Using Constraints to Model Disjunctions in Rule-Based Reasoning

Bing Liu and Joxan Jaffar

Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 119260, Republic of Singapore
{liub, joxan}@iscs.nus.sg

Abstract

Rule-based systems have long been widely used for building expert systems to perform practical knowledge intensive tasks. One important issue that has not been addressed satisfactorily is the disjunction, and this significantly limits their problem solving power. In this paper, we show that some important types of disjunction can be modeled with Constraint Satisfaction Problem (CSP) techniques, employing their simple representation schemes and efficient algorithms. A key idea is that disjunctions are represented as constraint variables, relations among disjunctions are represented as constraints, and rule chaining is integrated with constraint solving. In this integration, a constraint variable or a constraint is regarded as a special fact, and rules can be written with constraints and information about constraints. Chaining of rules may trigger constraint propagation, and constraint propagation may cause firing of rules. A prototype system (called CFR) based on this idea has been implemented.

1. Introduction

Rule-based systems are one of the great successes of AI (e.g., Newell 1973; Lucas & Van Der Gaag). They are widely used to build knowledge-based systems to perform tasks that normally require human knowledge and intelligence. However, there are still some important issues that have not been addressed satisfactorily in the current rules-based systems. One of them is the disjunction. This limits their problem solving power.

In the Constraint Satisfaction Problem (CSP) research, many efficient constraint propagation algorithms have been produced (Mackworth 1977; Hentenryck *et al* 1992). A number of languages or systems based on the model have also been developed and used for solving real-life problems (Jaffar & Maher 1994; Ilog Solver 1992).

In this paper, we show that some types of important disjunctions can be modeled with CSP. Thus, it is possible to use the simple representation scheme and efficient problem solving methods in CSP to handle these types of disjunctions. Specifically, the disjunctions can be represented as constraint variables and their domains. The relations among disjunctions can be represented as constraints. In this paradigm, constraint propagation and rule chaining are integrated. A constraint can be added as a special fact, and rules can be written with constraints and information about constraints. Chaining of rules may trigger constraint propagation, and constraint propagation may cause firing of rules. With the incorporation of CSP techniques, the power and the expressiveness of rule-based systems will be greatly increased. Based on this idea, a prototype system, called CFR, has also been implemented.

The idea of incorporating CSP into a logic-based system is not new. Constraint solving has long been integrated with logic programming languages such as Prolog. This integration has resulted in a number of Constraint Logic Programming (CLP) languages (Jaffar & Maher 1994), such as CLP(*R*) (Jaffar & Lassez, 1987) and Chip (Hentenryck 1989). These languages are primarily used for modeling and solving real-life optimization problems, such as scheduling and resource allocations. However, this work is different from that in CLP in a number of ways. The main difference is that CLP languages are all based on Horn clauses and backward chaining, while the proposed integration is based on forward chaining, which is suitable for solving a different class of reasoning problems. Integration of constraint solving and forward chaining has some specific problems that do not exist in CLP languages. The proposed integration is also mainly for improving reasoning capability of existing rule-based systems rather than for solving combinatorial search problems. Thus the types of constraints and their representations in the proposed approach are quite different from those in CLP languages.

We regard this work as the first step to a full integration of the CSP model with forward chaining rule-based systems. The current integration presented in this paper is still restrictive in the sense that it is mainly to help model and handle the problems with some disjunctions. A full integration could potentially change the way that people use rule-based systems and change the way that people solve practical reasoning problems, which are the main applications of the rule-based systems today. It may be just like the way that CLP languages have changed the way that people model and solve practical combinatorial search problems.

2. Rule-Based Systems and Constraint Satisfaction Problems

This section reviews rule-based systems and CSP. The coverage is by no means complete; rather the focus is on highlighting the problems with disjunctions in current rule-based systems.

2.1. Rule-Based Systems

A rule-based system consists of three main components.

1. A working memory (WM): a set of facts representing the current state of the system.
2. A rule memory (RM): a set of *IF-THEN* rules to test and to alter the WM.
3. A rule interpreter (RI): it applies the rules to the WM.

The rule interpreter repeatedly looks for rules whose conditions match facts in the WM. On each cycle, it picks a rule, and performs its actions. A rule is of the form:

IF <conditions> THEN <actions>

There are three common connectives in a rule-based system, i.e., *and, or* and *not*. We will only discuss *or* here as we are mainly interested in disjunctions. *or* in logic can be defined as *inclusive* ($\vee$) or *exclusive* ($\oplus$). Let us first look at the inclusive *or*. For example, "if something is a block or a pyramid, then it is a pointy_object" (adapted from (Charniak *et al* 1987)) can be expressed as follows:

IF *isa(?x, block)* $\vee$ *isa(?x, pyramid)*
THEN *add(isa(?x, pointy_object))*

where *?x* is a variable, and *add* adds a fact to the WM. This rule, however, cannot be used in a typical rule-based system. Instead, it is usually replaced by two rules:

IF *isa(?x, block)*
THEN *add(isa(?x, pointy_object))*, and
IF *isa(?x, pyramid)*
THEN *add(isa(?x, pointy_object))*.

However, this does not say exactly the same thing as the $\vee$ version does, since there might be situations where we know that either *?x* is a *block* or *?x* is a *pyramid*, but do not know which. In this case, neither of these rules applies, but the original one that uses $\vee$ does.

Now, let us look at the exclusive *or*. For example, the following formula says that "either *NYC* or *albany* is the capital of *NY*, but not both".

capital(NY, NYC) $\oplus$ *capital(NY, albany)*

This can be rephrased as two rules: "*NYC* is the capital of *NY*, if *albany* is not", and "*albany* is the capital of *NY*, if NYC is not"

IF *not(capital(NY, albany))*
THEN *add(capital(NY, NYC))*, and
IF *not(capital(NY, NYC))*
THEN *add(capital(NY, albany))*.

Unfortunately, *not* used in current rule-based systems is different from $\neg$ in logic. In a typical rule-based system, *not(P)* is satisfied if there is no fact in WM matching *P*.

In general, disjunctions are difficult to handle in reasoning. In Section 3, we will show that CSP provides a convenient model to represent these situations.

2.2. Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is characterized as finding values for variables subject to a set of constraints. The standard CSP has three components:

- Variables: A finite set $V = \{v_1, v_2, ..., v_n\}$ of n variables v_i, which are also referred to as constraint variables.
- Values: Each variable v_i is associated with a finite domain D_i, which contains all the possible alternative values for v_i.
- Constraints: A set $C = \{C_1, C_2, ..., C_p\}$ of p constraints or relations on the variables.

The main approach used for solving CSPs is to embed constraint propagation (also known as consistency check) techniques in a backtrack search environment, where backtrack search performs the search for a solution and consistency check techniques prune the search space.

Consistency techniques are characterized by using constraints to remove inconsistent values from the domains of variables. Past research has produced many techniques for such a purpose. The main methods used in practice are arc consistency techniques, e.g., AC-3 (Mackworth 1977), AC-5 (Hentenryck *et al*, 1992), and AC-7 (Bessiere *et al* 1995), and their generalizations and specializations (Hentenryck 1989; Hentenryck *et al* 1992; Liu 1996). For a complete treatment of these methods, please refer to (Mackworth 1977; Mohr & Henderson 1986; Hentenryck 1989; Hentenryck *et al* 1992; Bessiere *et al* 1995; Liu 1995; Liu 1996).

3. Modeling Disjunctions with CSP

This section shows how CSP can be used to model certain types of disjunction in a rule-based system. In this new paradigm with rules and constraints, the underlying techniques for reasoning are forward rule-chaining, constraint propagation and backtrack search.

3.1. The New Paradigm

In the new paradigm, constraints are integrated into rule-based reasoning. It is described by:

1. A working memory (WM): a set of facts representing the current state of the system. There are three types of facts:
   - Simple facts: these are the traditional facts used in the existing rule-based systems.
   - *csp*-disjunctions (inclusive and exclusive): these are special types of disjunctions (defined below) represented by the CSP model.
   - Constraints: these are relations on the *csp*-disjunctions.
2. A rule memory: a set of *IF-THEN* rules.

3. A rule interpreter: this applies the rules to WM by using the traditional forward rule-chaining mechanism, and it is integrated with the constraint solver below.
4. A constraint solver: this uses consistency check and backtrack search for constraint satisfaction. It is integrated with the rule interpreter above.

Thus, the key advance of this new paradigm lies in its use of the CSP model and a constraint solver, resulting in an integration of forward chaining and constraint solving.

3.2. Using Constraint Variables and Domains to Represent Disjunctions

This sub-section describes how constraint variables and their domains can be used to represent disjunctions. We assume the basic definitions of *term* and *atom*, which are *ground* when they contain no variables.

We now define the two kinds of disjunctions that we will handle, the inclusive *csp*-disjunctions and exclusive *csp*-disjunctions. In what follows, we shall, for simplicity with respect to our examples later, restrict the terms in disjunctions to differ only in the last argument.

Definition 1: An exclusive *csp*-disjunction has the following form

$\oplus(P(t_1, ..., t_{n-1}, t_{n1}), P(t_1, ..., t_{n-1}, t_{n2}), ..., P(t_1, ..., t_{n-1}, t_{nm}))$

where $P(t_1, ..., t_{n-1}, t_{ni})$ is a ground atom, $n \geq 1$, t_{ni} is a constant, and $i \neq j$ implies $t_{ni} \neq t_{nj}$. The expression is TRUE *iff* exactly one of the m ground atoms is TRUE.

Note that for all the atoms, the predicate symbols are the same, i.e., P, and so are the first n -1 ground terms. Note also that t_{ni} may appear in any position as long as they are at the same position in each atom. We arbitrarily choose to put them at the end.

This exclusive *csp*-disjunction can be represented by an expression $\oplus P(t_1, ..., t_{n-1}, D)$, where D is a set with the initial value $\{t_{n1}, t_{n2}, ..., t_{nm}\}$. During the reasoning process, some of the atoms (e.g., $P(t_1, ..., t_{n-1}, t_{nk})$) may be proven to be FALSE, then D will be modified to reflect the effect. Thus D changes during the reasoning process, but it is always a subset of $\{t_{n1}, t_{n2}, ..., t_{nm}\}$. When $|D| = 1$, we say D (= $\{t_{ni}\}$) is *decided*, which means that $P(t_1, ..., t_{n-1}, t_{ni})$ is TRUE. When $D = \varnothing$, it means that the exclusive *csp*-disjunction is proven to be FALSE.

An important point is that $\oplus P(t_1, ..., t_{n-1}, D)$ can be represented by a constraint variable, written as $\oplus P(t_1, ..., t_{n-1}, \_)$, whose initial domain is D.

For example,

$\oplus Isa(john, \{soldier, teacher\})$

can represent the fact that *John* is a *soldier* or a *teacher*, and that *John* is only in one of the professions. The corresponding constraint variable $\oplus Isa(john, \_)$ can be used in constraints which hopefully eventually determine *John*'s real profession.

The second type of *csp*-disjunction is defined below.

Definition 2: An inclusive *csp*-disjunction

$\vee(P(t_1,...,t_{n-1}, t_{n1}), P(t_1, ..., t_{n-1}, t_{n2}), ..., P(t_1, ..., t_{n-1}, t_{nm}))$

is like an exclusive *csp*-disjunction, except that this formula is TRUE *iff* $\exists P(t_1, ..., t_{n-1}, t_{ni})$ is TRUE.

This inclusive *csp*-disjunction can be represented by an expression $\vee P(t_1, ..., t_{n-1}, S)$, where the initial value of S is the power set of $\{t_{n1}, t_{n2}, ..., t_{nm}\}$ excluding the empty set. It is convenient to think of S in two parts (R, Q):

- A set of required elements R: the elements that have been proven to be true, i.e., whose associated atoms have been proven to be TRUE.
- A set of possible elements Q: the elements that belong to at least one possible value of S.

Then, R and Q satisfy these conditions: $R \cap Q = \varnothing$ and $R \cup Q \subseteq \{t_{n1}, t_{n2}, ..., t_{nm}\}$. The initial value of S may be $(\{\}, \{t_{n1}, t_{n2}, ..., t_{nm}\})$, and R will grow and Q will shrink in the reasoning process. When $Q = \varnothing$ and $|R| = 0$, we say the inclusive *csp*-disjunction is FALSE. When $Q = \varnothing$ and $|R| \neq 0$, we say S is *decided*, which means the following atoms are all TRUE:

$P(t_1, ..., t_{n-1}, r_1), P(t_1, ..., t_{n-1}, r_2), ...,$ and $P(t_1, ..., t_{n-1}, r_k)$

where $R = \{r_1, r_2, ..., r_k\} \subseteq \{t_{n1}, t_{n2}, ..., t_{nm}\}$. We can see that $\vee P(t_1, ..., t_{n-1}, (R, Q))$ (or $\vee P(t_1, ..., t_{n-1}, S)$) can be represented by a constraint variable $\vee P(t_1, ..., t_{n-1}, \_)$ whose initial domain is the pair (R, Q). Note that we now have constraint variables with a set as a domain, and with a pair of sets as a domain. Call the latter *set* constraint variables.

For example,

$\vee IsFdOf(mike, (\{\}, \{john, james, mary\}))$

can represent the fact that *john* or *james* or *mary* is a friend of *mike* (*or* is inclusive) with $R = \{\}$ and $Q = \{john, james, mary\}$. The corresponding constraint variable $\vee IsFdOf(mike, \_)$ can be used in constraints which hopefully eventually determine who are really *mike*'s friends. If it is decided that *john* is definitely a friend of *mike*, then $R = \{john\}$ and $Q = \{james, mary\}$.

3.3. Using Constraints to Represent Relations among Disjunctions

After introducing the two types of constraint variables to represent the two types of disjunctions, we now in the position to describe some of the constraints that can be used for representing relations among the disjunctions.

Constraint:

$cst\_eq(\oplus P_1(t_{11}, ..., t_{1(n-1)}, \_), \oplus P_2(t_{21}, ..., t_{2(m-1)}, \_))$

where $t_{11}, ..., t_{1(n-1)}, t_{21}, ...,$ and $t_{2(m-1)}$ are ground terms.

Let D_1 and D_2 be the domains of the constraint variables $\oplus P_1(t_{11}, ..., t_{1(n-1)}, \_)$ and $\oplus P_2(t_{21}, ..., t_{2(m-1)}, \_))$ respectively. This constraint ensures that the sets D_1 and D_2 are equal at all time. Its operational semantics is the following (which is an abstraction of the real algorithm implemented):

- $D = D_1 \cap D_2$; **if** $D \neq \varnothing$ **then**
  if $D = \{v\}$ **then**
  add $P_1(t_{11}, ..., t_{1(n-1)}, v)$ to WM;
  add $P_2(t_{21}, ..., t_{2(m-1)}, v)$ to WM
  endif
  $D_1 = D$; $D_2 = D$;
  return(TRUE);
  else return(FALSE)

For example, we have
$\oplus$*Isa*(*john*, {*soldier*, *teacher*, *professor*, *doctor*}), and
$\oplus$*Isa*(*james*, {*teacher*, *doctor*, *student*}).

If we know that *john* and *james* have the same profession, we can express this with the constraint

cst_eq($\oplus$*Isa*(*john*, _), $\oplus$*Isa*(*james*, _)).

The system will automatically propagate the constraint by using the built-in consistency algorithms to reduce both sets so that the following are obtained:

$\oplus$*Isa*(*john*, {*teacher*, *doctor*}), and
$\oplus$*Isa*(*james*, {*teacher*, *doctor*})

If due to some other constraint (or information) it is decided that *john* is a *teacher*, then the following two elements will be added to WM:

Isa(*john*, *teacher*), and *Isa*(*james*, *teacher*)

If we have the following rule in the rule memory:

IF *Isa*(?*x*, *teacher*)
THEN *add*(*has*(?*x*, *many_students*))

This rule will be fired to obtain two more facts:

has(*john*, *many_students*), and
has(*james*, *many_students*)

This example shows that constraint propagation and rule chaining are integrated.

Constraint:

cst_not_eq($\oplus P_1(t_{11}, ..., t_{1(n-1)}, \_)$, $\oplus P_2(t_{21}, ..., t_{2(m-1)}, \_)$)

where $t_{11}, ..., t_{1(n-1)}, t_{21}, ...$, and $t_{2(m-1)}$ are ground terms.

Let D_1 and D_2 be the domains of $\oplus P_1(t_{11}, ..., t_{1(n-1)}, \_)$ and $\oplus P_2(t_{21}, ..., t_{2(m-1)}, \_)$) respectively. Then the constraint's operational semantics is given by:

- **if** $|D_1| = 1$ and $|D_2| > 1$ **then**
  $D_2 = D_2 - D_1$;
  if $D_2 = \{v\}$ **then** add $P_2(t_{21},..., t_{2(m-1)}, v)$ to WM
  endif
  return(TRUE);
  elseif $|D_2| = 1$ and $|D_1| > 1$ **then**
  this case is similar to the above one;
  elseif $|D_1| = 1$ and $|D_2| = 1$ **then**
  if $D_1 \neq D_2$ **then** return(TRUE) **else** return(FALSE)
  endif
  else return(TRUE)

For example, we have
$\oplus$*Isa*(*john*, {*soldier*, *teacher*, *professor*, *doctor*}),
and $\oplus$*Isa*(*james*, {*teacher*, *doctor*, *student*}).

The following constraint says that *john* and *james* have different professions:

cst_not_eq($\oplus$*Isa*(*john*, _), $\oplus$*Isa*(*james*, _))

Constraint:

cst_not_in(v, $\oplus P(t_1, ..., t_{n-1}, \_)$)

where t_1, ..., and t_{n-1} are ground terms, and v is a constant.

Let D be the domain of $\oplus P(t_1, ..., t_{n-1}, \_)$. This constraint constrains that v is not a possible element in D, which also means that $P(t_1, ..., t_{(n-1)}, v)$ is FALSE. We have:

- $D = D - \{v\}$; **if** $D = \varnothing$ **then** return(FALSE)
  else if $D = \{u\}$ (or $|D| = 1$) **then**
  add $P(t_1,..., t_{n-1}, u)$ to WM;
  endif
  return(TRUE)
  endif

Constraint:

cst_set_eq($\vee P_1(t_{11}, ..., t_{1(n-1)}, \_)$, $\vee P_2(t_{21}, ..., t_{2(m-1)}, \_)$)

where $t_{11}, ..., t_{1(n-1)}, t_{21}, ...$, and $t_{2(m-1)}$ are ground terms.

Let (R_1, Q_1) and (R_2, Q_2) be the domains of $\vee P_1(t_{11}, ..., t_{1(n-1)}, \_)$ and $\vee P_2(t_{21}, ..., t_{2(m-1)}, \_)$ respectively. Then this constraint is handled by:

- $R = R_1 \cup R_2$; $Q = \{r \mid r \in Q_1 \cap Q_2, r \notin R\}$;
  if $R \subseteq R_1 \cup Q_1$ and $R \subseteq R_2 \cup Q_2$ and ($R \neq \varnothing$ or $Q \neq \varnothing$)
  then $R_1 = R$; $R_2 = R$; $Q_1 = Q$; $Q_2 = Q$;
  for each $r \in R$ and $r \notin R_1$ **do**
  add $P_1(t_{11}, ..., t_{1(n-1)}, r)$ to WM;
  for each $r \in R$ and $r \notin R_2$ **do**
  add $P_2(t_{21}, ..., t_{2(m-1)}, r)$ to WM;
  return(TRUE);
  else return(FALSE)

For example, we have
$\vee$*IsFdOf*(*mike*, ({*john*}, {*james*, *steve*, *david*})), and
$\vee$*IsFdOf*(*andrew*, ({*steve*}, {*john*, *kate*, *david*}))

If we set the constraint

cst_set_eq($\vee$*IsFdOf*(*mike*, _), $\vee$*IsFdOf*(*andrew*, _)),

which says that *mike* and *andrew* have the same set of friends, we will obtain:

$\vee$*IsFdOf*(*mike*, ({*john*, *steve*}, {*david*})), and
$\vee$*IsFdOf*(*andrew*, ({*john*, *steve*}, {*david*})).

Two more facts will be added in WM, i.e.,

IsFdOf(*mike*, *steve*), and *IsFdOf*(*andrew*, *john*).

Constraint:

cst_set_not_in(v, $\vee P(t_1, ..., t_{n-1}, \_)$)

where t_1, ..., and t_{n-1} are all ground terms.

Let (R, Q) be the domain of $\vee P(t_1, ..., t_{n-1}, \_)$, this constraint constrains that v is not a possible element in Q, which means that $P(t_1, ..., t_{n-1}, v)$ cannot be TRUE. Its operational semantics is obvious, and omitted.

3.4. Introducing Choice Making and Backtracking

The consistency techniques used above for constraint solving are all based on arc consistency (Hentenryck *et al* 1992; Liu 1995). Arc consistency alone may not be sufficient to solve a CSP because arc consistency does not guarantee global consistency (Mackworth 1977). Then, a combination of backtrack search and consistency check is required. This approach can be described as an iterative procedure of two steps: consistency check and choice making. If a choice is proved to be wrong (when the consistency check returns FALSE), backtracking will be initiated. In the process, the previous state is restored, and an alternative is selected (Hentenryck 1989).

Let us define some choice making functions. Each of them sets up a choice point for later backtracking. The choice functions are also constraints because each value selection will trigger consistency check.

Choice function: *cst_select*($\oplus P(t_1, ..., t_{n-1}, \_)$, *func*)

where t_1, ..., and t_{n-1} are all ground terms, and *func* is a user defined procedure.

Let D be the domain of $\oplus P(t_1, ..., t_{n-1}, \_)$, this function selects a value v from D using the procedure *func*. *func* allows the user to control the selection process in order to find the solution quickly. This choice function behaves as follows:

- **if** there is no more value to be selected in D **then**
  return(FALSE)
  else v is selected from D using *func*;
  $D = \{v\}$;
  add $P(t_1, ..., t_{n-1}, v)$ in WM;
  return(TRUE)
  endif

For example, we have:

$\oplus$*Capital*(*NY*, {*NYC*, *albany*}),

which says that the capital of *New York* (*NY*) is either *NYC* or *albany*, but not both. We can apply the selection by using

cst_select($\oplus$*Capital*(*NY*, _), *func*).

Suppose that *func* chooses the first possible value first, i.e., *NYC*. After it is selected, *Capital*(*NY*, *NYC*) will be automatically added in WM, and then constraint propagation will be carried out, etc. When backtracking occurs, the second value will be tried and so on.

Choice function: *cst_set_select*($\vee P(t_1, ..., t_{n-1}, \_)$, *func*)

where t_1, ..., and t_{n-1} are all ground terms, and *func* is a user defined procedure.

Let (R, Q) be the domain of $\vee P(t_1, ..., t_{n-1}, \_)$. This function selects a value V (a set) from Q ($V \subseteq Q$) using the procedure *func*. It behaves as follows:

- **if** there is no more value to be selected from Q **then**
  return(FALSE)
  else A set V is selected from Q using *func*;
  $Q = \varnothing$; $R = R \cup V$;
  for each $r \in V$ **do** add $P(t_1,..., t_{n-1}, r)$ to WM;
  return(TRUE)
  endif

For instance, we have

$\vee$*IsFdOf*(*mike*, ({*john*}, {*james*, *mary*, *steve*}))

and we know that *mike* has only two friends. We can try the following:

cst_set_select($\vee$*IsFdOf*(*mike*, _), *func*)

Suppose that *func* chooses the first possible value first, i.e., *james*, which effectively rules out the other values. Then, *mike*'s friends are *john* and *james*. We obtain

$\vee$*IsFdOf*(*mike*, ({*john*, *james*}, {})).

After that, other necessary operations are performed, e.g., adding *IsFdOf*(*mike*, *james*) to WM and constraint propagation, etc. When a selection is proved to be wrong, backtracking will be performed. The second element, the third element, etc., will be tried and so on.

3.5. Some Test Functions on Constraint Variables

Here, we present some test functions on constraint variables. They are used to exploit the partial information provided by disjunctions for various purposes.

Test function: *test_in*(T, $\oplus P(t_1, ..., t_{n-1}, \_)$)

where t_1, ..., and t_{n-1} are all ground terms, and T is a set of constants.

Let D be the domain of $\oplus P(t_1, ..., t_{n-1}, \_)$. This test function behaves as follows:

- **if** $D \subseteq T$ **then** return(TRUE) **else** return(FALSE)

For example, we have $\oplus$*Capital*(*NY*, {*NYC*, *albany*}), which says that the capital of *New York* (*NY*) is either *NYC* or *albany*, but not both, and the following rule:

IF *include*(?*tour*, *capitalOf*(*NY*)) and
test_in({*NYC*, *albany*}, $\oplus$*Capital*(*NY*, _))
THEN *add*(*join*(*I*, ?*tour*)))

This rule allows the system to act on the partial information, i.e., *test_in* does not have to find the fact *Capital*(*NY*, *NYC*) or *Capital*(*NY*, *albany*) in WM before firing. Instead, it only needs to check whether any one of these two cities or both are the only possible values for the capital of *NY*. It does not matter which.

If WM has the following two facts:

include(*tour*16, *capitalOf*(*NY*)), and
$\oplus$*Capital*(*NY*, {*NYC*, *albany*})

the rule will fire to add *join*(*I*, *tour*16)) to WM.

Test function: *test_set_in*(T, $\vee P(t_1, ..., t_{n-1}, \_)$)

where t_1, ..., and t_{n-1} are all ground terms, and T is a set of constants.

Let (R, Q) be the domain of $\vee P(t_1, ..., t_{n-1}, \_)$, This test function behaves as follows:

- **if** $(T \cap R) \neq \varnothing$ or ($R = \varnothing$ and $Q \subseteq T$) **then**
  return(TRUE)
  else return(FALSE)

For example, we wish to express that "if something is a block or a pyramid, then it is a pointy_object" (*or* is inclusive). We can write:

IF *test_set_in*({*block*, *pyramid*}, *isa*(?*x*, _))
THEN *add*(*isa*(?*x*, *pointy_object*))

3.6. Complications With the Integration of Choice Making and Rule Chaining

Combining backtrack search and forward chaining creates some complications. The problem lies in the handling of inconsistency. For our discussion, we classify two types of inconsistency. The first type is the normal inconsistency in logic (IL), e.g., both A and $\neg A$ are deduced, and the other is the inconsistency of constraints (IC). IC is easy to detect and to handle because when the domain of a constraint variable is empty, it is known that there is a inconsistency, and backtracking can be used to deal with it. However, IL is hard to detect as most rule-based systems are informal systems that have no mechanism for this purpose. This has some implications for our proposed integration.

- If a rule-based system is unable to detect IL, then (1) constraints cannot be conditions in a rule, (2) choice making and backtracking should not be allowed.

The reason is that both (1) and (2) could introduce IL. Due to space limitation, we are unable to discuss this further. Interested readers, refer to (Liu & Jaffar 1996).

In general, if a rule-based system is unable to detect IL, (1) and (2) should not be allowed. Then, constraints can only appear as consequents of rules, and there will be no backtrack search but only consistency check.

However, if an inconsistency checker is implemented for detecting IL, then both (1) and (2) can be allowed, and both IC and IL will trigger backtracking.

Apart from the above two situations, a third one is also reasonable. We assume that only ICs may occur in an application, then we can also allow both (1) and (2) because IC is easily detected. Our prototype system makes this assumption. This assumption is realistic because that is the case in most existing rule-based systems. They do not have mechanisms for detecting IL. It is the user's responsibility not to introduce any or to check it.

4. An Implementation

We have implemented a prototype system (called CFR) in Common Lisp. Below are some implementation issues.

- Apart from WM and rule memory in a rule-based system, a constraint variable memory is introduced to store constraint variables.
- For consistency check of constraints involving normal constraint variables, we used those algorithms in (Hentenryck *et al* 1992; Liu 1995) as they are the most efficient algorithms. For set constraints, we designed our own algorithms as there is little reported work on this type of constraints. Consistency check of *cst_eq*, *cst_not_eq*, and *cst_set_eq* can all be done in linear time to the size of the domain D or $|R \cup Q|$. *cst_not_in* and *cst_set_not_in* can be done in constant time.
- A choice stack is used to keep track of the choices that have been made and to remember the information necessary for restoring state upon backtracking. This is similar to CLP languages such as CHIP (Hentenryck 1989). The difference is that each choice here has to remember the facts that have been added to WM after a choice is made. When backtracking comes to the choice, these facts must be removed.
- Finally, the pattern matching algorithm for rule-chaining needs to be modified to accommodate the constraint satisfaction facility. Due to the space limitation, we are unable to discuss this and many other issues.

Below, we briefly describe the syntax of rules, constraint variables, and constraints in CFR.

***IF-THEN* rules**: A rule is defined using the construct:

(define-rule <name> <conditions> -> <actions>)

For example, the rule:

```
(define-rule is_food
   (edible ?x)
     -> (add `(is_food ,x)))
```

says that if there is a fact in WM that matches *(edible ?x)*, this rule will fire and add the evaluation result `` `(is_food ,x) `` to WM. `` `(is_food ,x) `` is in Lisp syntax ("`", "'", and "," are used according to their meanings in Lisp), and x here will be substituted to whatever value $?x$ has after matching with the fact in WM.

Constraint variable declarations:

1). $\oplus P(t_1, \ldots, t_{n-1}, D)$ => (corresponding to)
(cst_in '(P t_1 ... t_{n-1} D))
e.g., $\oplus capital(NY, \{NYC, albany\})$ =>
(cst_in '(capital NY (NYC albany)))

2). $\vee P(t_1, \ldots, t_{n-1}, (R, Q))$ =>
(cst_set_in '(P t_1 ... t_{n-1} (R Q)))
e.g., $\vee IsFdOf(joe, (\{steve\}, \{john, kate\}))$ =>
(cst_set_in '(IsFdOf joe ((steve) (john kate))))

Constraints:

1). $cst\_eq(\oplus P_1(t_{11}, \ldots, t_{1(n-1)}, \_), \oplus P_2(t_{21}, \ldots, t_{2(m-1)}, \_))$
=> (cst_eq '(P_1 t_{11} ... $t_{1(n-1)}$ _) '(P_2 t_{21} ... $t_{2(m-1)}$ _))
e.g., $cst\_eq(\oplus Isa(john, \_), \oplus Isa(james, \_))$ =>
(cst_eq '(Isa john _) '(Isa james _))

2). $cst\_not\_eq(\oplus P_1(t_{11},\ldots,t_{1(n-1)},\_),\oplus P_2(t_{21}, \ldots, t_{2(m-1)}, \_))$
=> (cst_not_eq '(P_1 t_{11} ... $t_{1(n-1)}$ _) '(P_2 t_{21} ... $t_{2(m-1)}$ _))
e.g., $cst\_not\_eq(\oplus Isa(john, \_), \oplus Isa(james, \_))$ =>
(cst_not_eq '(Isa john _) '(Isa james _))

3). $cst\_set\_eq(\vee P_1(t_{11},\ldots,t_{1(n-1)}, \_), \vee P_2(t_{21}, \ldots, t_{2(m-1)}, \_))$
=> (cst_set_eq '(P_1 t_{11} ... $t_{1(n-1)}$ _) '(P_2 t_{21} ... $t_{2(m-1)}$ _))
e.g., $cst\_set\_eq(\vee IsFdOf(mike, \_), \vee IsFdOf(joe, \_))$
=> (cst_set_eq '(IsFdOf mike _) '(IsFdOf joe _))

Due to lack of space, we will not describe the corresponding constructs in CFR for the other constraints and choice and test functions. They are quite similar to the ones above.

5. An Example

We now present a simple example to illustrate how rules and constraints interact with each other in the reasoning process. The rule definitions here are self-explanatory.

```
(define-rule professor
  (isa ?x science_professor)
  -> (add `(works_in_a ,x university))
     (cst_in `(teaches ,x (computer math physics
     chemistry biology))))
(define-rule computer
  (isa ?x science_professor)
  (has_no ?x computer)
  -> (cst_not_in 'computer `(,x teaches _)))
(define-rule math
  (is_good_in ?x math)
  (isa ?x science_professor)
  -> (cst_in `(teaches ,x (computer math physics))))
(define-rule csp-test
  (test_in (physics math) (teaches ?x _))
  -> (add `(gives_lecture_in ,x science_building)))
(define-rule lab
  (does_not_do ?x lab_work)
  -> (cst_not_in 'chemistry `(teaches ,x _))
     (cst_not_in 'biology `(teaches ,x _)))
(define-rule degree
  (teaches ?x ?y)
  -> (add `(likes ,x ,y))
     (cst_set_in `(has ,x `(((PhD in ,y)) `((MSc in ,y)))))
```

Let us run the system with the following facts:

```
(add '(isa fred science_professor))
(add '(has_no fred computer))
(add '(isa john science_professor))
(add '(does_not_do john lab_work))
(cst_eq '(teaches john _) '(teaches fred _))
```

After all the rule chaining and constraint propagation, the working memory becomes:

```
1:  (isa fred science_professor)
2:  (works_in_a fred university)
3:  (cst_fact (teaches fred _) (math physics))
4:  (has_no fred computer)
5:  (isa john science_professor)
6:  (works_in_a john university)
7:  (cst_fact (teaches john _) (math physics))
8:  (does_not_do john lab_work)
9:  (gives_lecture_in john science_building)
10: (gives_lecture_in fred science_building)
```

Fact 3 and 7 are special facts representing two constraint variables and their remaining domains. From them, we know that both *fred* and *john* teach either *math* or *physics*, but we still do not know which.

Let us say that we are not satisfied with the result. We would like to make a guess about what they teach. We can use the following selection function:

```
(cst_select '(teaches fred _) #'car)
```

This selects *math* as the subject that *fred* teaches. After constraint propagation and rule chaining, we obtain the fact that *john* also teaches *math*. The following facts are deduced:

```
11: (teaches fred math)
12: (teaches john math)
13: (likes fred math)
14: (likes john math)
15: (has fred (PhD in math))
16: (has john (PhD in math))
17: (cst_set_fact (has fred _) ((PhD in math))
                                ((MSc in math)))
18: (cst_set_fact (has john _) ((PhD in math))
                                ((MSc in math)))
```

The last two facts (17 and 18) say that *fred* and *john* have a *PhD* in *math* and may or may not have a *MSc* in *math*.

If later we have some more information saying that *fred* does not have a *PhD* degree in *math*, this can be expressed like this:

```
(cst_set_not_in '(PhD in math) '(has fred _))
```

It immediately causes a conflict with fact 17 because fact 17 says that *fred* has a *PhD* in *math*. Then, backtracking is performed. The facts from 11 to 18 are removed to restore the previous state. *physics* is selected this time as the subject that *fred* teaches, which in turn causes a number of facts to be produced:

```
11: (teaches fred physics)
12: (teaches john physics)
13: (likes fred physics)
14: (likes john physics)
15: (has fred (PhD in physics))
16: (has john (PhD in physics))
17: (cst_set_fact (has fred _) ((PhD in physics))
                                ((MSc in physics)))
18: (cst_set_fact (has john _) ((PhD in physics))
                                ((MSc in physics)))
```

Since *math* is eliminated as the possible course that *fred* and *john* teach. Fact 3 and 7 in WM become:

```
3: (cst_fact (teaches fred _) (physics))
7: (cst_fact (teaches john _) (physics))
```

The kind of reasoning illustrated here cannot be carried out in an existing rule-based system.

6. Related Work

The most closely related work to our research is constraint logic programming (CLP) (Jaffar & Maher 1994) where a considerable amount of research has been done to integrate constraint satisfaction with logic programming. A number of systems have been built, and many successful applications have also been reported (Jaffar & Maher 1994). Two representative CLP languages are CLP(*R*) (Jaffar & Lassez 1987) and CHIP (Hentenryck 1989). These languages are based on Horn clauses and backward chaining. Our work is different from CLP in a number of ways. The main differences are as follows.

1. Our proposed technique is based on forward chaining rather than backward chaining as in CLP languages. Forward chaining and backward chaining reason from different directions and are suitable for solving different types of problems. Forward chaining are mainly used for building expert systems for solving real-life knowledge intensive tasks. Since the CLP languages based on backward chaining have been very successful in practice for solving practical combinatorial search problems, it is only natural that forward chaining should also be integrated with constraint solving to provide a more powerful reasoning technique for solving practical reasoning problems.
2. In CLP languages, backtracking and choice making are provided by the host language Prolog. While in forward chaining, backtracking and choice making facilities have to be added, which creates some complications as discussed in Section 3.6.

To the best our knowledge, limited work has been done on combining constraint solving with forward chaining rule-based system. BABYLON (Christaller *et al* 1992) is one of the hybrid environments for developing expert systems that has attempted to include constraint solving in its rule-based system. BABYLON provides representation formalisms of objects, rules, Prolog and constraints. CONSAT is the constraint system of BABYLON, which is separated from others and cannot access rules. Although in the condition part of the rules, it is possible to verify whether a constraint is satisfied, the action part of a rule cannot access constraints. This is quite different from our system, within which constraint solving and rule-chaining are integrated. Rules can post and test constraints, and constraint satisfaction can also trigger chaining of rules.

7. Conclusion

This paper shows how CSP can be used to model two types of important disjunctions in rule-based reasoning. These disjunctions have not been handled satisfactorily in the current rule-based systems. In the proposed scheme, the simple representation and efficient algorithms in CSP are used to deal with these types of disjunction. This results in the integration of two important types of reasoning techniques, i.e., constraint solving and (forward) rule-chaining. Hence. the power of rule-based systems is increased.

The current integration of CSP with rule-based reasoning is still restricted, i.e., mainly for modeling the two types of disjunction. Our next step is to deal with general constraints in a forward chaining framework.

Acknowledgments: Bing Liu thanks Peter Lucas from Utrecht University for his advice on some expert system issues. We thank Roland Yap for his help. Finally, we are grateful to the AAAI reviewers for their insightful comments.

References

Bessiere, C., Freuder, E. C. and Regin, J-C. 1995. "Using inference to reduce arc consistency computation," *IJCAI-95*, 592-598.

Charniak, E., Riesbeck, C., McDermott, D. and Meehan, J. 1987. *Artificial Intelligence Programming*, Lawrence Erlbaum Associates Inc.

Christaller, T., di Primio, F., Schnepf, U. and Voss, A. 1992. *The AI Workbench BABYLON*. Academic Press.

Hentenryck, P.V. 1989. *Constraint Satisfaction in Logic Programming*, MIT Press.

Hentenryck, P.V., Deville, Y. and Teng, C-M. 1992. "A generic arc consistency algorithm and its specializations," *Artificial Intelligence* 27, 291-322.

Ilog Solver. 1992. *Reference Manual*, ILOG, France.

Jaffar, J. and Lassez, J. 1987. "Constraint logic programming," *Proceedings of the Fourteenth Annual ACM Symposium on Principle of Programming Language*.

Jaffar, J. and Maher, M. 1994. "Constraint logic programming: a survey." *J. Logic Programming* 19, 503-581.

Liu, B. 1995. "Increasing functional constraints need to be checked only once," *IJCAI-95*, 586-591.

Liu, B. and Jaffar, J. 1996. *Using Constraints to Model Disjunction in Rule-Based Reasoning*. DISCS Technical Report.

Liu, B. 1996. "An improved generic arc consistency algorithm and its specializations." To Appear in *Proceedings of Fourth Pacific Rim International Conference On Artificial Intelligence (PRICAI-96)*.

Lucas, P. and Van Der Gaag, L. 1991. *Principles of Expert Systems*, Addison-Wesley.

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

Mackworth, A.K. 1992. "The logic of constraint satisfaction," *Artificial Intelligence* 58, 3-20.

Mohr, R. and Henderson, T. 1986. "Arc and path consistency revisited," *Artificial Intelligence* 28, 225-233.

Newell, A. 1973. "Production systems: models for control structure," In *Visual Information Processing*, W.G. Chase (Eds), Academic Press, 1973.