible to work around inconsistencies. In others, it is necessary to correct or remove them. In either case, the inconsistencies must be found before they can be dealt with, and therefore, it is important for a knowledge base system to have algorithms for detecting inconsistencies. For the purpose of the research described here, an inconsistency occurs when a part of the knowledge base violates the constraints set forth in its own rules. A situation in which the data in the knowledge base does not match what it is supposed to be modeling could be considered an inconsistency. This is not necessarily to say it is an inconsistency when that a knowledge base does not match reality, unless reality is what it is meant to model. For example, a knowledge base intended to model a hypothetical situation or an error case may contain completely unrealistic data, but still be consistent with respect to its own rules.

Both the constraints and the knowledge base data are represented as conceptual graphs. The structure of the constraint graphs is one of the main focuses of this paper, and is described in detail later. There are no restrictions on the structure of the data. The procedure takes a constraint, and then scans the remaining graphs for violations of this constraint.

There are many different definitions of the term *knowledge base* in use. For example, in Zhao1, the author's definition of a knowledge base is a system that deals with incomplete data, and must be able to incorporate new data or learn new facts, as it is used Zhao1. For the purposes of this thesis, a knowledge base defined as a collection of data items that is used by an automated system for a specific purpose. An example of a knowledge base would be an inventory system for a factory that kept track of what items were in stock, and what items had been ordered. Whatever the purpose of the data items, they are generally used to perform calculations, make a decision or manipulate other data. The accuracy of these computations depends on the accuracy of the data items.

In order to produce accurate results, the knowledge base and its data items must be consistent. Morell1, for example, describes an inconsistency as when the knowledge base "asserts something that is not true of the modeled domain." Liu2 considers a knowledge base inconsistent if it “contains conflicting information about the system, and/or violates predefined constraints.” This thesis focuses primarily on consistency between a knowledge base's data and its rules. For example, if the factory's inventory system lists a certain crate as being stored in two different locations, this would be an inconsistency because it is physically impossible for one object to be in two places. If some computation were to be performed using this data, for example, a query to determine which items were being stored in a particular section of the warehouse, the results may be incorrect. The results may indicate that the crate in question was stored in one section, when it is actually in another, or it may indicate that there are two of the same crate in stock. For the purpose of this thesis, an inconsistency is defined as a knowledge base subset that describes a situation that cannot exist, according to the knowledge base's rules.

Inconsistencies can be introduced into the knowledge in several ways. Data added to the knowledge base can be incorrect, or in conflict with existing data. The knowledge base rules can change, and formerly consistent data can conflict with the new rules. An example of the former situation is when two knowledge bases are merged, as in Liu1. The approach taken in Liu1 assumes one knowledge base to be consistent, and uses this as a reference. The other knowledge bases to be merged are ranked based on

their consistency with the reference knowledge base. The algorithm uses possibilistic logic to determine which parts of the merged knowledge bases are to be kept, and which discarded as inconsistent Liu1.

Once inconsistencies are found in a knowledge base, the problem becomes how to deal with them. Several approaches to dealing with known inconsistencies have been used, such as deleting items that are in conflict with others, and altering the calculations performed using the knowledge to take the inconsistencies into account.

Hunter1, for example, discusses methods for using weakly-negative logics and four-valued logics to deal with conflicting information. With a four-valued logic, statements can by marked as both True and False, or neither True nor False [Hunter1]. In a classical logic system, if a statement were considered True in one part of the knowledge base and False in another, there is no way to represent this contradicting situation. Marking is as either True or False would result in a conflict. The ability to mark a statement as both or neither makes it possible to keep track of inconsistencies and avoid conflicts without having to delete potentially inconsistent items from the knowledge base. The approach taken in this thesis uses only classical logic. This is primarily because the work here is done using conceptual graphs. Conceptual graphs are based on classical logic. If four-valued logics are implemented in conceptual graphs some time in the future, the algorithm described here can be altered to take advantage of them. Tahara1 takes a similar approach to working around inconsistencies, using a three-valued logic.

Ariel1, by contrast, is not concerned with managing the inconsistencies at all. The approach taken here involves deleting the inconsistent parts of the knowledge base and only using the remaining assertions to draw conclusions [Arieli1]. This assumes that it is possible to trace the source of an inconsistency to a specific assertion, and that deleting that assertion will not cause other inconsistencies. For example, if there are two assertions that contradict each other, while deleting one will remove that particular inconsistency, it is not always possible to know which of the two to remove. If the inconsistency is caused by the relationships between a set of assertions, not by one single assertion, removing one may not solve the

problem. It is also possible that removing an assertion will cause other inconsistencies if other parts of the knowledge base are dependent on it.

One question that may be asked is, why are constraints needed at all? Is it possible to check for inconsistencies with only the knowledge base itself? In some cases, the rules of first-order logic are enough to determine if there is an inconsistency in the data. For example, if a situation is asserted to be true in one part of the knowledge base and false in another part, this would be an inconsistency. In most cases, however, you must have some metaknowledge, or knowledge about the data in the knowledge base, in order to find errors in the knowledge base [Morell1]. You must know what you are looking for in order to find it. Constraints define the structure of data items and the relationships between them, making it possible to find parts of the knowledge base that are malformed. Some kind of quality assurance "should be an integral part of the development process" [Zlatareva1]. Constraints put limits in the ways the data in a knowledge base can be interpreted. The use of constraints in the construction of a knowledge base results in more precise modeling [Sourrouillle1].

This proposal first describes some previous related work. It will then describe the different kinds of inconsistencies that the algorithm described here can detect and provide examples of each one. Finally, it will explain how the algorithm will determine which nodes to check and how the check will occur.

CHAPTER II

BACKGROUND

Conceptual graphs are a knowledge representation system developed by Sowa [Sowa1]. This system represents knowledge as a graph consisting of nodes and connections between the nodes. The two types of nodes used by the algorithm described here are the Concept and Relation nodes. Concept nodes represent objects or attributes of objects. Relation nodes represent the relationships between Concepts. The two types of nodes are connected with arrows. The example below is a example of a graph that represents a Widget that has a color of Blue.

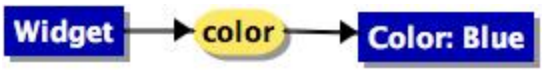


Figure 2.1 Example

Conceptual Graph

Both the Widget item and its color are represented as Concept nodes. The “color” relation node indicates that the Color node belongs to the Widget node. Without this relation, the system would know that a Widget exists and Blue exists, but these would be independent of each other. The direction of the arrow is also important. It indicates that the Widget has the color Blue, not the opposite. The relation indicated by the arrow, on the other hand, can be treated as bi-directional. The above graph could be read as "a widget with a color attribute of blue" or "a blue color that applies to a widget."

Other conceptual graph constructs used here are the context and line of identity. A context is a grouping of nodes that are treated as a single entity. It is meant to represent a situation rather than a single item. A line of identity connects two concepts that are meant to represent the same entity. This is different from an "equals" relation, in that it indicates that the two Concepts refer to a single entity rather than two identical entities.

I decided to implement this method using conceptual graphs for a number of reasons. One reason is that the graphical format is easy for a human user to understand. Expressive power is another. Using conceptual graphs, it is possible to have the knowledge base's data and its rules represented in the same format, allowing the rules to be checked without the need for translators or other outside operators.

Hunter2 describes a method for measuring how inconsistent a knowledge base is. This is done by measuring the “information content” of sets of data [Hunter2]. The more consistent information a set has, the higher the information content [Hunter2]. It was not possible to measure the level of inconsistency of a set using classical logic, so the author uses “quasi-consistent logic”, instead Hunter2. Approaches like this one can be useful in comparing the consistencies of two knowledge bases, such as when two knowledge bases are to be merged. Although it could theoretically be used to locate the source of inconsistencies by measuring the information content of progressively smaller sets, this would require too much processing time to be usable. In order to locate an inconsistency, it may be necessary to check every possible subset in the knowledge base. Therefore, approaches like this are best used when it the sources of inconsistency are already known, or are not relevant.

Dubois1 combines quasi-classical and possibilistic logic into “quasi-possibilistic logic,” and uses this new logic to handle existing inconsistencies in a knowledge base. It uses this logic to define a measure of consistency called “coherence” [Dubois1]. Sets with a coherence measure of 1 are considered completely consistent, and those with a measure of 0 are considered completely inconsistent [Dubois1]. Computing this coherence measure takes advantage of concepts from then four-values logic used in some other approaches described here. The coherence of a statement is based on whether it is assigned the value

of “Both” (both True and False) or “Neither” (Neither True nor False) in four-valued logic [Dubois1]. Dubois1 also defines a “significance function,” a measure of how harmful a particular inconsistency is to the consistency of the rest of the knowledge base. Computing this measure requires some meta-knowledge about the types of information in the knowledge base and their relative importance [Dubois1]. An inconsistency is given a higher significance value if it involves data that is considered more vital to the function of the knowledge base [Dubois1]. Inconsistencies in data considered peripheral to the knowledge base are given a low significance value.

Although measuring the level of inconsistency within a knowledge base or between two knowledge bases is useful in some contexts, the work here is more concerned with locating specific data items or sets of data items that are causing inconsistencies. The inconsistencies the algorithm here is concerned with are between data items and knowledge base rules, rather than between two data items or two knowledge bases. The analysis performed here is assumed to be on a single stand-alone knowledge base. If two knowledge bases are involved, it is assumed that they have already been merged, and the exact details of how they are merged is not taken into account. It is also not concerned with the overall level of inconsistency in a knowledge base. or how inconsistent an item or set is in relation to others. Classical logic is sufficient to accomplish this, so possibilistic logic was not used. It is possible that it will be used some future work.

Liu2 describes a rule-based system for detecting inconsistencies in UML models. The system is based on rules which describe inconsistencies and what actions to take when these inconsistencies are encountered. The methods for dealing with the inconsistencies are described by another set of rules which represent the knowledge base user’s choices for how to resolve the inconsistency. Inconsistencies are either removed or modified. There is no option for working around them.

Sourrouille1 describes a method for describing and checking constraints for UML models, which the author calls "Constraint Checker" or "CC." In this approach, constraints are represented as notes attached to the UML model elements [Sourrouillle1]. The constraints consist of five "tags" which define the

constraint itself, the level at which the constraint is checked, the conditions under which the constraint applies and the state of the model if the constraint is not satisfied [Sourrouillle1]. The actual constraints are represented by Boolean expressions. The expression must always be true under the conditions described in the tags, or the constraint is not satisfied. These constraints were intended to be able to be checked automatically, and also allow for the Constraint Checker to be able to offer suggestions about how to rectify any violations found [Sourrouillle1]. Because the constraints used by the Constraint Checker and the UML model elements are not represented in the same format, there must be a mechanism for the two to communicate in order for it to be used. The author addresses this problem by using XML as a communication format and developing tools to translate XML into Constraint Checker files [Sourrouillle1]. This work differs from the work presented here in that the constraints require a parser or some other outside tool in order to be checked against the knowledge base they constrain. In the approach presented in this thesis, the constraints are represented in the same format as the data they apply to. Therefore, there is no need to translate them into another format in order to check them. The constraint checking could conceivably be done by the inference engine itself, without the need for outside tools.

Conceptual graphs have been used to represent constraints in Salomons1. In this paper, the author uses conceptual graphs to describe manufacturing constraints. This work is somewhat different than the work presented in this thesis, in that the constraints in this paper describe the geometric and spatial relationships between physical objects, rather than between data items. It represents manufactured items, or parts of items, as concepts, and the relative positions of the items, such as "against" or "parallel" as relations [Salomons1]. The author's approach to checking the actual items against the constraints does not directly use the conceptual graphs. The graphs are only used to represent the constraints. The knowledge represented by the graphs is interpreted by the inference engine to verify that the items in question satisfy the constraints. By contrast, the approach presented here takes advantage of the graph representation to verify that the data is consistent with the constraints.

Some more similar work was done in Mineau1. This paper describes a method for representing constraints on agent interactions. Conceptual graphs are used to represent both the actions in question, and the constraints on them. The actions are represented using demons to represent the preconditions and postconditions of a specific action [Mineau1]. The constraints in this method are represented as situations that are never supposed to happen. They are graphs that should test as true [Mineau1]. Verifying that the actions conform to the constraints is done by computing projections of the constraint graphs onto the action graphs. If such a projection exists, the actions is said to violate the constraint [Mineau1]. This work is similar to the work presented here in that it is using conceptual graphs to represent constraints on other conceptual graphs. Where it differs is in the handling of exceptions. In Mineau's paper, exceptions to constraints are represented as separate graphs, and are linked to the constraint graph by an exception operator [Mineau1]. The exact nature of the operator is not specified. My approach, is contrast, is able to handle exceptions using only the standard conceptual graph operators. This is discussed in more detail in Chapter 3.

CHAPTER III

APPROACH

Constraints are represented by conceptual graphs. Constraint graphs have two parts, a "pattern" and a "constraint." The pattern refers to what types of subgraphs the constraint applies to. It specifies a graph that is intended to match a subgraph in the knowledge base exactly. An example pattern is pictured below. This pattern would mean that the constraint attached to it only applied to "Widget" concepts that had a "color" attribute of "Blue."

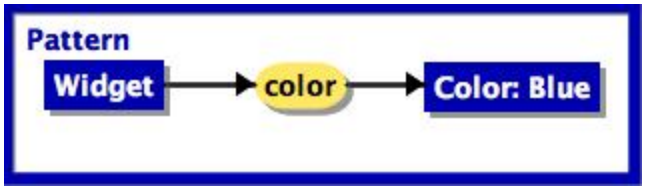


Figure 3.1 Example of a Pattern Subgraph

The constraint part defines what form the subgraph in question can take. An example of a constraint graph is pictured below. This graph says that is impossible for a "Widget" concept to have a "size" attribute of "Small." The blue-shaded area in Figure 3.2 is called a "negative context," and indicates that the contents are represented as false, like a NOT operator in a Boolean expression.

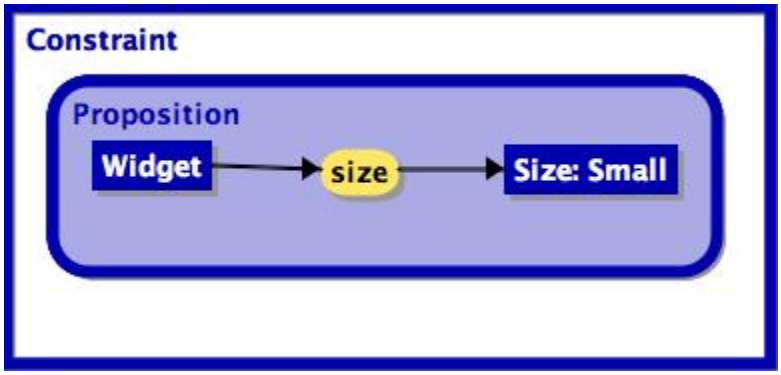


Figure 3.2 Example of a Constraint Subgraph

The entire constraint consists of both these parts, connected by a "constrains" relation. This relation connects a constraint subgraph with a pattern subgraph, and identifies which subgraphs the constraint applies to. Each constraint contains one pattern subgraph and one constraint subgraph connected with the “constrains” relation. It is assumed that each constraint only applies to one type of subgraph. In a situation where a constraint graph applies to more than one pattern, it is necessary to create multiple constraint, one for each pattern. Identity lines are drawn between the relevant parts of the pattern subgraph and the constraint subgraph. The example shown below represents a rule that states that blue Widgets cannot be small. More specifically, the example is a constraint that applies to all Widgets with a color attribute of Blue, and states that the Widget in question cannot have a size attribute of Small.

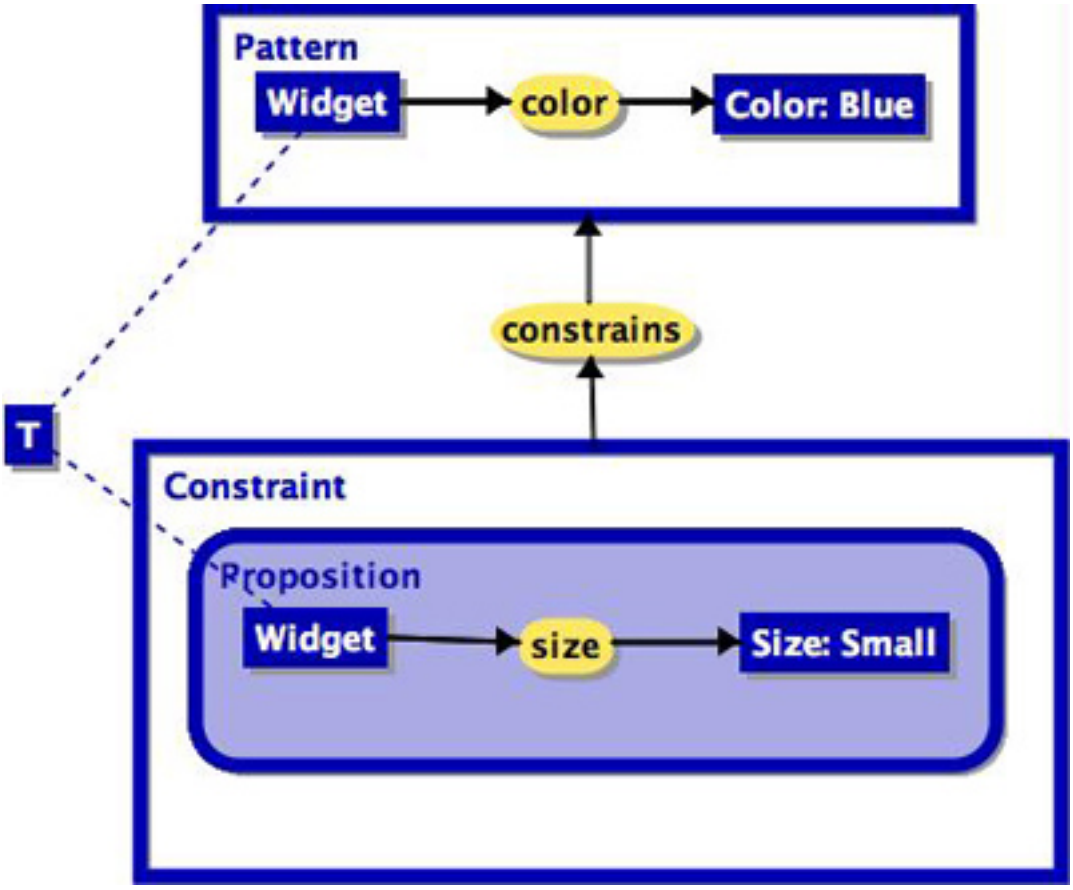


Figure 3.3 Example of a Complete Constraint Graph

This figure is an example of a mutual exclusion constraint, which will be discussed in more detail later.

For the purpose of this thesis, there are three types of constraints that are considered to be basic for the purposes of the thesis: cardinality constraints, mutual exclusion constraints and allowed values constraints. A cardinality constraint is one that puts limits on how many of a certain data item can exist or how many of a certain attribute a data item can have. This can be in the form of a maximum amount, a minimum amount, or a range. Below are some examples of cardinality constraints.

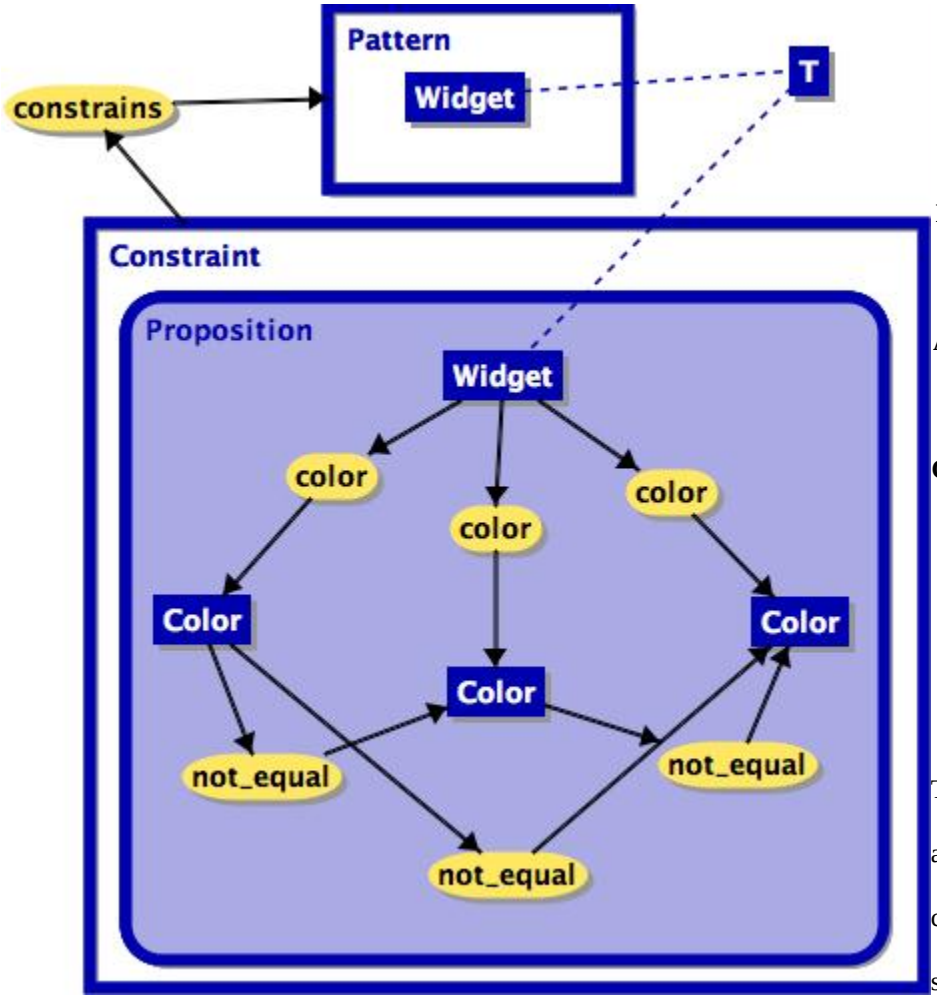


Figure 3.4 Example of an Attribute Value Cardinality Constraint

The above constraint states that a widget may not have more than two unique color attributes. More specifically, it states that it is not the case that a Widget will have three color attributes that are not equal to one another. If a Widget has three identical color attributes, for example, if it were marked as Red in three different places, this would not be a violation of the constraint. Constraints of this type require that the not_equal relation be defined as indicating that one concept has a value that is not identical to the other.

Although this maximum-value constraint will work if is made to apply to all instances of a particular node, the number of checks performed can sometimes be reduced by having the constraint apply only to nodes that already have at least the maximum amount of attributes, as in the example below.

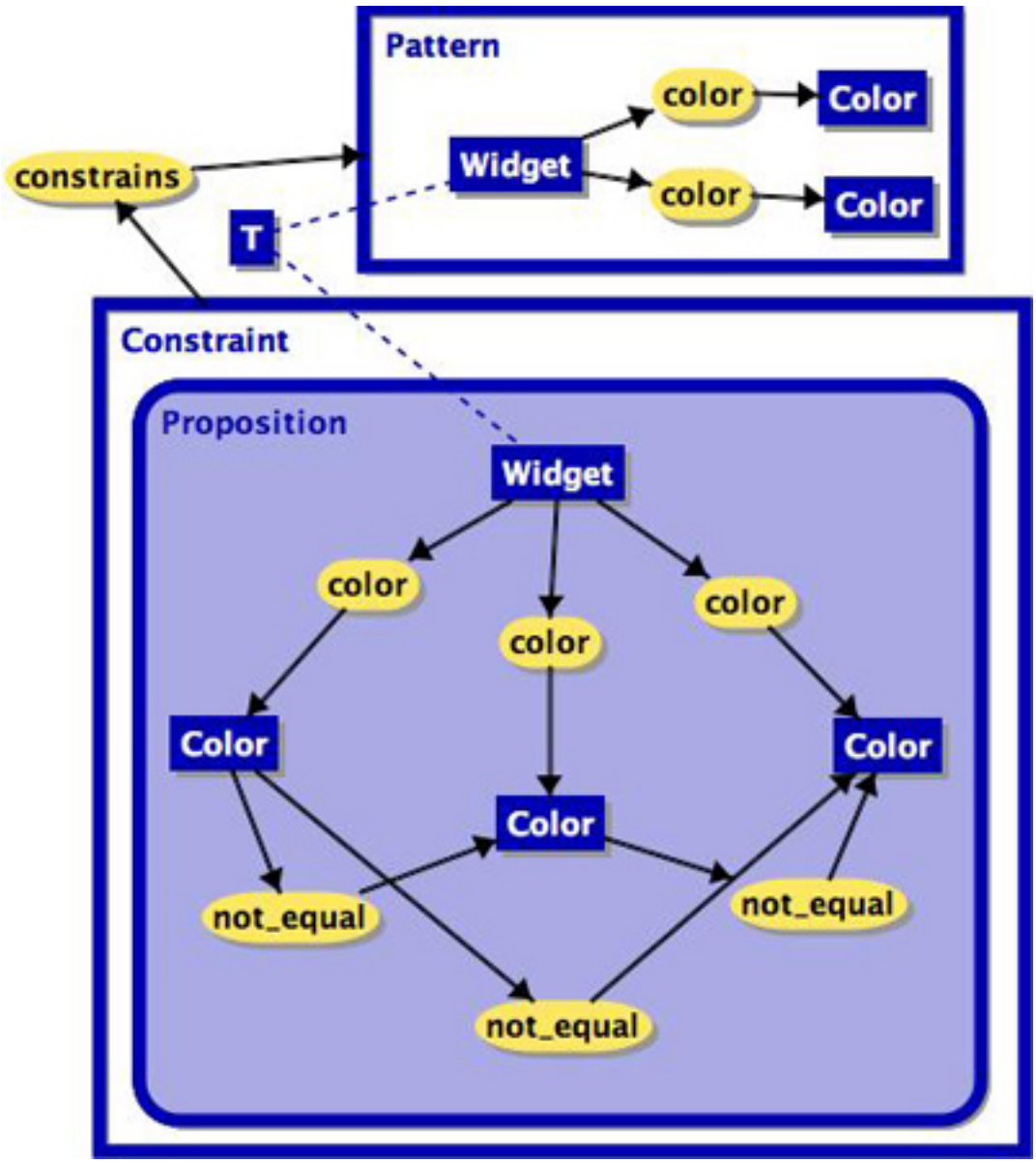


Figure 3.5 More efficient maximum cardinality constraint

As the contents of the Pattern graph are required to match the knowledge base subgraph exactly, the "not_equal" relation is not included. Therefore, even with the addition of the two color attributes, the constraint cannot distinguish whether the two Color attributes are identical or not. Therefore, this may not limit the checks to only those with the maximum amount of attribute values. It will still cut out some instances that definitely have less than the maximum values.

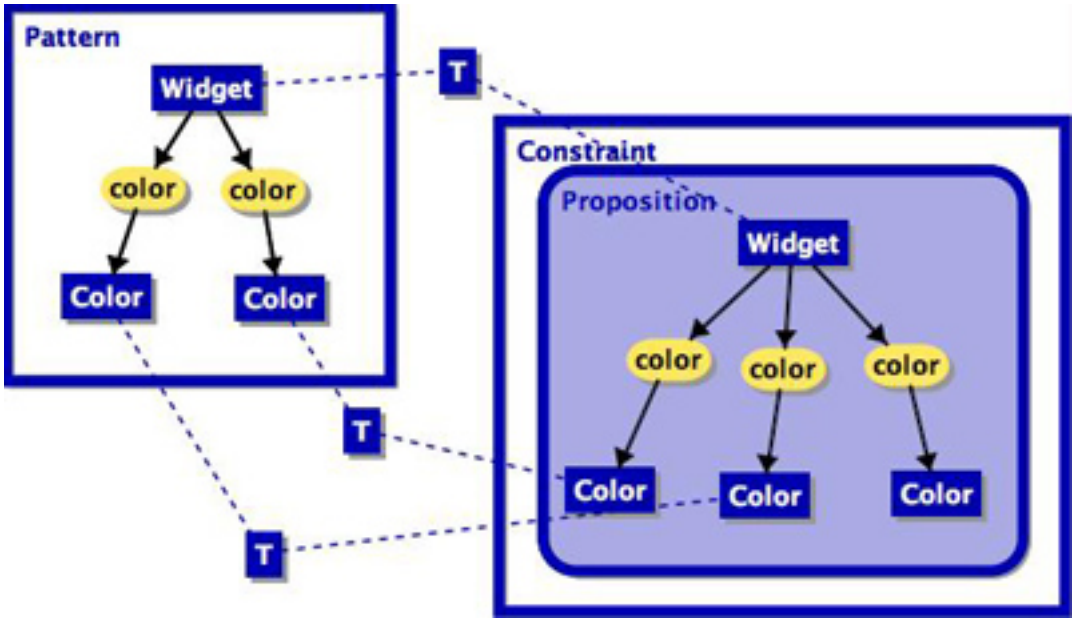


Figure 3.6 Example of an Attribute Cardinality Constraint

The above constraint states that a Widget may not have more than two colors attributes, regardless of the value of these attributes. Specifically, it says that if a Widget has two color attributes, it may not have a third that is distinct from the first two. This kind of cardinality constraint is not concerned with the values of the attributes. A Widget with three color attributes of Red would be a violation, as would one with color attributes of Red, Green and Blue.

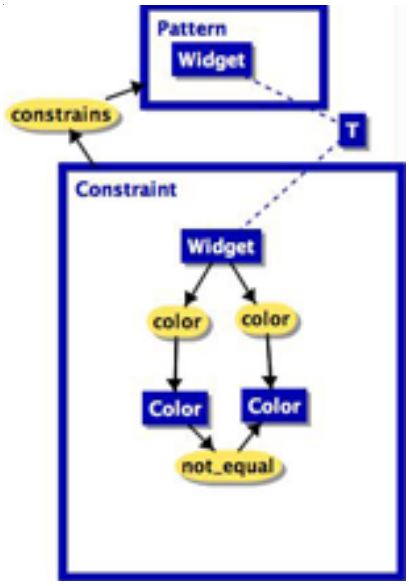


Figure 3.7 Example of a minimum value constraint

Constraints that specify the minimum number of attributes or unique values are handled in a similar fashion, but without the negation context. The above graph is an example, showing a constraint that specifies that Widgets must have at least two colors. This type of constraint also requires a "not_equal" relation to be defined. Constraints that specify an exact number is attributes or values can be created by a combination of the above two types. For example, of a Widget were to have exactly 2 Colors, this could be accomplished by one constraint that specified that Widgets must have at least 2 Colors, and one that specified that Widgets may have no more than 2 Colors. Constraints of this nature can be represented as two separate constraints, or as a two-part constraint. Two-part constraints are described later in this chapter.

A mutual exclusion constraint is one that specifies two or more situations which cannot exist simultaneously. If any one of the situations specified by the constraint is true, then all others must be false. Below are some examples of mutual exclusion constraints.

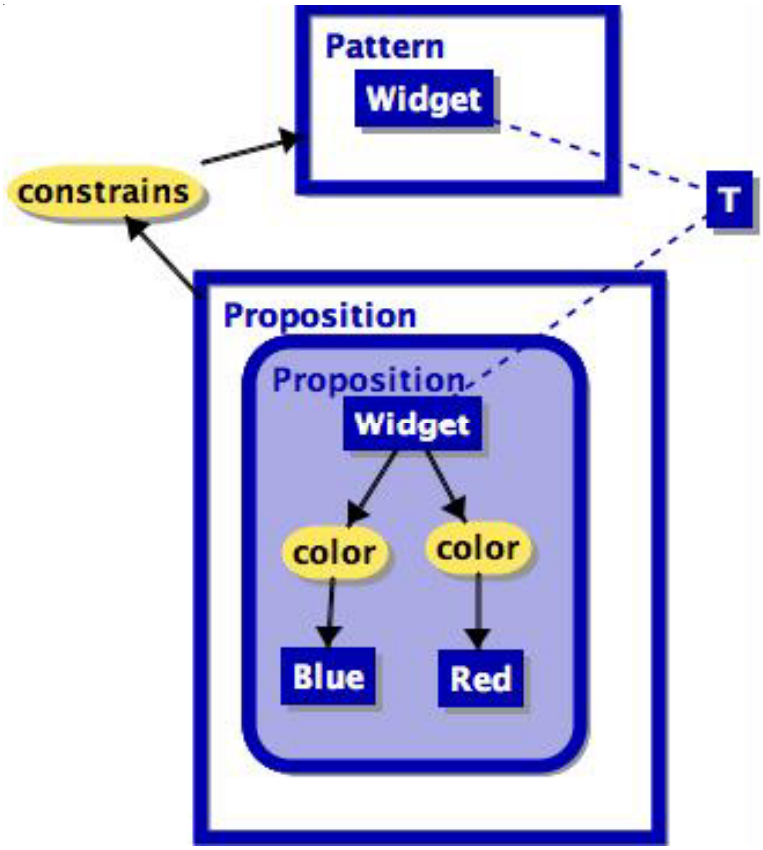


Figure 3.8 Example of a Mutual Exclusion Constraint

The above constraint states that a Widget may not have color attributes of Blue and Red at the same time. While this graph is sufficient to describe the constraint, it may not be desirable from a performance standpoint. As it is, it applies to all Widget concepts, which means that the checking algorithm will have to check every Widget concept in the graph. Below is a more streamlined version.

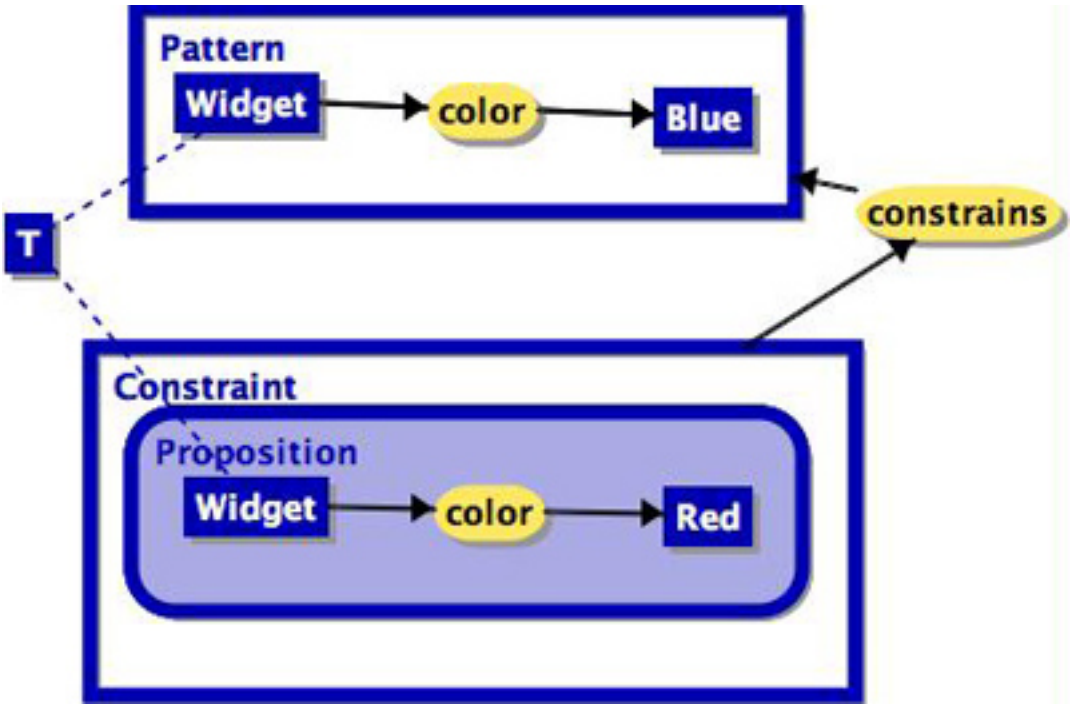


Figure 3.9 A More Efficient Mutual Exclusion Constraint

The above constraint states that all Widgets with a color attribute of Blue cannot also have a color attribute of Red. This is more efficient than the previous example because it only applies to Widgets with a Blue attribute, instead of to all Widgets. It is not necessary to have a matching constraint that applies to red Widgets. If Blue and Red are mutually exclusive, that means that a single Widgets cannot have both attributes. A Widget cannot have both Blue and Red attributes if it does not have a Blue attribute. Therefore, if a Widget does not have a Blue attribute, whether or not is has a Red attribute is not relevant as far as this constraint is concerned.

An allowed-values constraint applies to attributes of data items. It specifies what values an attribute is allowed to take. Below is an example of an allowed-values constraint.

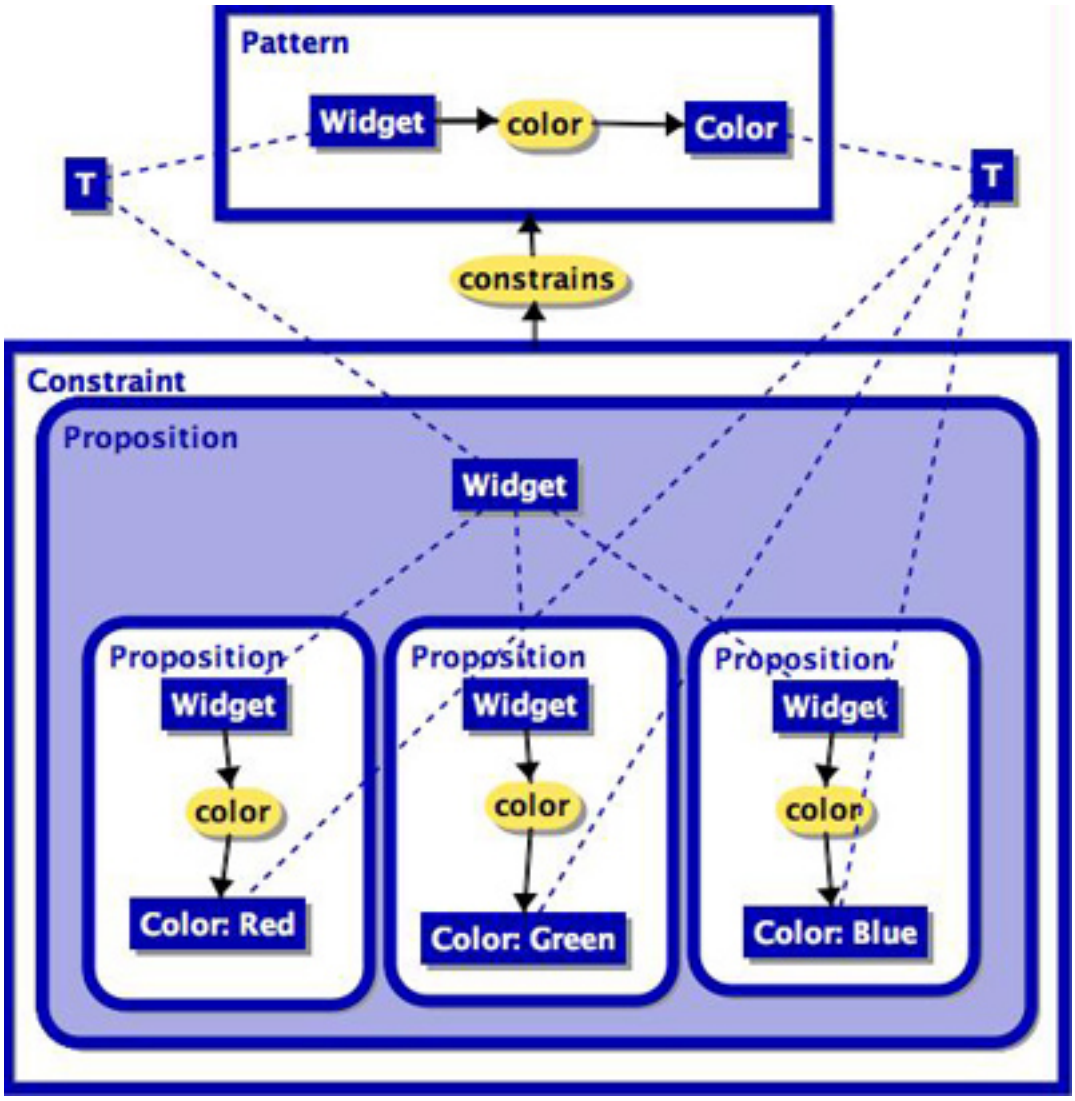


Figure 3.10 Example of an Allowed-Values Constraint

The example above shows a constraint that says that the "Color" attributes for a "Widget" concept can be Red, Blue or Green. More specifically, it says that if a Widget has a Color attribute, the value of that attribute cannot be not Red, not Green and not Blue. A value other than Red, Green or Blue, such as Orange, would be considered a violation of the constraint.

These three constraint types can be combined to form more complex constraints. For example, a knowledge base could have a constraint that stated that "Widget" items can have one and only one "Color"

attribute of Red, Blue or Green. This kind of constraint can be represented in two ways. In simpler cases it is possible to represent a more complex constraint using a single graph, like in the example below.

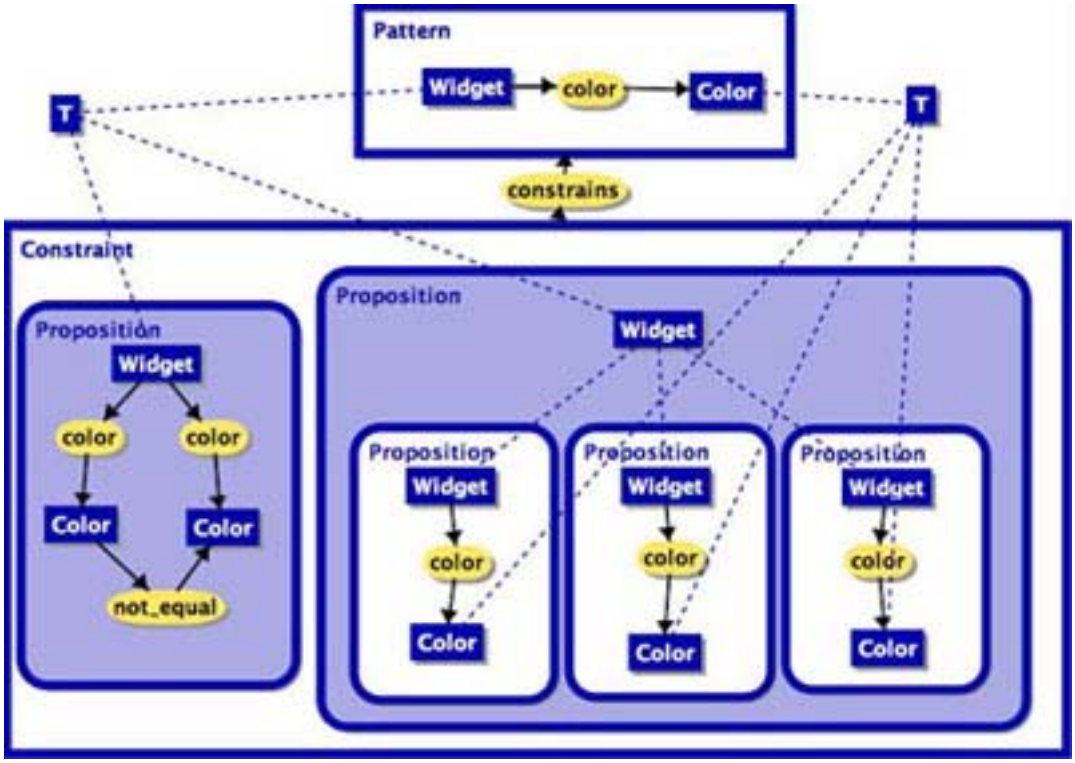


Figure 3.11 Example of a two-part constraint

In more complex cases, it is necessary to break the constraint up into simpler ones and represent it using several smaller graphs. For the example above, we would break it up into two constraints: one that specified that a Widgets color could be Red, Green or Blue, and one that specified that Widgets can only have one color.

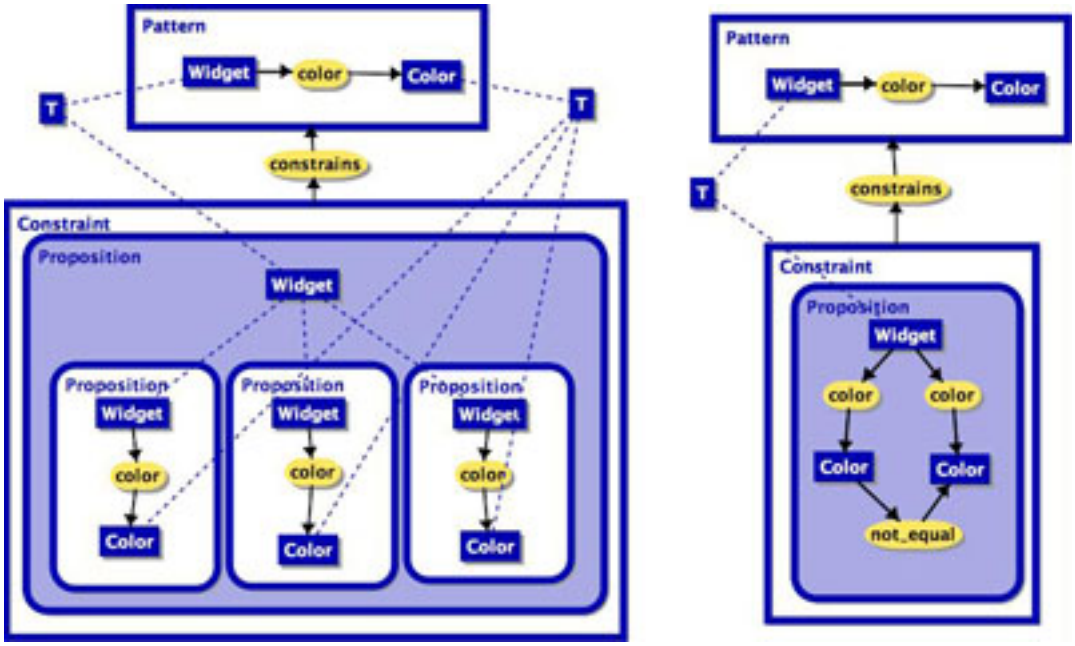


Figure 3.12 Example of a separated two-part constraint

The above types of constraints can all have exceptions and special cases. It is also necessary to be able to represent these. Sometimes a rule only applies if a certain condition does not hold. For example, a knowledge base may have a rule that states a Widget may only have two Color attributes, unless one of the attributes is Blue. Constraints like this are represented similar to a first-order logic "or," where a subgraph can either satisfy the constraint *or* have the exception case. A example of such a constraint is shown below.

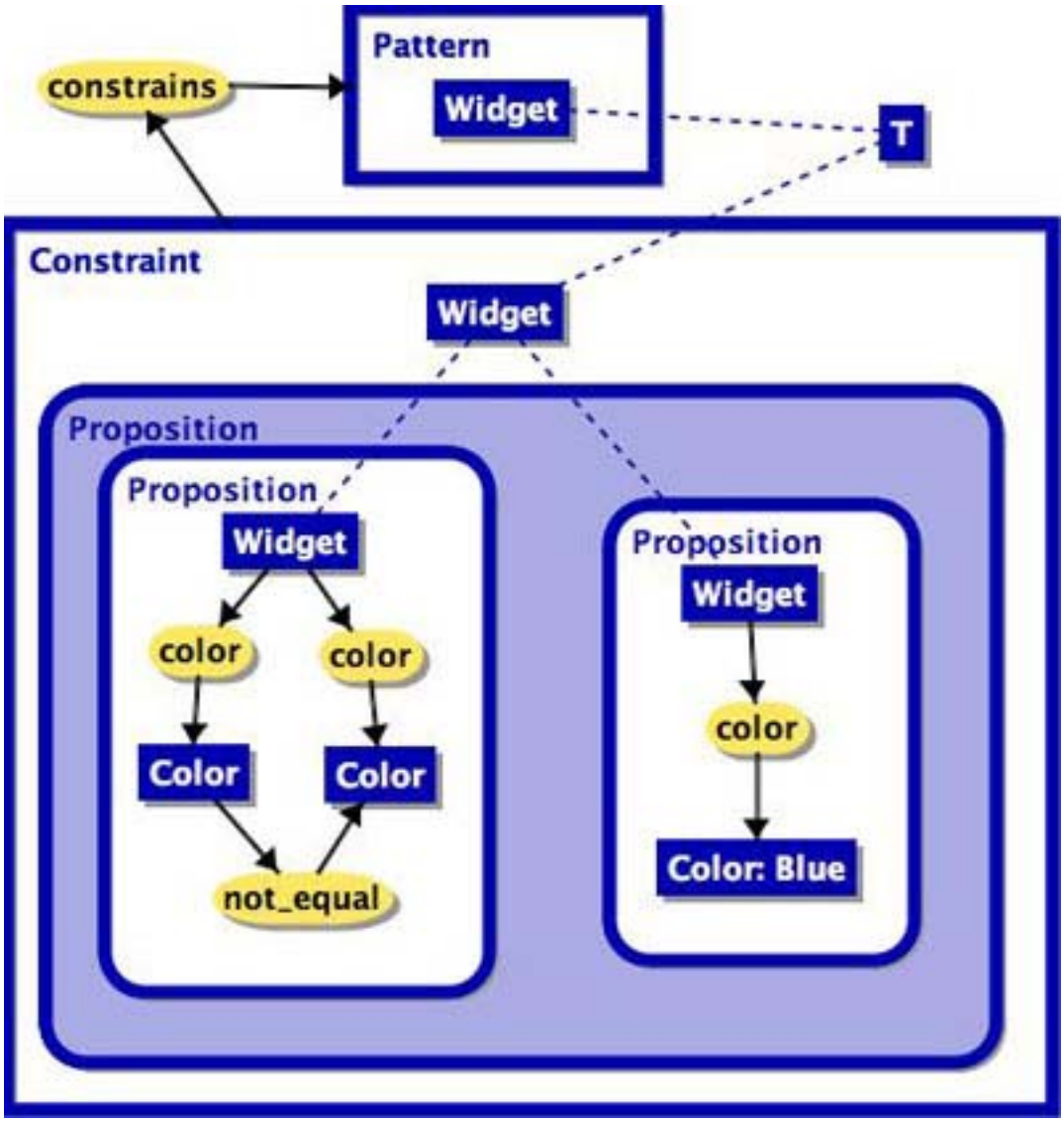


Figure 3.13 Example of a Constraint with an Exception

This constraint states that a Widget can either have two or fewer Color attributes, *or* have a Color attribute of Blue. In this case, a Widget concept would be in violation of this constraint if it had more than two Color attributes and did not have a Blue attribute.

The previous constraint example did not have any information on how many Color attributes a Widget could have if it was an exception to the rule. It is possible for a knowledge base to have a rule where this is specified. For example, there could be a rule that states that a Widget without a Blue attribute

can have two Color attributes, and one with a Blue attribute can have three. Cases like this would be represented by two constraint graphs. One would apply to all Widgets, and state that they can only have two Color attributes unless one is Blue. The second would only apply to Widgets with a Color attribute of Blue, and would state that they can have three Color attributes. An example is pictured below.

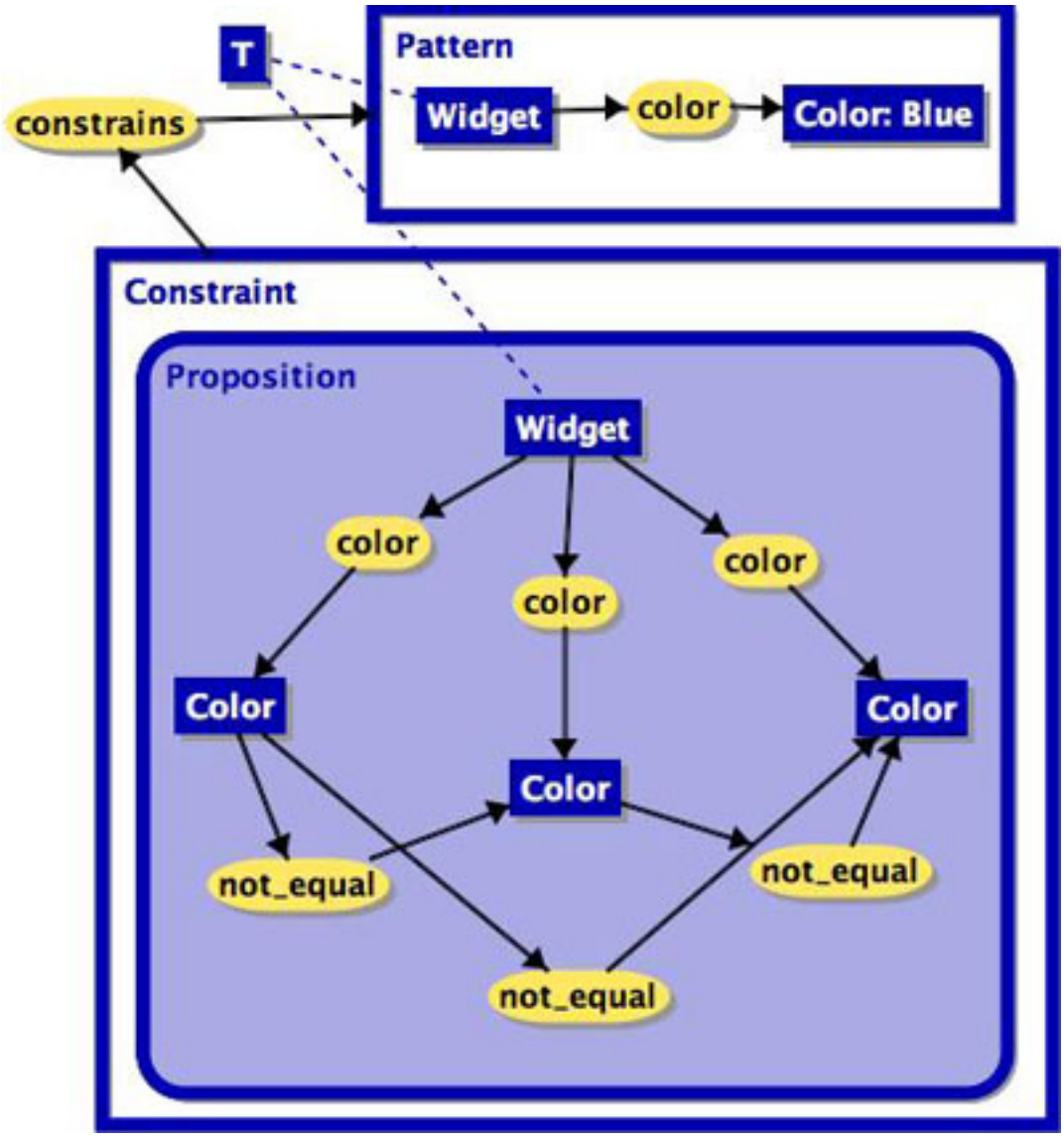


Figure 3.14 Example of a Constraint with a Special Case

A question that could be asked about modeling constraints with special cases is, why not have one constraint that applies only to subgraphs that contain the special case and one that applied to those without the special case? At first glance, it would seem that this approach would be more efficient than having one constraint apply to all subgraphs, because it would only have to check the subgraphs that contain the special case. The problem lies in determining that whether or not a subgraph is an exception. The constraint's pattern subgraph is meant to match the knowledge base subgraph exactly. It specifies what *is* there, not what isn't there. It would be possible to use a negation context, or define a "not" operator to specify what the subgraph to be matched should not contain, but this would require more processing time in order to determine if a subgraph is a match. For example, if the constraint graph specifies a "Widget" node with a "Blue" color attribute, the algorithm only needs to search until it finds a single Blue attribute, regardless of how many other color attributes the Widget has. In a best-case scenario, it finds a Blue attribute on the first try and doesn't need to look at the rest of the Widget's attributes. If it is looking for a Widget without a Blue color attribute, it must check all color attributes for every Blue node it finds. Unless the number of cases that do not contain the special case is small compared to the knowledge base as a whole, there is no gain in performance from using a constraint that only applies to subgraphs that do not contain the special case.

Being able to represent constraints is only one part of the puzzle. In order for the constraints to be useful, it is necessary to be able to check the data against the constraints. To check a constraint, the system would look would subgraphs that matched the constraint's pattern subgraph, as in the example below.

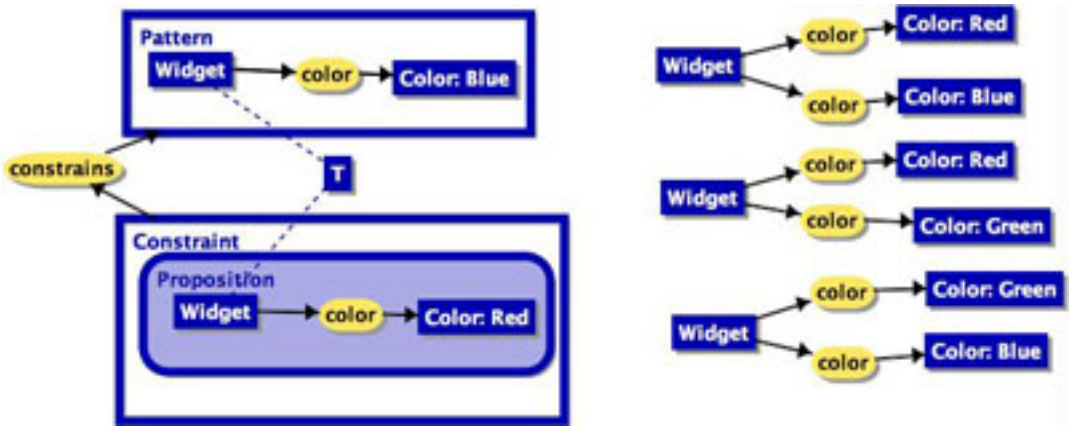


Figure 3.15 Example Knowledge Base

For each matching subgraph found, identity lines are drawn between the relevant parts of the pattern subgraph and the matching parts in the found subgraph. The graph containing the constraint and the subgraph is then evaluated. If it evaluates as true, as in the example below, the subgraph is considered consistent with the constraint.

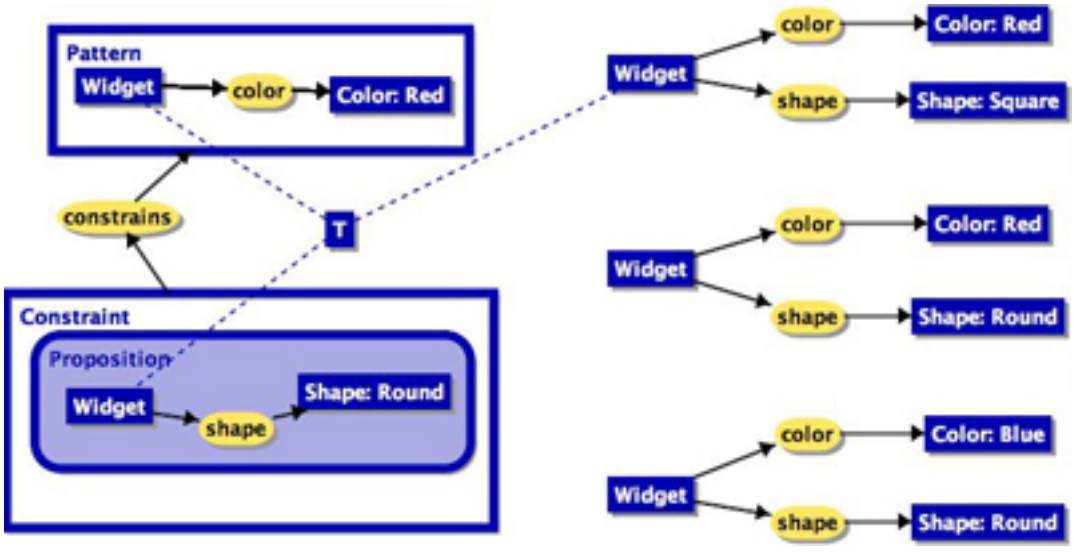


Figure 3.16 Example of a Consistent Subgraph

If the graph tests as false, as in the example below, the subgraph is considered a violation of the constraint.

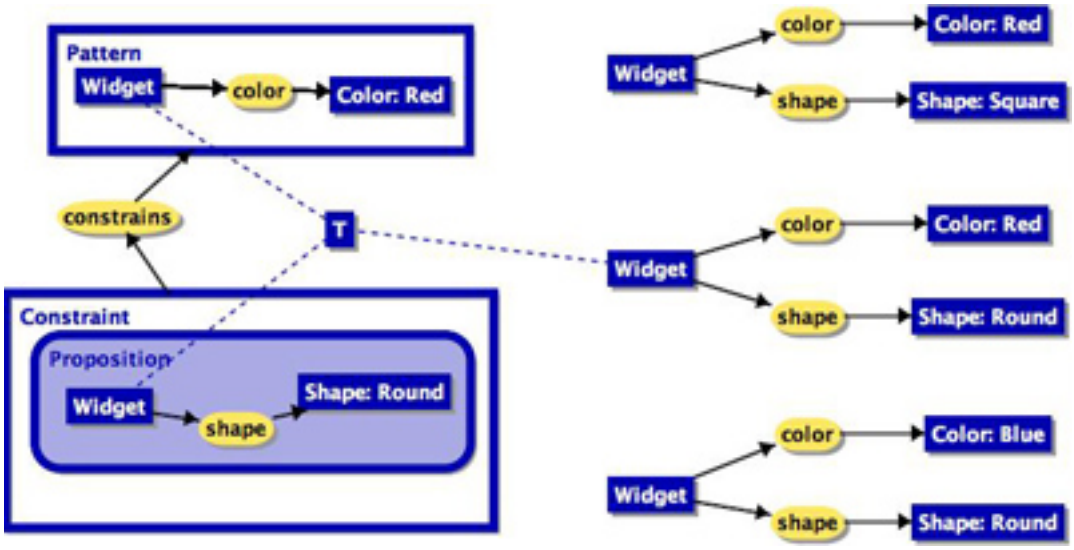


Figure 3.17 Example of an Inconsistent Subgraph

Once the test is performed and the subgraph marked either consistent or inconsistent, the identity lines between the found subgraph and the pattern subgraph are erased.

CHAPTER IV

PROPOSED WORK

The ultimate purpose of this thesis is to develop an algorithm for checking the contents of a knowledge base against a set of constraints. The first step in this algorithm will be to determine which parts of the knowledge base to check for a given constraint. The most general solution is to check every possible subset of knowledge base nodes against each constraint. However, this is not a desirable solution. For a knowledge base containing n nodes, this would involve checking 2^n subsets for each constraint. Most of these checks would be either irrelevant or impossible to check, since if the subset does not have any nodes in common with the constraint's pattern, the system will have no way to check it.

One way to eliminate these unnecessary checks is to only check subsets that are the same size as the pattern. For example, for a constraint of size *k*, all subsets of size *k*, regardless of how the nodes were connected, would be checked. For a knowledge base of size *n* and a constraint of size *k*, this will require checking n-choose-k subsets per constraint.

We can further reduce the amount of unnecessary subset checks by taking the nodes in the subset into account. In each constraint, at least one node in the pattern will connected to a node in the constraint part by an identity line. The constraint will only be relevant to subsets containing these nodes, so there is no need to check subsets that do not contain them.

Only taking the nodes themselves into account can still result in unnecessary checks. If the two nodes are connected in the pattern, checking two unconnected nodes, even if they are of the same type, will not be relevant. For example, if the constraint applies to Widget items with a Blue color attribute, the

algorithm only needs to check Widget and Blue nodes that are connected to each other. If the two are not connected, the two nodes may not have anything to do with each other, and the constraint may not be applicable. There is the possibility that missing connections in the knowledge base could cause inconsistencies, but the algorithm used here would not be able to find them. To further reduce the amount of checks made, it is also necessary to take the connections between the nodes into account. This can be done using a depth-first-search algorithm to compare the knowledge base subset to the pattern. One node in the pattern is designated as the "pivot" node, and the knowledge base is searched for subsets containing this node. Picking the most efficient pivot node requires some knowledge about the structure of the knowledge base, which this algorithm does not have. Therefore, it is assumed that either the primary pivot has already been designated by the creator of the constraint, or there exists an algorithm that will determine the pivot for a given constraint. Each subset is compared to the constraint's pattern, and if it matches, the consistency check is done. If a pattern contains more than one unconnected subset, there will need to be a pivot node for each connected part. For a knowledge base of size k and a constraint of size c, there are at most k pivot node instances in the knowledge base, and c-1 non-pivot nodes in the constraint. At most c-1 knowledge base nodes will be checked for each non-pivot node in the constraint. For each pivot node instance in the knowledge base, a maximum of $(c\text{-}1)^2$ nodes will be checked. In the worst case scenario, k = c. This is not a common scenario, but it is possible, if , for example, the kowledge base were very small, or there was a constraint meant to apply to the knowledge base as a whole. In this situation, $k * (k\text{-}1) * (k\text{-}1)$ nodes checked per constraint.

Nodes that have an identity line linking them to a node in the constraint graph are considered essential to the constraint, and can potentially be used as pivot nodes. If there is only one node with a link to the constraint graph, there is no question which node to use. However, some problems can arise if the constraint depends on multiple nodes. If only a single pivot is used, choosing the wrong pivot node can result in unnecessary checks being done, or important information being skipped. For example, consider the following constraint and knowledge base fragment.

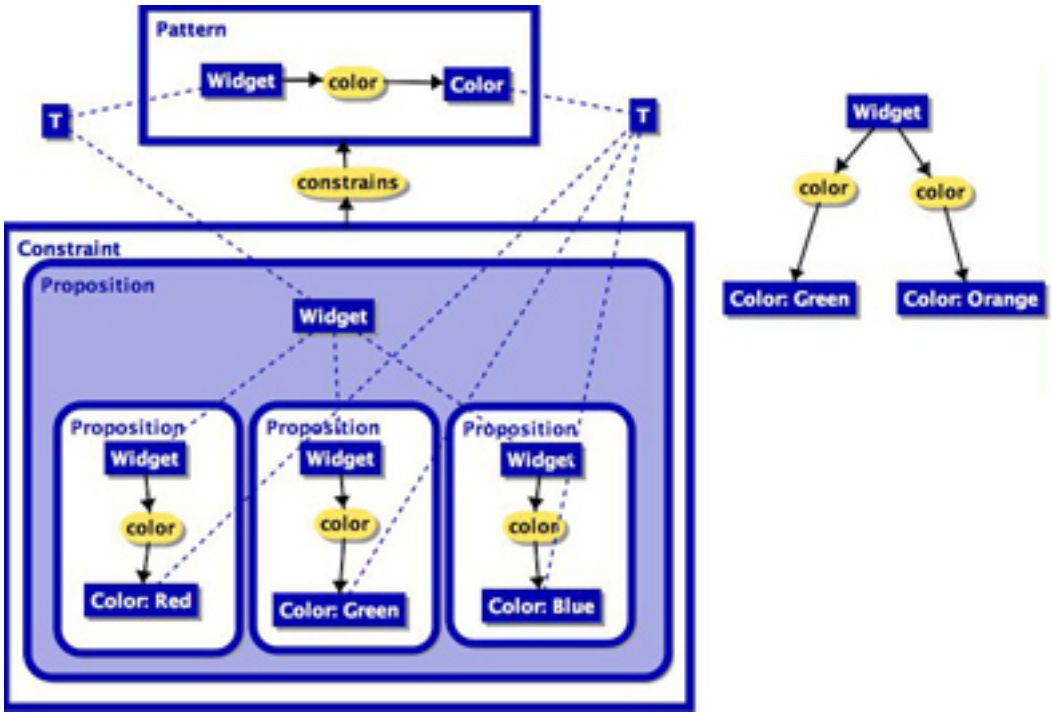


Figure 4.1 Pivot node example

If the Widget node is chosen as the pivot node, the data graph shown will only be checked once, and only one of the Color attributes, the one the algorithm finds first, will be taken into account. If the Green color is found first, the set will be marked as consistent, and the Orange color ignored, leaving an inconsistency in the knowledge base that is not found. If the Color node is used as a pivot node, more checks are made, but the inconsistency is found.

This problem can be avoided by using an ordered set of pivot nodes, instead of only a single one. One of the nodes is designated the primary pivot node. A primary pivot node is simply an instance of a particular node, and is used to determine whether the subgraph in question is to be checked, just as if there were only a single pivot node. Other nodes in the pattern subgraph with links to the constraint subgraph can be used as secondary pivot nodes. A secondary pivot node is defined as an instance of a particular node, with a specific direct relationship to the primary pivot node. A pivot node is defined as being in a direct relationship with another if there is a path in the graph connecting the two that does not go through any other pivot node. In the approach used below, the shortest path between the two nodes takes precedence, so

if the shortest path contains a pivot node, it is not considered a direct relationship, even if a longer path that does not contain a pivot node exists. This does not have to be the case with every knowledge base.

Figure 4.2 Direct relationship example

In the pattern graph shown in Figure 4.2, assuming that all the nodes shown have identity lines connecting them to nodes in the constraint graph. If the Widget node is designated as the primary pivot, the Color node can be designated as a secondary. The Scarlet node, however, cannot be a secondary pivot, because there is no path connecting it to the Widget node that doesn't go through a secondary pivot node. However, since there is a direct relationship to a secondary pivot, it can be used as a tertiary pivot. If the Color node was designated as primary, both the Widget and Scarlet nodes could have been designated as secondaries. If the Color node was not a pivot node, the Scarlet node could be a secondary pivot.

In the example used in Figure 4.1, we can designate the Widget node as primary. This means that a subgraph in the knowledge base must contain a Widget node in order to be considered relevant to the constraint. Once a Widget node is found, there is an extra step. The Color node is designated as a secondary pivot node. This means that the knowledge base subgraph must have not only a Widget node, but a Color node in the same relationship to the Widget node as in the pattern subgraph to be checked. A primary pivot node can be linked to multiple secondary nodes, making it possible for multiple Color attributes for a single Widget to be checked.

For more complicated pattern graphs, there can be tertiary pivot nodes, defined as an instance of a particular node in a specific direct relationship to a secondary node, or even quaternary pivot nodes. Of course, the computational complexity increases with the amount of pivot nodes used.

One question that can be asked is why the hierarchy is necessary. Taking the example used in Figure 4.2 and assuming that all three nodes shown are used as pivot nodes, can the Scarlet node be defined

as attached to the primary through a Color node, ignoring the fact that the Color node was a pivot? The example below shows how this approach can lead to problems.

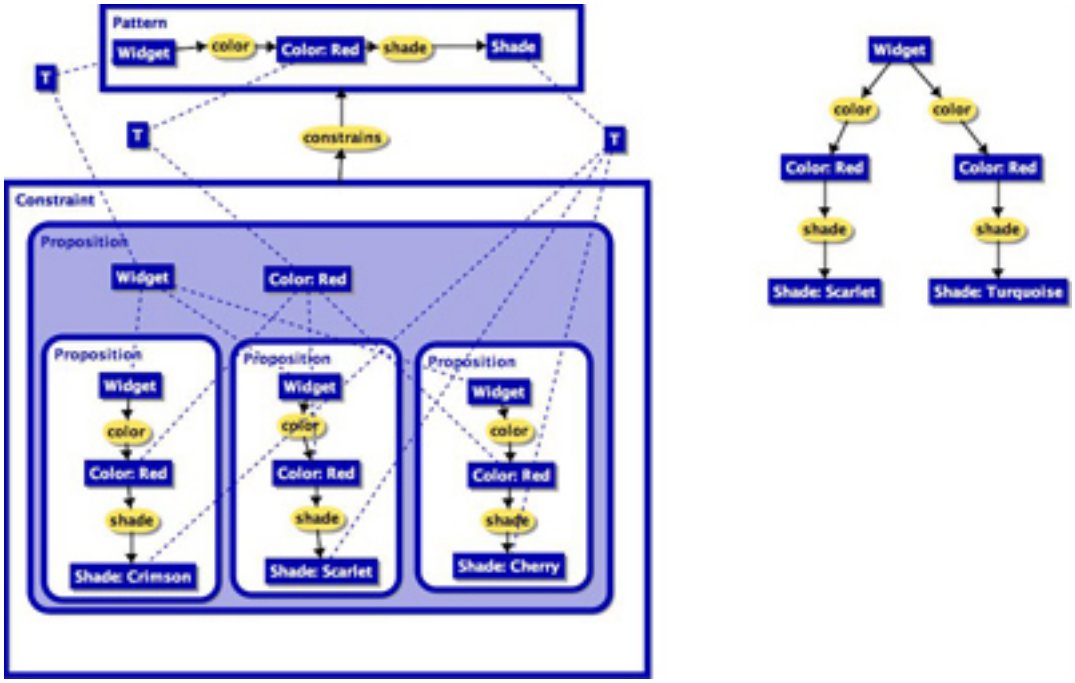


Figure 4.3 Pivot node hierarchy example 1

The constraint above states that if Widgets have a Color attribute of Red, the Red must have a Shade attribute of Crimson, Scarlet or Cherry. The knowledge base fragment contains a Widget that has two Red attributes, one with a shade of Scarlet and one with a Shade of Turquoise. All three of the concept nodes in the pattern subgraph are used as pivot nodes, because all three have identity lines to nodes in the constraint subgraph. For the purpose of the example, the Widget node is considered the primary pivot, making the Color node a secondary and the Shade node a tertiary pivot. Now that the pivots are determined, the next step is to decide which nodes in the knowledge base fragment to test. There are two possibilities for the secondary node, and two possibilities for the tertiary node. If the tertiary pivot node is not required to be connected to a secondary pivot node, there is nothing to guarantee that the Shade node being tested is connected to the Color node being tested, as in the example below.

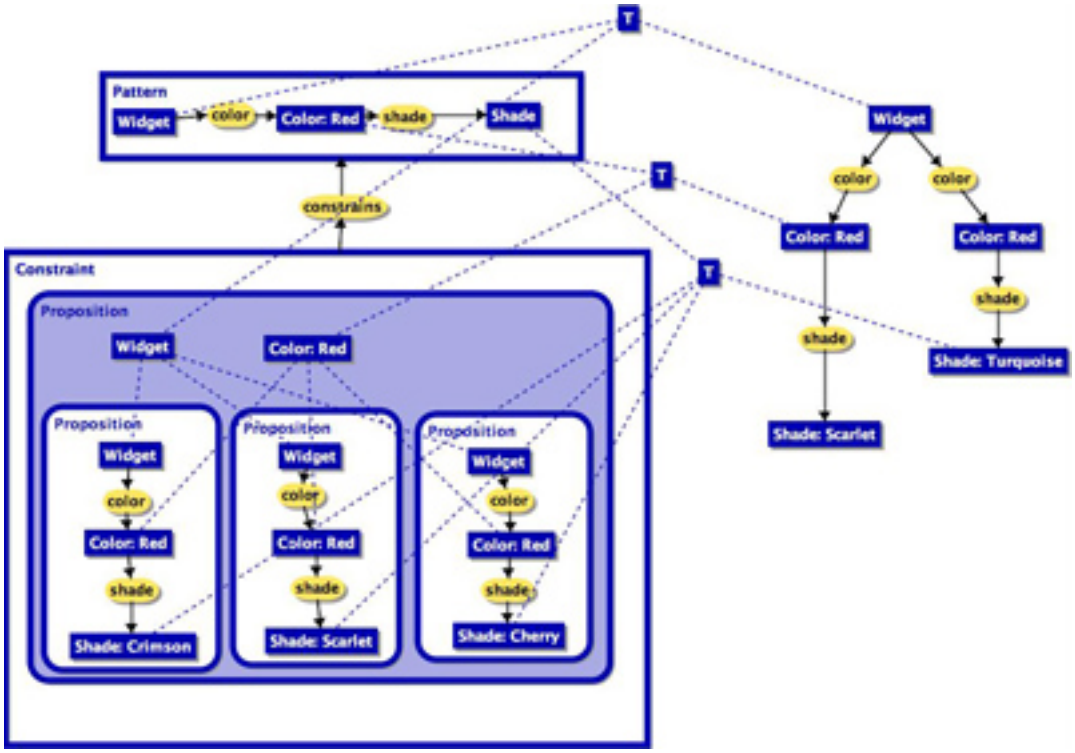


Figure 4.4 Pivot node hierarchy example 2

If the test were performed with the identity lines as shown, the subset would test as inconsistent, even though the Color node in question does not violate the constraint. Incorrect identity lines like this could result in incorrect subsets being marked inconsistent. However, if the Shade node were required to be connected to a Color node being used as a secondary pivot, the condition shown above could not happen.

The question now becomes how to determine which node to consider primary, and which to consider secondary. For any constraint, the most "important" node should be used as a primary pivot node. However, what constitutes "important" differs depending on the knowledge base and how it is represented. Therefore, some metaknowledge is necessary to determine the best method for picking the correct pivot node. There is no easy, automated way to pick the best pivot node every time. There are, however, some methods that can pick useful pivot nodes most of the time. Again, the consistency-check algorithm does not determine the pivot nodes. It is assumed that they are already determined, either by the knowledge base creator, or by a pre-existing algorithm. Developing an algorithm to determine the most efficient pivot nodes

for a given constraint and knowledge base is a complicated topic, and is not covered in detail in this thesis. However, it is useful to discuss some of the basic methods for choosing a pivot node.

One method is to pick the least common node. This requires some preprocessing, in order to determine the number of each type of node, which would increase the processing time by a factor proportional to n. The advantage to this option is that it potentially leads to the least number of checks being done. The disadvantage is that it does not take the structure or meaning of the knowledge base subgraph into account, and therefore is more likely to result in inconsistencies being missed. As the processing time required to match knowledge base subgraphs to constraints and determine pivot nodes increases with the the complexity of the constraint, it is sometimes advantageous to use the node that would result in the least complex pivot hierarchy as the primary.

If the knowledge base graphs to be checked are represented in a tree-like form, where the direction of the arrows between nodes is relevant, as with the item/attribute pairs used in most of the examples in this thesis, a tree-climbing algorithm can be used to find the top of the tree. Since it can be assumed that the top of the tree is the item, the decision can then be made whether to use the item itself or one of the attributes as the pivot. It is possible to have item/attribute precedence built into the constraint, using a relation attached to the pattern graph. It should be noted that in conceptual graphs, connections between nodes are considered bidirectional, and the direction of the arrows represented in the graph are arbitrary. For a tree-climbing method to be useful, the direction of the arrows must be taken into account.

Once a subset of knowledge-base nodes to be checked has been found, it is checked for consistency with the constraint. The procedure for this was described in the previous chapter.

After a subset has been determined to be consistent or inconsistent, it must be marked. Marking the individual nodes is not sufficient, because a node can belong to more than one subset. If the subset was found to be inconsistent, it is not always possible to determine which node in a subset caused it to be inconsistent. The algorithm needs a way to mark a subset as a whole. One possibility is maintain a database of all subsets checked and their status, and compare a subset to the contents of the database before deciding

whether or not to check it. This is not desirable, because it could potentially require $O(2^n)$ storage space, and 2^n database lookups per subset. Another possibility is to assume that a node can be a pivot node for exactly one subset, and mark the pivot node. This requires the assumption that each pivot node can only have one possible subset that matches the constraint, and this may not be the case.

The algorithm developed here will be analyzed based on the time to check a single constraint relative to the size of the knowledge base. It is assumed that the time to check a subgraph once it has been identified occurs in unit time. At present, the time needed to find and identify a subgraph is not taken into account, but it may be the subject of future research. The size of the knowledge base will be determined by the number of individual concepts that will need to be checked in order to determine whether or not a violation of a constraint has occurred. Relations and identity lines between concepts are not taken into account. The three constraint types defined above, cardinality, mutual exclusion and allowed values will be analyzed separately. This part of the analysis is expected to take about two weeks.

As part of the analysis, the process of checking a sample knowledge base will be walked through step by step. This sample will include all types on constraints, and graphs that are consistent and inconsistent with these constraints. The results of this walkthrough should show that the algorithm detects all the inconsistencies present, and does not mark consistent graphs inconsistent.

CHAPTER V

ALGORITHM

The algorithm to be used is below.

Notation:

- Pivot nodes are represented as a set *(L,N,N2,R,K)*, where *L* = the pivot level, *N* = the node type that the pivot must be an instance of, *N2* is the pivot node of level *(L-1)* that it is connected to, and *R* = the relationship between *N* and *N2*, and *K* is a set of knowledge base nodes that are valid instances of this pivot. *K* is reset to null whenever a new instance of the primary pivot is found, to avoid cross-contamination. The pivot level of the primary pivot node is 1, the level for secondaries is 2, etc. *N2* is a null instance for the primary pivot.
- Members of *K*, described above, take the form *(p1,p2)*, where *p1* is the knowledge base node that is an instance of the pivot node, and *p2* is the parent pivot node of *p1* that must be a valid pivot in order for *p1* to be a valid pivot. This is necessary to keep track of the hierarchy of the knowledge base nodes and avoid situations like the one shown in Figure 4.4.
- *L(c)* refers to the pivot level of node *c*.
- *N(c)* refers to the node instance
- *N2(c)* refers to the pivot node of level *(L(c) - 1)* that node *c* is connected to.
- Pattern(*g*) refers to the pattern subgraph of constraint graph *g*.
- Constraint(*g*) refers to the constraint subgraph of constraint graph *g*.

Functions:

P_Pivot(S):

This is the function that determines the primary pivot for a given constraint, It takes a set of possible pivots *S* as input and returns the node in S that is determined to be the primary pivot node. For the purposes of this algorithm, it is assumed to run in unit time. As the primary pivot or algorithm for finding it is assumed to be determined beforehand by the constraint's creator, this function is basically a placeholder.S_Pivot(c, S):

This function finds non-primary pivot nodes connected to a specific ancestor. It takes a pivot node *c* and a set of possible pivot nodes *S* as inputs, determines which members of S have a direct relationship to *c*, and returns a set of pivot nodes of pivot level $L(c) + 1$. To find secondary pivots, call S_Pivot with the primary pivot as *c*, tertiary pivots can be found by calling S_Pivot with a secondary pivot as *c*, etc.

get_children(p, S):

This function is used by S_Pivot to traverse the constraint pattern graph and find child pivots of *p*. It takes a starting node *p* and a set of possible pivots *S* as input, and recursively travels down the tree until it finds either a leaf node or an instance of a pivot node in *S*.

check_nodes(a,b)

This function takes knowledge base node a and a constraint node b as inputs and returns True if the nodes connected to b is a subset of the nodes connected to a. This checks all descendant nodes, pivots or non-pivots. K_Pivot(c,P):

This function takes a knowledge base node *c* and a constraint graph pivot node *P* as inputs. It returns True if *c* is a possible instance of *P*, and False if it is not. It can be called recursively on child nodes of *c* to find subpivots.

Find_Subsets(c,k):

This function takes a constraint graph pivot node *c* and a knowledge base node *k*, such that k is a valid instance of *c*, as inputs and finds a set of sets of knowledge base nodes that consist of one valid pivot node in the knowledge base for each child pivot of *c*.

```
// This is the main body of the algorithm. It iterates through the knowledge base, looking
// for instances of the primary pivot, calls functions to check for a match to the pattern
// and find all possible instances of non-primary pivots, and checks each subset of pivots
// for consistency.
Let G = set of all constraints in the Knowledge Base.
Let K = set of all knowledge base nodes
For all constraints g in G:
        // First, find all pivot nodes
        Let S = set of all nodes in Pattern(g) with identity lines to nodes in
          Constraint(g)
        // Determine the primary pivot.
        // The process depends on the structure of the constraint and KB.
        Let p = P_Pivot(S)
        Let P = {p} // P is the set of all pivot nodes assigned a place in the hierarchy
        Let L = 1 // The pivot level of the current node
        //Determine pivot node hierarchy
        While P is not equal to S: // While there are pivot nodes still unassigned
                // Find all pivot nodes of level L
                Let P2 = set of all members x of P such that L(x) = L
                // For each member of the set, find all its children
                For all members x of P2:
                        // It is possible for a pivot node of level n to be a possible child
                        // of multiple level n-1 nodes, but can only have 1 place in the
                        // hierarchy. The if condition is necessary to prevent a node
                        // being assigned as the child of multiple parents.
                        If P is not equal to S:
                                Let T = S_Pivot(x,(S - P))
                                Let P = P + T
                // Once all the level L nodes are assigned, go to level L+1
                // If any of the level L pivots have child pivots, the S_Pivot call will
                // have assigned them a level of L+1.
                Let L = L + 1
        // Find instances of the primary pivot
        For all nodes k in K:
                If k is an instance of N(p):
                        // Reset set of possible instances
                        For all pivot nodes c in P:
                                Let K(c) = empty set
                        // Determine if k is a valid instance of p
                        if K_Pivot(k,p) = True:
                                // If k is an instance of p, find all valid pivot
                                // node subsets
                                // The subset is initialized to contain only the primary
                                K(p) = K(p) + {(k, null)}
                                Let J = Find_Subsets(k,p)
```

```
                                        // Once subsets are found, check each one,
                                          // and mark appropriately
                                        For each subset j in J:
                                                For each pivot node n in P:
                                                        Let m = the member of k such
                                                          that m is also a member of K(n)
                                                        // A node cannot be connected to
                                                          // more than one pivot. If a node
                                                          // has been selected as an instance
                                                          // for two pivots at the same time,
                                                          // the subset is not valid.
                                                        If m does not already have an
                                                          identity line:
                                                                Draw an identity line
                                                                  between n and m
                                                        else:
                                                                Subset is invalid
                                                If Subset is not invalid:
                                                        Test subgraph for truth
                                                        If subgraph tests as False:
                                                                Mark subgraph
                                                                  Inconsistent
                                                For each pivot node e in P:
                                                        Erase all identity lines.
        End.


// S_Pivot  is used to organize the pivot nodes in the pattern graph into a hierarchy. It takes
// a single pivot node and returns a pivot node hierarchy with that node at the top.
Function S_Pivot (p, S):
        Return_Pivots = get_children (p, S)
        For each member r of Return_Pivots:
                Let L(r) = L(p) + 1
                Let R(r) = relationship between r and p
                Let N2(r) = p
        return Return_Pivots

// get_children is used to find child pivots of a given pivot. It takes a single pivot node and returns
// a set of its children.
Function get_children(p, S):
        if p is a member of S:
                return {p} // Pivot found
        Let S2 = set of all nodes with arrows from p
        if S2 is empty:
                return {} // Non-pivot leaf node found
        else:
                For each member s of S2:
                        Return_Set = Return_Set + get_children(s,S)
        return Return_Set

        // K_Pivot is used to determine if a given KB node is a valid instance of a particular
        // pivot node. It is called recursively on child pivots.
        Function K_Pivot(c,p):
                // If the pattern graph does not match the knowledge base graph, this
```

```
        // is not a valid pivot
    If check_nodes(c, N(p) ) = False:
            Return False
    Let S1 = set of all pivot nodes x in P such that L(x) = L(p) + 1 and
      N2(x) = p
    // If there are child pivots in the pattern, check child nodes in knowledge
      // base for valid pivots.
      // It is assumed that at least one match in the knowledge base exists, or the
      // check_nodes would have returned False.
    For each node s in S1:
            // Find all KB nodes that may be instances of pivot node s
            Let S2 = set of all knowledge base nodes b such that b is an instance
              of N(s) and Relationship(b,c) = R(s)
            For each node d in S2:
                    // Determine if KB node d is a valid instance of pivot s
                    if K_Pivot(d,s) = True:
                            // if KB node d is a valid instance of pivot s,
                            // add it to s's set of possible instances.
                            K(s) = K(s) + {(d,c)}
    Return True

// check_nodes is used to determine if the KB subgraph is a match to the pattern
// subgraph. It takes a pattern graph node and a KB node as input, and returns True is
// the pattern graph's children are a subset of the KB node's children. This checks both
pivot and non-pivot nodes.
Function check_nodes(a,b):
    // In order to detect nodes connected in a cycle, identity lines are drawn
    // between pattern graph nodes and their matching KB nodes.
    // If a has an identity line, then there is a cycle in the pattern.
    If there is an identity line from a:
            If b is connected to a through an identity line:
                    // The KB nodes match the cycle.
                    Return True
            Else:
                    // The cycle is not matched.
                    Return False
    // If there is no identity line between the two nodes, draw one.
    Draw an identity line between a and b
    Let A = set of all nodes with arrows from a
    Let B = set of all nodes with arrows from b
    While B is not empty:
            Let b1 = the next node in B
            Let A2 = all nodes a1 in A such that a1 = b1 and relationship(a1)
              = relationship(b1)
            Let Found = False
             For each node a1 in A2 and while Found = False:
                    If check_nodes(a1, b1) = True:
                            Remove the identity line between a and b
                            Found = True
            If Found = False
                    Remove the identity line between a and b
                    return False
            Let A = A - a1
            Let B = B - b1
```

```
        Remove the identity line between a and b
        Return True

// Find_Subsets is used to find all possible combinations of KB nodes that are valid
// matches to the pattern graph's pivot hierarchy. In order for a node in the KB to be
// a valid pivot in a particular subset, its parent must also be in the subset.
Function Find_Subsets (c,k):
        Let Return_Set = empty set
        // If c has no child pivots, return a set containing only  k
        Let W = set of all pivot nodes x in P such that N2(x) = c
        If W is empty:
                Return_Set = {{k}}
        else:
        // If there are child pivots, find the subsets for each child
                For each node w in W:
                        Let K2 = set of all members y of K(w) such that the
                          relationship between y and c = R(w).
                        For each node q in K2:
                                Child_Sets = Find_Subsets(q,w)
                                w_sets = w_sets + Child_sets
                        // w_sets is all possibilities for pivot w and its children
                        if Return_set is empty:
                                Return_set = w_sets
                        else:
                                // Return_sets has all possibilities for all pivots
                                // searched before w. Each of these sets must be
                                // paired with each of w's sets.
                                For each set q1 in Return_sets:
                                        For each set q2 in w_Sets:
                                                New_Set = q1  q2
                                                New_Return =
                                                        New_Return + New_Set
                                Return_sets = New_Return
                For each set s in Return_Sets:
                        s = s + k
        return Return_Sets
```

CHAPTER VI

ANALYSIS

When analyzing the performance of this algorithm, there are four variables to be considered: Rule base size, constraint complexity, knowledge base size and knowledge base complexity. “Rule base size” refers to how many constraints exist in the knowledge base rules, regardless of structure or size. “Constraint complexity” refers to the complexity of the structure of a given constraint. This has two parts- number of levels in the pivot node hierarchy tree, and how many pivot nodes are in a given level. “Knowledge base size” refers to the number of individual nodes in the knowledge base. “Knowledge base complexity” refers to how many potential subordinate pivot nodes a given primary pivot node has.

There are also two distinct processes to be considered- analyzing a constraint to determine its pivot node hierarchy and searching the knowledge base for subsets to test against a given constraint. First, the process of determining the pivot node hierarchy for a constraint will be analyzed. This process is performed once per constraint, regardless of knowledge base contents, to the size and complexity of the knowledge base does not need to be considered in this step. The following variables will be used:

- T = the maximum distance from the primary pivot to the farthest leaf node, pivot and non-pivot, in a constraint
- P = the number of pivot nodes in the largest constraint in the rule base
- K = the maximum number of child nodes, pivot or not, that any node in the a constraint has
- C = number of constraints in the rule base

The following code determines the pivot hierarchy:

```
1. Let S = set of all nodes in Pattern(g) with identity lines to nodes in Constraint(g)
2. Let p = P_Pivot(S)
3. Let P = {p}
4. Let L = 1
5. While P is not equal to S:
6.         Let P2 = set of all members x of P such that L(x) = L
7.         For all members x of P2:
8.                 If P is not equal to S:
9.                         Let T = S_Pivot(x,(S - P))
10.                        Let P = P + T
11.                        Let L = L + 1
```

Lines 1-4 are executed once per constraint.

The loop that starts at line 5 and ends at line 11 runs until all the pivot nodes in the constraint have been assigned a place in the hierarchy. If only one node is removed from the temporary set per iteration, the loop runs a maximum of *P* times.

Line 6 executes once per loop iteration.

The loop that starts at line 7 and ends at line 11 executes once for each child pivot, with a maximum of *P* per outer loop interation, *P^2* total. However, in a real knowledge base, actually going through *P^2* iterations would be unlikely, if not impossible. In order for the outer loop to iterate *P* times, only one node must be assigned a place in the hierarchy per iteration, and therefore, the inner loop would only execute once. In general, more iterations of the out loop means fewer iterations of the inner loop.

Lines 9-11 execute once per inner loop iteration, for a maximum of *P^2*. The S_Pivot call on line 9 is analyzed below. In addition, the entire loop executes *P* times. At this point, the performance of the constraint analysis can be described as follows:

C (P(4 + P (1 + P (2 + <S_Pivot>))))

The code for the S_Pivot function is below:

```
1. Return_Pivots = get_children (p, S)
2. For each member r of Return_Pivots:
3.         Let L(r) = L(p) + 1
4.         Let R(r) = relationship between r and p
5.         Let N2(r) = p
```

6.Return Return_Pivots

Line 1 is executed once. The performance of the get_children function is analyzed below.

The loop that starts on line 2 and ends on line 5 executes once for each member of the return set. The return set contains the child nodes of the pivot node being analyzed. It cannot contain more than *P* nodes. A node cannot have itself or any of its ancestors as children, so it will always contain less than *P*. The lower the parent node is in the hierarchy, the smaller this set will be.

Line 6 is executed once.

At this stage, the performance for S_Pivot can be described as follows:

<get_children> + 3P + 1

This makes the performance for the constraint analysis:

C (P(4 + P (1 + P (2 + (<get_children> + 3P + 1)))))

The code for the get_children function is as follows:

```
Function get_children(p, S):
        if p is a member of S:
                return {p}
        Let S2 = set of all nodes with arrows from p
        if S2 is empty:
                return {}
        else:
                For each member s of S2:
                        Return_Set = Return_Set + get_children(s,S)
        return Return_Set
```

get_children is a recursive function. However, it does not need to be analyzed as one. Its function is to determine the position in the pivot tree of each pivot in the pattern graph, and therefore must touch each node in the tree at least once. Each node is touched at most twice- once as a “child” pivot and once as a “parent” pivot. In the worst-case scenario, every node in the pattern graph is a pivot. Assuming a maximum of *T* levels in the tree and *K* children for each non-leaf node, the maximum number of “touches” is proportional to the maximum number of nodes in the constraint graph, which can be represented as

$$2(K^T + K^{(T-1)} + ... + K^2 + K + 1).$$

As $(K^{(T-1)} + ... + K^2 + K + 1)$ can never be greater than K^T, we can simplify the representation by using $2K^T$ as an upper bound for the maximum number of nodes, making the maximum number of "touches" performed by the get_children function $4K^T$. Using this, we can state the constraint analysis portion's performance as

$$C(P (4 + P (1 + P (2 + (4K^T + 3P + 1))))).$$

After simplifying the statement, it becomes

$$C((P^3)(4K^T) + 3P^4 + 3P^3 + P^2 + 4P).$$

The number of nodes in a constraint serves as an upper bound for the number of pivot nodes in the constraint, so we can use $2K^T$ as an upper bound for P. After this substituting $2K^T$ for P and simplifying the statement, it becomes

$$C(112K^{4T} + 24K^{3T} + 4K^{2T} + 8K^T).$$

In summary, the performance of the constraint analysis portion of the algorithm is polynomial with respect to the the maximum number of child nodes, exponential with respect to the maximum distance from the primary pivot to the farthest leaf node, and linear with respect to number of constraints. It is not affected by the size of the knowledge base.

For the subset search portion of the algorithm, the following variables will be used:

- T = the maximum distance from the primary pivot to the farthest leaf node, pivot and non-pivot, in a constraint
- P = the number of pivot nodes in the largest constraint in the rule base
- K = the maximum number of child nodes, pivot or not, that any node in the a constraint has
- C = number of constraints in the rule base
- B = number of nodes in the knowledge base
- J = the maximum number of valid pivot node sets for any primary pivot node instance

- L = the maximum number of child nodes for any knowledge base node.

It should be noted that the value of *L* has no relation to *K*. It is possible for a knowledge base subgraph to have multiple nodes, or no nodes at all, corresponding to any given node in a constraint. It is assumed that *L* is greater than or equal to *K*. Otherwise, the constraint would have no matches in the knowledge base. It should also be noted that there is no variable for the tree depth of a knowledge base subgraph. No matter how many levels a knowledge base subgraph has, the algorithm is only interested in the part that matches the constraint's graph. Therefore, it will check no more than *T* levels in the knowledge base, no matter how deep the graph goes.

```
The following code searches for matching subsets for a given constraint:
1. For all pivot nodes c in P:
2.          Let K(c) = empty set
3. if K_Pivot(k,p) = True:
4.          K(p) = K(p) + {(k, null)}
5.          Let J = Find_Subsets(k,p)
6.          For each subset j in J:
7.                    For each pivot node n in P:
8.                              Let m = the member of k such that m is also a
                                  member of K(n)
9.                              If m does not already have an identity line:
10.                                       Draw an identity line between n and m
11.                             else:
12.                                       Subset is invalid
13.                   If Subset is not invalid:
14.                             Test subgraph for truth
15.                             If subgraph tests as False:
16.                                       Mark subgraph Inconsistent
17.                             For each pivot node e in P
18.                                       Erase all identity lines.
```

This section is executed once per primary pivot instance per constraint. In a worst-case scenario, every node in the knowledge base is a pivot node instance for every constraint in the rule base. This scenario is unlikely, but it is possible, given the right constraint and knowledge base structure.

Lines 1 and 3 are executed for all instances. The loop beginning on line 4 is only executed if the instance is found to be a valid primary pivot. In the worst-case scenario, it will be executed for all instances, so we will assume that is the case. Lines 4 and 5 are executed once. The loop beginning at line 7

is executed once for each subset found in the Find_Subsets call in line 5, for a maximum of *J* iterations.

The loop beginning at line 8 is executed for every pivot node in the constraint, for a maximum of *P* iterations. Either line 10 or line 12 is executed, but not both. The loop starting on line 14 is only executed if the subset was not marked as invalid in the previous loop. In a worst-case scenario, no subsets will be marked invalid, so we can assume it will be executed for all subsets. The test on line 14 is assumed to be performed in unit time. Line 16 is only executed if the subgraph tests as false. In a worst-case scenario, all subgraphs would test as false, so we can assume it executes for all. Line 18 executes once for every pivot node, for a maximum of P. Therefore, the performance of the constraint match segment of the algorithm can be described as

$$C(4 + \langle K\_Pivot \rangle + \langle Find\_Subsets \rangle + J(4P + 3)).$$

The K_Pivot function takes a knowledge base node and a constraint pivot node as inputs, and determines if the knowledge base node in question is a valid instance of the constraint pivot. If it is determined to be a valid instance, the function finds all possible instances of sub-primary pivots. At least one possible match must be found for every mode in the pattern graph for the primary pivot instance to be considered a valid instance. It is possible for a constraint graph node to have multiple matches in the knowledge base, but these will be dealt with by the Find_Subsets function.

The code for the K_Pivot function is as follows:

```
1. Function K_Pivot(c,p):
2.         If check_nodes(c, N(p) ) = False:
3.                 Return False
4.         Let S1 = set of all pivot nodes x in P such that L(x) = L(p) + 1 and
             N2(x) = p
5.         For each node s in S1:
6.                 Let S2 = set of all knowledge base nodes b such that b is an instance
                     of N(s) and Relationship(b,c) = R(s)
7.                 For each node d in S2:
8.                         if K_Pivot(d,s) = True:
9.                                 K(s) = K(s) + {(d,c)}
10.        Return True
```

K_Pivot is a recursive function, containing another recursive function, check_nodes. The number of iterations for both functions is limited by the size of the constraint currently being checked. The loop

beginning on line 5 executes once for every child pivot of the constraint pivot node passed to the function. Assuming a maximum of *K* child nodes for each node in a constraint and a worst-case scenario of every child node being a pivot, the loop executes *K* times for each non-leaf pivot. The loop starting on line 7 is executed once for each knowledge base node that is a potential match for the constraint pivot being analyzed in that loop iteration. In order to be a match, a knowledge base node must be of the same type and have the same relationship to its parent node as the constraint pivot. The relationship between a parent pivot and child pivot can be a single relationship, or a chain of several. For simplicity's sake, this analysis assumes a single relationship. In a worst-case scenario, every child knowledge base node is a potential match for every child pivot. This scenario is unlikely, but is possible, given the right constraint structure. Given a maximum of *L* child nodes for every knowledge base node, the performance of K_Pivot for a constraint node of level *X* can be described as follows:

$$K\_Pivot(X) = check\_nodes(X) + 1 + K(1 + L(K\_Pivot(X-1))).$$

In order to determine the performance of K_Pivot, it is first necessary to do the same for check_nodes. Check_nodes takes a constraint graph node (not necessarily a pivot) and a knowledge base node and determines if they are a match. If the constraint node's children are a subset of the knowledge base node's children, they are considered a match. It is only concerned with finding one matching subset in the knowledge base, and is not concerned with how many different subsets there may be. Once it has found a matching subset, it returns True without checking anything more. In a worst-case scenario, it will have to check all the knowledge base node's children before deciding whether the nodes are a match.

The code for check_nodes is as follows:

```
Function check_nodes(a,b):
1.      If there is an identity line from a:
2.          If b is connected to a through an identity line:
3.              Return True
4.          Else:
5.              Return False
6.      Draw an identity line between a and b
7.      Let A = set of all nodes with arrows from a
8.      Let B = set of all nodes with arrows from b
9.      While B is not empty:
```

```
10.             Let b1 = the next node in B
11.             Let A2 = all nodes a1 in A such that a1 = b1 and relationship(a1)
                  = relationship(b1)
12.             Let Found = False
13.              For each node a1 in A2 and while Found = False:
14.                     If check_nodes(a1, b1) = True:
15.                             Remove the identity line between a and b
16.                             Found = True
17.             If Found = False
18.                     Remove the identity line between a and b
19.                     return False
20.             Let A = A - a1
21.             Let B = B - b1
22.     Remove the identity line between a and b
23.     Return True
```

If the knowledge base node already has an identity line to a constraint node, meaning there is a cycle in the graph, the function returns a value immediately. In a worst-case scenario, there will be no cycles and lines 2-5 will be skipped. If there is no cycle, a new identity line is drawn and the sets of children for both the knowledge base node and the constraint node are compiled. The loop beginning on line 9 is executed once for every child node, for a maximum of *K* iterations. The loop beginning on line 13 is iterated once for every knowledge base node that matches the current constraint node, until either a match is found or there are no more knowledge base nodes. In a worst-case scenario, it will need to iterate through every node, every time, for a maximum of *L* iterations. The performance of check_nodes for a constraint node of level *X* can be approximated as follows:

$$\text{check\_nodes}(X) = 5 + K(5 + L(\text{check\_nodes}(X-1))).$$

In order to find a general formula for this, we can calculate the performance for the first few values of *T*:

$\text{check\_nodes}(1) = 5K + 5$
$\text{check\_nodes}(2) = 5 + K(5 + L(5K + 5)) = 5 + 5K + 5LK + 5LK^2$
$\text{check\_nodes}(3) = 5 + K(5 + L(5K + 5)) = 5 + 5K + 5LK + 5LK^2 + 5(L^2)(K^2) + 5(L^2)(K^3).$
...

Following the apparent pattern, the performance of the check_nodes function for the maximum constraint tree depth of *T* can be approximated as follows:

$$5(L^0)(K^0) + 5(L^0)(K^1) + 5(L^1)(K^1) + 5(L^1)(K^2) + ...$$

$$+ 5(L^{(T-1)})(K^{(T-1)}) + 5(L^{(T-1)})(K^{T}).$$

We can use the following formula as a upper-bound approximation:

$$10(L^{(T-1)})(K^{(T-1)} + K^{T}).$$

Now that we have an approximation for the performance of check_nodes, we can finish analyzing K_Pivot.:

$$K\_Pivot(X) = 10(L^{(X-1)})(K^{(X-1)} + K^{X}) + 1 + K(1 + L(K\_Pivot(X-1))).$$

Since the check_nodes performance for nodes of level *T* is the maximum, we can simplify the analysis by using this as an upper bound and treating it as a constant. Now, we must calculate the first few values for the performance of K_Pivot in order to find a general formula. For the sake of simplicity, we will use the constant *H* to represent the performance of check_nodes.

$K\_Pivot(1) = (L^{0})(K^{1}) + (1+H)$
$K\_Pivot(2) = (L^{1})(K^{2}) + (L^{0})(K^{1}) + (1+H)(LK + 1)$
$K\_Pivot(3) = (L^{2})(K^{3}) + (L^{1})(K^{2}) + (L^{0})(K^{1}) + (1+H)((L^{2})(K^{2}) + LK + 1)$
...
$K\_Pivot(X) = (L^{(X-1)})(K^{X}) + ... + (L^{0})(K^{1}) + (1+H)((L^{(X-1)})(K^{(X-1)}) + ... + LK + 1)$

We can use the following approximation for an upper bound:

$$2(L^{(X-1)})(K^{X}) + (2+2H)(L^{(X-1)})(K^{(X-1)}).$$

The last function to analyze is Find_Subsets. This function takes the sets of valid pivot nodes instances found by K_Pivot and determines which combinations make up valid sets. It starts at the primary pivot, and is called recursively for each child pivot and valid knowledge base node that matches it. It is possible for this function to be called multiple times for each pivot node, if there are multiple matches in the knowledge base. The same is true from knowledge base nodes, if they are a match for more than one pivot. Although a knowledge base node can only be matched with one pivot per subset, it can be matched with different ones in different subsets. The sets are constructed by adding the parent node to each of the returned sets from the recursive calls. The code for Find_Subsets is as follows:

```
Function Find_Subsets (c,k):
1.      Let Return_Set = empty set
2.      Let W = set of all pivot nodes x in P such that N2(x) = c
3.      If W is empty:
4.              Return_Set = {{k}}
5.      else:
6.              For each node w in W:
7.                      Let K2 = set of all members y of K(w) such that the
                          relationship between y and c = R(w).
8.                      For each node q in K2:
9.                              Child_Sets = Find_Subsets(q,w)
10.                             w_sets = w_sets + Child_sets
11.                     if Return_set is empty:
12.                             Return_set = w_sets
13.                     else:
14.                             For each set q1 in Return_sets:
15.                                     For each set q2 in w_Sets:
16.                                             New_Set = q1  q2
17.                                             New_Return =
                                                  New_Return + New_Set
18.                             Return_sets = New_Return
19.             For each set s in Return_Sets:
20.                     s = s + k
21.     return Return_Sets
```

First, the function initializes the return set and compiles a set of all child pivots (not necessary child nodes) of the pivot node it received as input. If there are no child pivots, it returns a set containing only the knowledge base node it received. If there are child pivots, the loop beginning on line 6 is entered. It executes once for every child pivots, for a maximum of *K* iterations. For every child pivot, a set of knowledge base nodes determined to be valid instances of it is compiled. The loop beginning on line 8 executes for every node in this set. At worst, every child node of the passed knowledge base node will be a valid instance, for a maximum of *L* iterations. Once the possible subsets for a particular child pivot are found, these sets are combined with the sets for the previous child pivots, if any exist. Assuming a maximum of *J* subsets, this executes a maximum of *J-1* times. After all these combined sets are compiled, the parent node is added to each set, executed a maximum of *J* times. After this, the set of sets is returned.

$$\text{Find\_Subsets}(X) = 2 + K(1 + L(\text{Find\_Subsets}(X-1) + J)) + J$$
$$\text{Find\_Subsets}(0) = 3$$
$$\text{Find\_Subsets}(1) = (J+3)(L^1)(K^0) + (L^0)(K^1) + J + 2$$
$$\text{Find\_Subsets}(2) = (J+3)(L^2)(K^2) + (2J+2)(L^1)(K^1) + (L^1)(K^2) + (L^0)(K^1) + J + 2$$
$$\text{Find\_Subsets}(3) = (J+3)(L^3)(K^3) + (2J+2)(L^2)(K^2) + (2J+2)(L^1)(K^1) + (L^2)(K^3) + (L^1)(K^2) + (L^0)(K^1) + J + 2$$
...

$$\text{Find\_Subsets}(T) = (J+3)(L^{T})(K^{T}) + (2J+2)(L^{(T-1)})(K^{(T-1)}) + \ldots + (2J+2)(L^{1})(K^{1}) + (L^{(T-1)})(K^{T}) + \ldots + (L^{0})(K^{1}) + J + 2$$

We can use the following approximation as an upper bound:

$$(2J+6)(L^{T})(K^{T}) + 2(L^{(T-1)})(K^{T}).$$

Now that we have determined the performance of all the sub-functions, the performance of the subset search portion of the algorithm can be represented as follows:

$$C(4 + 2(L^{(T-1)})(K^{T}) + (2+2H)(L^{(T-1)})(K^{(T-1)}) + (2J+6)(L^{T})(K^{T}) + 2(L^{(T-1)})(K^{T}) + J(4P + 3)).$$

This statement can be simplified somewhat. The variable H was used to substitute for the maximum performance time of the check_nodes function. An upper-bound approximation for P, $2K^{T}$, was determined earlier. If we assume the maximum size for the constraint pattern and the maximum number of children for knowledge base nodes, J can be approximated by $2(L^{T})(K^{T})$. After making these substitutions and simplifying, the performance of the algorithm can be described as follows:

$$C[4 + 2(L^{2T})(K^{2T}) + 40(L^{(2T-2)})(K^{(2T-2)}) + 12(L^{T})(K^{T}) + 4(L^{(T-1)})(K^{(T-1)}) + 40(L^{(2T-2)})(K^{(2T-1)}) + 16(L^{T})(K^{2T}) + 4(L^{(T-1)})(K^{T})$$

.

To simplify the formula further, we can use L as an upper bound for K. If K is greater than L, that means there exists a constraint in which one of the pattern graph nodes has more children than any knowledge base node. Such a constraint would make no sense, as it would have no possible matches anywhere in the knowledge base. After substituting L for K and simplifying, the performance can be described as

$$C[4 + 2L^{4T} + 40L^{4T-4} + 4L^{2T-2} + 12L^{2T} + 40L^{4T-3} + 16L^{3T} + 4L^{2T-1}].$$

In summary, the variables that determine the performance of this algorithm are the number of nodes in the knowledge base, the number of constraints, the maximum number of children a node can have, and the maximum depth of a constraint's pattern graph. This algorithm was developed under the assumption that the performance would be polynomial with respect to knowledge base size. This appears to be true. If we assume all other variables to be constant, it is indeed polynomial, indeed linear, with respect to the number of nodes in the knowledge base, and polynomial with respect to the number of children each node

has. However, it is exponential with respect to the depth of the constraint pattern. The depth of any subgraph in the knowledge base does not affect the performance, as the number of levels checked is limited by the depth of the constraint pattern.

What this means for knowledge base creators is that they should design their knowledge base and constraints to minimize the depth of the constraint graph trees. A pivot with a large number of child nodes is faster to process than a multiple-level tree with a small amount of nodes in each level. In the pattern graph example in Figure 4.2, using the Color node as the primary pivot would be more efficient than using the Widget or Scarlet nodes, as the graph becomes a two-level tree. If a given pivot has two subtrees, one deeper than the other, the designer should structure the constraint so that the shorter subtree is checked first.

One possible way to improve the performance of the algorithm is to allow multiple constraint subgraphs to be connected to a given pattern subgraph. Although the algorithm would still need to determine the pivot hierarchy for each constraint subgraph individually, this would eliminate redundant searches for the same knowledge base subgraph.

This algorithm was developed on the assumption that there were no restrictions on the structure of the knowledge base graphs. If there are some known limitations, it may be possible to take advantage of them to develop more efficient versions. If it impossible- not merely inconsistent- for a given situation to exist, there is no reason for the algorithm to check for it. For example, if a given node cannot have more than one of a certain relationship, the algorithm could be altered to stop checking for other instances of this relationship once it finds one, no matter how many other child nodes remain.

It may also be possible to do the opposite, and design the knowledge base with a particular search strategy in mind. Limiting the depth of the knowledge base subgraphs is an obvious step, but there are other possibilities. For example, a designer might specify that child nodes must be arranged alphabetically, enabling the algorithm to take advantage of this to speed up checking for the existence of a particular node. A knowledge base structured as a binary tree could use a binary search to find matches.

A search algorithm specific to a given knowledge base structure can determine matches faster than the general algorithm presented here. There is still plenty of work to be done in developing these specialized algorithms. More research can be done in determining the ideal search strategies for given

knowledge base structures. Now that we have a general algorithm, the next step in the process could be to develop algorithms optimized for general classes of knowledge base structures. This could lead to developing generalized methods for determining the most efficient search for a given knowledge base.

CHAPTER VII

CONCLUSION AND FUTURE WORK

This algorithm is designed to test individual knowledge base pieces for consistency with constraints, and locate subsets that are inconsistent- that is, subsets that describe a situation that cannot exist, according to the knowledge base's rules. However, it may be the case where there are constraints meant to apply to the knowledge base as a whole, rather than subsets. For example, there may be a limit on how many of a particular node may exist, regardless of what other nodes they may be connected to. The only way this algorithm can check a constraint of this nature is by representing this constraint with the entire knowledge base in the pattern subgraph. This is not possible unless the entire structure is known beforehand, and it would be very costly, timewise, to analyze. A possible solution to this problem is to have special constraints which contain global information. The same constraint structure already used, with a pattern to match and a rule to check it against, could be modified to include actors in the rule portion, which would update the global information.

Once inconsistencies are found, the next step is to figure out what to do with them. Depending on the structure and functionality of the knowledge base, it could either work around them or try to fix them. Arieli1, Hunter1 and Tahara1 describe methods using possibilistic logic or multi-valued logic to allow a knowledge base to work around an inconsistency without causing a contradiction. It may be possible to adapt these methods to conceptual graphs. Fixing them could involve deleting nodes, deleting links between nodes, or altering nodes, but there is nothing inherent in the constraint structure that says which to do. There would need to be another set of rules to determine which nodes to alter or delete, and what to do

to them. A constraint could have a third component which would contain directions for how to deal with an inconsistency if one is found.

In some cases, there may need to be an intermediate step to determine the "cause" of the inconsistency before the algorithm can decide how to deal with it. For example, if a node is found to have too many of a certain type of child node, it may be necessary to determine which are known or assumed to be valid children and which are "extras." A third set of rules could be added to determine the source of the inconsistency.

REFERENCES

[Arieli1] Arieli, O., An Algorithmic Approach to Recover Inconsistent Knowledge Bases, *Logics in Artificial Intelligence: European Workshop*, Oct. 2000.

[Dubois1] Dubois, D., Konieczny, S., Prade, H., Quasi-Possibilistic Logic and its Measures of Information and Conflict, *Fundamenta Informaticae 57*, 2003, pp. 101-125.

[Hunter1] Hunter, A., Reasoning with Inconsistency in Structured Text, *The Knowledge Engineering Review*, Vol. 15, Issue 4, Dec. 2000, pp. 317-337.

[Hunter2] Hunter, A., Measuring Inconsistency in Knowledge via Quasi-classical Models, *Eighteenth National Conference on Artificial Intelligence*, pp. 68-73, 2002.

[Liu1] Liu, W., Qi, G., Bell, D., Adaptive Merging of Prioritized Knowledge Bases, *Fundamenta Informaticae 73*, 2006, pp. 389-407.

[Liu2] Liu W., Easterbrook, S., Mylopoulos. J., Rule-Based Detection of Inconsistency in UML Models, *Fifth International Conference on the Unified Modeling Language*, Oct. 2002.

[Mineau1] Mineau, G., Representing and Enforcing Interaction Protocols in Multi-Agent Systems: An Approach Based on Conceptual Graphs, *Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology (IAT'03).*

[Morrell1] Morell, L., Use of Metaknowledge in the Verification of Knowledge-based Systems, *Proceedings of the 1st international conference on Industrial and engineering applications of artificial intelligence and expert systems*, Volume 2, pp. 847-857.

[Salomons1] Salomons, O., Van Slooten, F., Van Houten, F., Kals, H., Conceptual Graphs in Constraint-Based Redesign, *Solid Modeling 95*, pp. 55-64 [*].

[Sourrouille1] Sourrouille, J., Caplat, G., Constraint Checking in UML Modeling, *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering SEKE '02* , pp. 217-224.

[Sowa1] Sowa, J. F., Conceptual Structures, Information Processing in Mind and Machine, Addison–Wesley, 1984.

[Tahara1] Tahara, I., Nobesawa, S., Reasoning with Inconsistent Knowledge Base, *Systems and Computers in Japan*, Vol. 37, Issue 3, Mar. 2006, pp. 41-48.

[Zhao1] Zhao, S., Shen, S., The Consistency Problem of Knowledge Base, *Proceedings of the ACM*, 1990, pp. 745-750 [*].

[Zlatareva1] Zlatareva, N., An Integrated Approach to Quality Assurance of Expert System Knowledge Bases, *CIKM Î93*, pp. 85-94 [*].