# Smart clients: Constraint satisfaction as a paradigm for scaleable intelligent information systems

**Marc Torrens i Arnal and Boi Faltings**

Artificial Intelligence Laboratory (LIA)
Swiss Federal Institute of Technology (EPFL)
IN-Ecublens, CH-1015, Lausanne
Switzerland
torrens@lia.di.epfl.ch, faltings@lia.di.epfl.ch

**Abstract**

Many information systems are used in a problem solving context. Examples are travel planning systems, catalogs in electronic commerce, or agenda planning systems. They can be made more useful by integrating problem-solving capabilities into the information systems. This poses the challenge of *scaleability*: when hundreds of users access a server at the same time, it is important to avoid excessive computational load.

We present the concept of *smart clients*: lightweight problem-solving agents based on constraint satisfaction which can carry out the computation- and communication-intensive tasks on the user's computer. We present an example of an air travel planning system based on this technology.

## Intelligent Information Systems

The world today is full of information systems which make huge quantities of information available. A good example is the travel domain, where information systems accessible through the Internet provide information about schedules, fares and availability of almost any means of transport throughout the world.

The first generation of information systems provided simple database access facilities such as SQL which allow a user to access specific information. The current generation provides some intelligence for locating the right information, for example by searching for flights at a certain fare or with certain schedule constraints.

However, all information systems are ultimately used not to just provide information, but to *solve problems.* Thus, we believe that the next generation of intelligent information systems should provide explicit support for the problem-solving activities that a user carries out with them. For example, a travel information system should help the user plan an entire trip according to constraints and preferences, and not just give information about certain airline schedules.

One dimension of this new generation will be the integration of various information systems into a uniform framework using agents. Such integration is apparent for example in shopping robots such as Jango (Jango 1998), or in the integrated travel information system designed by Siemens as a demonstration within the FIPA (FIPA 1998) consortium.

Another dimension will be to provide explicit problem-solving capabilities: help with configuring a complete solution, possibly consisting of many parts. For example, a travel planning system would *configure* an entire trip with matching outgoing and return flights, ground connections, etc. An insurance planner could configure a suitable insurance package from offers of different companies with different parameters. Such problem solvers will be essential to help people deal with the complexity of the information provided by the servers.

## Smart clients

Another issue which has to be faced in information systems is *scaleability*: the ability to support large numbers of simultaneous users. Client-server computing allows such scaleability by distributing the computational load to the client computers. The concept of *thin clients* has extended client-server computing to much larger systems, in particular the Internet.

Traditional wisdom would say that in order to make a client intelligent, it will have to include a lot of complex code and data, i.e. be a very fat client. The point of this paper is to show that constraint satisfaction allows us to have smart (intelligent) clients that are also smart (thin), by marrying two characteristics:

- constraint satisfaction provides search algorithms which are both very simple and compact to implement, and at the same time very efficient.
- constraints allow representing complex information in a compact form.

As a result, smart clients are efficient autonomous problem-solvers which at the same time are small enough to be sent through a network in a short time.

Such problem-solvers are particularly useful for catalog-type systems, where user has to select from a range of possibilities: this set can be represented as a CSP (Constraint Satisfaction Problem) and solved by incremental constraint posting.

However one can imagine similar smart clients for designing and planning applications.

## Architecture

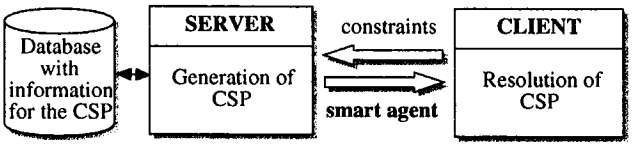


Figure 1: A smart-agent architecture.

The architecture supporting smart-agent methodology is shown in Fig. 1. The client sends a request containing the user constraints to the server. The server will access databases in order to generate the corresponding CSP taking into consideration the constraints of the user. The CSP is packaged with a search algorithm to form an agent which is transfered to the client side. In this way the user can browse through the different solutions by interacting with the agent locally.

We decompose the process into two parts:

- the information server compiles all relevant information from the database and the user constraints (query) into the corresponding CSP. The CSP is a compact representation of all solutions that the problem can have given the initial restrictions of the user.
- the server sends a smart agent consisting of the CSP and search algorithms to the client. This allows the user to browse through all the possible solutions. Since the agent executes on the client, response time can be very fast and the user can compare different alternatives without placing unnecessary load on the server.

Building the CSP requires only a small fraction of time compared to solving the CSP, so having the agent executed on the client significantly reduces server overload. After having generated the CSP there is no longer need of accessing the server, so the agent sent by the server is completely autonomous.

Note that this architecture is protected by a pending patent.

## Java Constraint Library (JCL)

We implemented the Java Constraint Library (JCL), which allows us to package constraint satisfaction problems and their solvers in compact autonomous agents suitable for transmission on the Internet. We will first give a brief introduction to constraint satisfaction techniques and then describe JCL.

### Constraint satisfaction problems

Constraint Satisfaction Problems (CSPs) are ubiquitous in applications like configuration (Sanjay Mittal 1989; Sabin and Freuder 1996), planning (Stefik 1981), resource allocation (Choueiry 1994; Sathi and Fox 1989), scheduling (Fox 1987) and many others. A CSP is specified by a set of variables and constraints among them. A solution to a CSP is a set of value assignments to all variables such that all constraints are satisfied. There can be either many, 1 or no solutions to a given problem. The main advantages of constraint based programming are the following:

- It offers a general framework for stating many real world problems can be stated in a succinct and elegant way.
- A constraint based representation can be used to synthesize solutions of the problem as well as for verification purposes (i.e. showing that a solution satisfies all constraints).
- The nature of the representation allows a formal description of the problems as well as a declarative description of search heuristics.

A finite, discrete Constraint Satisfaction Problem (CSP) is defined by a tuple $P = (X, D, C, R)$ where $X = \{X_1, \ldots, X_n\}$ is a finite set of variables, each associated with a domain of discrete values $D = \{D_1, \ldots, D_n\}$, and a set of constraints $C = \{C_1, \ldots, C_l\}$. Each constraint $C_i$ is expressed by a relation $R_i$ on some subset of variables. This subset of variables is called the *connection* of the constraint and denoted by $con(C_i)$. The relation $R_i$ over the connection of a constraint $C_i$ is defined by $R_i \subseteq D_{i1} \times \ldots \times D_{ik}$ and denotes the tuples that satisfy $C_i$. The *arity* of a constraint $C$ is the size of its connection.

A large body of techniques exists for efficiently solving CSPs. For more details see (Tsang 1993).

### The Java Constraint Library (JCL)

We have implemented a library of common constraint satisfaction techniques in the Java Constraint Library (JCL). It provides services for:

- creating and managing discrete CSPs
- applying preprocessing and search algorithms to CSPs

JCL can be used either in an applet[1] or in a standalone Java application. The purpose of JCL is to provide a framework for easily building agents that solve CSPs on the Web. JCL is divided into two parts: A basic constraint library available on the Web and a constraint shell built on the top of this library, allowing CSPs to be edited and solved. JCL allows the development of portable applications and applets using the constraint mechanisms. It can be downloaded from `http://liawww.epfl.ch/~torrens`.

The library contains search and preprocessing algorithms. The search algorithms allow us to find the solutions of a CSP, while the preprocessing algorithms are used to simplify a CSP by eliminating values and

[1] An *applet* is an application designed to be transmited over the Internet and executed by a Java-compatible Web browser.

compound labels that do not affect its solutions. Several search algorithms are implemented in JCL. There are three main algorithms derived from *Chronological Backtracking* (BT) that are: *Backmarking* (BM), *Backjumping* (BJ) and *Forward Checking* (FC) (Kondrak and van Beek 1997). Some combinations of them are implemented in CSPLib (van Beek 1995) and adapted in JCL. Fig. 2 shows a hierarchy of the algorithms in JCL.

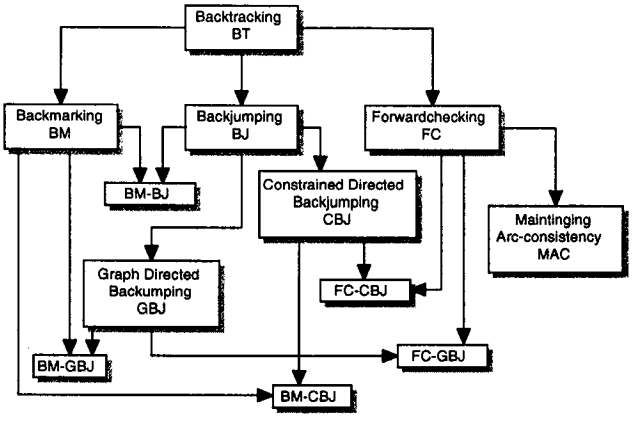


Figure 2: A hierarchy of the search algorithms implemented in JCL.

The following two preprocessing algorithms are implemented in JCL: *Arc-consistency* (AC) and *Path-consistency* (PC) (Mackworth 1977).

## Representing solution spaces

Combinatorial problems can have an enormous number of solutions, arising through the combinations of variable values. For a problem with $n$ variables of uniform domain size $d$, there can be up to $d^n$ different solutions. A complete list would require $nd^n$ units of storage.

If variables are completely independent, the space can be represented as a cross product of all their values. This would require only $n \cdot d$ units of storage.

However, in most cases the admissible combinations are restricted by constraints. In the worst case, a constraint can be stored as a list of all the admissible tuples. In such a case, a $k$-ary constraint can require up to $d^k$ units of storage. In a network with $n$ variables, there can be at most $n \cdot (n-1) \cdot (n-k+1) \leq n^k$ such constraints, so in the worst case we require at most

$$(n \cdot d)^k$$

units of memory to store a network of degree $k$. If $k$ is relatively small with respect to $n$, this is *exponentially better* than storing all admissible combinations. The price to pay, of course, is that solutions can only be accessed by solving the NP-complete problem of constraint satisfaction. We call a space of solutions described in this way a constraint space.

## Example: Air Travel Planning

We are all faced with the problem of arranging trips. Typically, we have to meet with a set of people in different cities. Each of the people has certain days on which they are available for a meeting. Transportation schedules impose additional constraints.

In the current state of affairs, schedule information can only be obtained by queries to travel agents or Web servers for particular routes, dates and times. Thus finding the optimal plan would require separate queries for every part of every alternative itinerary. Since each query implies response times on the order of 1 minute, this makes travel planning very tedious. The prototypical business Air Travel Planning (ATP) system is a kind of personal assistant designed to facilitate arranging these kinds of trips using the concept of smart agents.

Consider the following example of a travel planning problem:

> I live in Bern, Switzerland, and would like to visit colleagues in Princeton, New Jersey, and London. I would like to spend at least two days in each place, and will need to travel in the first two weeks of February.

Of course, I also have some preferences about airlines, departure times, transfer airports and so on, but these are too complicated to state in a first query.

Since I live in Bern, I can leave from any of three Swiss airports (Zurich, Basel, Geneva, abbreviated as ZRH, BSL, GVA). Also, for Princeton I can fly to two New York airports (JFK, EWR) or to Philadelphia (PHL), and there are three airports in London to consider (LGW, LHR, LCY). Finding the best plan for my trip involves checking all combinations of flights between these airports. Considering only direct flights (except flights from London to Philadelphia, where we consider one-stop flights because there are no direct flights), there are in fact more than 4 million solutions for this problem. An intelligent tool would be of great help to manipulate this large set.

There are two important questions about efficiency in this kind of information systems:

1. How many server accesses are required ?
2. How much information has to be sent from the server to the client ?

Let us analize these two questions in the traditional flight information systems and in our smart-agent based system. In table 1 we show the approximated answers to the example describe above.

**Conventional approach** Let us consider planning such a trip using the systems currently available on the WWW. One type of system, most commonly offered by airlines themselves, allows simply to inspect flights or connections for one particular leg at a time. On a multi-leg trip such as this one, this would require the customer to carefully note down all solutions for the different legs

| | Conventional | Smart agents |
|---|---|---|
| Server access | 447039.6 | 1 |
| Size of 1 transfer | $\approx 60\,Kb$ | $\approx 180\,Kb$ |
| Size of total transfers | $\approx 26.82\,Gb$ | $\approx 180\,Kb$ |

Table 1: Conventional systems vs. Smart Agents

and finally put together a solution by hand - not a very satisfactory way of planning such a trip.

Fortunately, tools such as Travelocity (Travelocity 1998) allow us to configure multi-leg trips. Complete itineraries are constructed on the server and returned to the customer for selection. In an example such as the one given above, we could in principle browse through all the 4 million possible solutions, evaluating each manually as to whether it satisfies our constraints. Considering that solutions are displayed in web pages with about 10 solutions at a time, this would involve an enormous number of transfers. Each web page sent back has about 60 Kbytes, so in total we need to transfer (and look at) about 24 Gbytes ($4 * 10^6/10 * 60$ Kbytes) of information to see the complete solution space.

However, a smart user can save some time by exploiting regularities of the domain, such as the fact that most flights operate daily and usually have space available. But this means precisely that the tool is not very intelligent: it still requires the customer to do most of the work.

**Smart clients** Smart clients offer the possibility to support the customer's decision-making process with an intelligent scratchpad. In contract to conventional tools, it can keep track of all options and choices and avoid having to reload information which had already been requested earlier.

In this example, the smart client collects information about *all* flights which could be part of a solution in one single server access. Since the information is encoded as a CSP, it only needs to record the *sum* of the information for each possible flight, not all their combinations. In this example, there are 795 possible flights, each of which is encoded in a text line containing no more than 80 bytes. Thus, the entire CSP takes up no more than 63 Kbytes ($795 * 80bytes$). Added to this the size of the JCL (100 Kbytes) and the graphical interface (20 Kbytes), the complete smart client is no longer than 183 Kbytes. Considering that the size of an average response page from a conventional server such as Travelocity is already 60 Kbytes, this is not a particularly large agent. It can be transmitted through the internet in less than 1 minute.

Let us formalise the problem of arranging travel plans using a constraint-based formalism. We define an itinerary as a sequence of legs or segments between different destinations: $itinerary = \{leg_0, leg_1, \ldots, leg_n\}$. Then, the CSP encoding the travel planning problem with $n$ legs is defined by a tuple $P = (X, D, C)$ where:

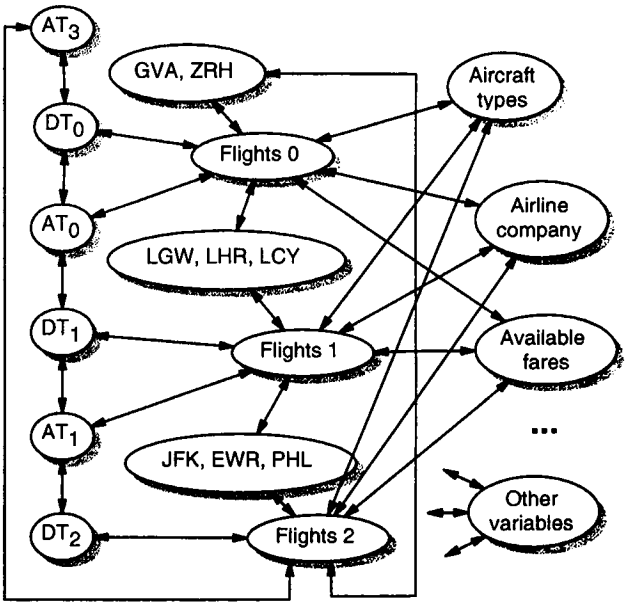


Figure 3: The constraint graph representing all the possible solutions for the given example. Nodes represent variables and the edges are the constraints. The constraints guarantee that the user's preferences are taken into consideration and the flight schedules are satisfied.

- $X = \{DT_0, \ldots, DT_n, AT_0, \ldots, AT_n, Airports_0, \ldots, Airports_n, Flights_0, \ldots, Flights_n, AirCrafts, Fares, Airlines, \ldots\}$ is a set of variables. There are several kind of variables:
  - $DT_i$ and $AT_i$ represent the dates and times where the traveller could depart and arrive respectively.
  - $Airports_i$ represents the possible airports near the destination of $leg_i$ of the itinerary.
  - $Flights_i$ stands for the possible flights in between the destinations of $leg_i$ and $leg_{i+1}$.
  - The problem can have more variables for encoding the user's preferences. For example, variable $AirCrafts$ is used for encoding the type of aircraft, variable $Fares$ for encoding the different types of fares, $Airlines$ for the different airline companies, etc...
- $D = \{D_1, \ldots, D_n\}$ is the set of domains. There are several kind of domains depending on the type of the associated variable:
  - For variables $DT_i$ or $AT_i$: the domain contains all possible departure and arrival times for the $leg_i$.
  - For variables $Airports_i$: the domain is a set of airports for the departure of the $leg_i$.
  - For variables $Flights_i$: the domain is the set of possible flights from $Airports_i$ to $Airports_{i+1}$.
  - For variables $AirCrafts$, $Fares$ and $Airlines$: the domain is the set of different aircraft, the set of available fares or the set of airline companies respectively.

- $C = \{C_1, \ldots, C_k\}$ is the set of constraints. Basically, there are two kinds of constraints: those imposed by the user's preferences and those imposed by flight schedules. There are constraints on the variables $Flights_i$, $Airports_i$, $DT_i$ and $AT_i$ that guarantee that the flight is compatible with the airports, departures times and arrival times. A binary constraint in between $AT_i$ and $DT_{i+1}$ takes into consideration that the flight for $leg_{i+1}$ departs after the flight for $leg_i$ arrives. Then most of the user's preferences are expressed by means of constraints between $Flight_i$ variables and $Aircrafts$, $Airlines$, $Fares$ and others.

In Fig. 3, we show the constraint graph representing the example described above.

The CSP in the smart client implicitly contains all 4470396 possible solutions. Because it runs locally on the customer's computer, these combinations can be searched using advanced techniques.

Once the customer has decided on a particular solution, the smart client generates a new request to the server, which can then initiate a booking process.

## Browsing solution spaces

Very often, the initial constraints given by the customer define a very large space of possible solutions. For example, even if I want to travel on a particular day, there is often a bewildering number of possible flights and combinations of them that I could take. While it is possible to optimize for a single criterion, such as price or travel time, this might cause the user to miss solutions which would have been preferred: a small increase in price might be acceptable in return for an advantage in another criterion. Interactive *browsing* is necessary to find the right solutions in such a multi-criteria problem.

Formulating solution spaces using the CSP formalism offers interesting possibilities for this browsing process. In particular, users can narrow down their choices by *posting* additional constraints on the solutions they see. These constraints could be formulated on any attribute or combination of attributes of the solutions. For example, in an airline travel problem, it would be possible to formulate constraints such as "minimum layover time has to be 90 minutes" and filter flight combinations in this way. Since the constraint satisfaction algorithms are completely uniform, such additional constraints can be incorporated into the search at any moment.

The smart client can thus implement approaches similar to that in (Linden *et al.* 1997), where sample solutions are proposed to the customer and stimulate formulation of additional constraints to rule out undesirable features. However, with smart clients this mechanism can be much more powerful, since it is not limited by the need to communicate with the central server.

When posted constraints become contradictory, techniques such as partial constraint satisfaction ((Freuder 1989)) can be used to help the user decide which constraints can be dropped to make the problem consistent again. In fact, within the constraint satisfaction framework there are numerous techniques for multi-criteria tradeoff which can be applied in this context.

## Networked Information Systems

Often, problem solving requires information from many different servers. For example, in travel planning I do not only have to consider the air travel, but also:

- schedules of ground transportation to and from airports,
- the availability of people that I want to meet, and
- my own availability.

Travel planning requires finding a combination of meeting time, ground and air transportation that meets the constraints of the participants as well as the transportation means.

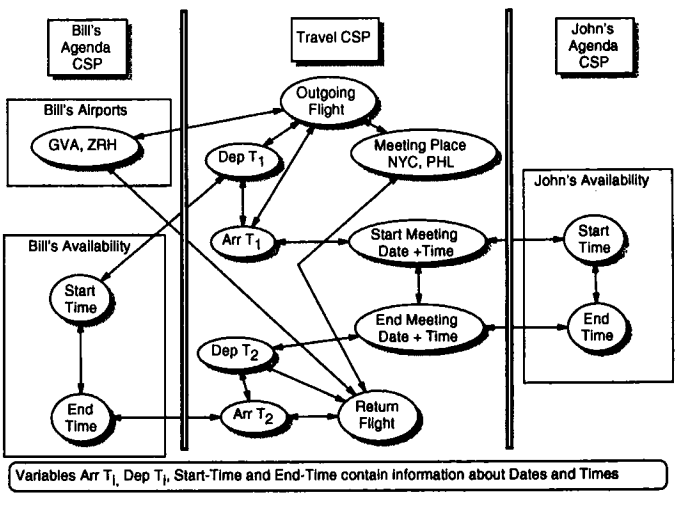


Figure 4: A CSP for a complete travel planner combining information from several servers. Ovals are variables, the arcs are constraints. Note that Bill is living in Bern (Switzerland), so he can leave from both Geneva or Zürich airports. On the other side, Bill can arrive at New York or Philadelphia because both are equally convenient for reaching Princeton. The Travel CSP uses same formulation of the problem such as the one described in Fig. 3.

One can figure out that information about schedules and agendas is available through different agents accessible through a network. The planning problem could then be solved in the following steps:

- gather information about schedules and availability from each of the servers,
- construct the combined problem, and
- solve this combined problem in a single process.

Here again, the formulation of solutions spaces as CSP can be of great help. The planning agent can

combine the CSPs obtained from different servers into a single CSP which represents the entire problem.

Consider the following example: Bill, living in Bern, Switzerland, has to arrange a 3-hour business meeting with John, who is located in Princeton, USA. Both have agenda agents which can be queried for time intervals when they are free, and the airline provides its schedules on a server. Bill can then leave the job of planning the entire trip to his agent. Fig. 4 shows the CSP that this planning agent would have to construct and solve. Any solution to this CSP will be a consistent combination of times and flights.

To construct this complete CSP, the planning agent will have to obtain parts of its from different servers: Bill's agenda, the airline, and John's agenda. If each of them can provide its information as a CSP, it becomes easy for the planning agent to compose them into a single problem. It would also be possible to integrate other CSPs representing for example, ground transportation, hotels, etc. The solutions to the combined problem can then be generated and browsed using a smart client just as in the case of a single server. It would be extremely difficult and inefficient to provide such a service without the CSP formalism.

## Conclusions

The modern world overwhelms people with massive information overload. We believe that Artificial Intelligence techniques have an important role to play in dealing with it. Up to now, much work has concentrated on techniques for retrieving or filtering information as such. We believe that the next step is to actively support the user in the problem-solving process. Only in this way can we achieve further substantial reductions in the complexity a user has to deal with.

Since most problem-solving is either compute- or knowledge-intensive, providing such functionality poses severe scalability problems. In an information system that has to serve thousands of users with small response times, the time that can be allotted to each individual user is very small. Smart clients offer a way out of this scaleability problem by making available the computation capacity of each user, thus linearly scaling the available capacity with the total load. The work we presented shows that contrary to what one might assume, this paradigm is very manageable for real applications. We hope to encourage others to investigate such an architecture for other problems as well.

Another important aspect of our work is the use of the constraint satisfaction paradigm. It allows us to represent complex problems and their solution algorithms in a very compact way. While constraint satisfaction is extreme in its compactness and simplicity, declarative techniques used in AI are in general more compact than traditional software. In networked environments where size of applications becomes important, this is another interesting feature which can be exploited in many other applications.

## References


Berthe Y. Choueiry. *Abstraction Methods for Resource Allocation.* PhD thesis, Swiss Federal Institute of Technology in Lausanne, 1994.

FIPA. *Foundation of Intelligent Physical Agents. http://www.fipa.org*, 1998.

Mark Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Morgan Kaufmann Publishers, Inc., Pitman, London, 1987.

Eugene C. Freuder. Partial Constraint Satisfaction. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 278–283. Morgan-Kaufmann, 1989.

Jango. *Excite Product Finder. http://www.jango.com*, 1998.

Grzegorz Kondrak and Peter van Beek. A Theoretical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence*, 89:365–387, 1997.

G. Linden, S. Hanks, and N. Lesh. Interactive Assessment of User Preference Models: The Automated Travel Assistant. In *Proceedings of Sixth International Conference on User Modeling*, 1997.

Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

Daniel Sabin and Eugene C. Freuder. Configuration as Composite Constraint Satisfaction. In *Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

Felix Freyman Sanjay Mittal. Towards a generic model of configuration tasks. In *Proceedings of the 11 $^{th}$ IJCAI*, pages 1395–1401, Detroit, MI, 1989.

A. Sathi and M. S. Fox. Constraint-Directed Negotiation of Resource Allocations. In L. Gasser and M. Huhns, editors, *Distributed Artificial Intelligence Volume II*, pages 163–194. Pitman Publishing: London and Morgan Kaufmann: San Mateo, CA, 1989.

Mark Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16(2):111–140, 1981.

Travelocity. *http://www.travelocity.com*, 1998.

Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London, UK, 1993.

Peter van Beek. *CSPLib : a CSP library written in C language.* ftp://ftp.cs.ualberta.ca/pub/vanbeek/software, 1995.