, , 1–21 ()

SmartClients: Constraint satisfaction as a Paradigm for Scaleable Intelligent Information Systems

MARC TORRENS marc.torrens@epfl.ch
Artificial Intelligence Laboratory, Computer Science Department,
Swiss Federal Institute of Technology (EPFL), Ecublens 1015, Lausanne, Switzerland

BOI FALTINGS boi.faltings@epfl.ch
Artificial Intelligence Laboratory, Computer Science Department,
Swiss Federal Institute of Technology (EPFL), Ecublens 1015, Lausanne, Switzerland

PEARL PU pearl.pu@epfl.ch
Database Laboratory, Computer Science Department,
Swiss Federal Institute of Technology (EPFL), Ecublens 1015, Lausanne, Switzerland

Editor: Peggy S. Eaton, Thom Fruewirth, Milind Tambe

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 *SmartClients*: 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.

Keywords: constraint satisfaction, software agents, electronic commerce, electronic catalogs, configuration, information systems

1. Introduction

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, most 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 informa-

tion 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 [6], or in the integrated travel information system designed by Siemens as a demonstration within the FIPA [3] 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.

Scaleability is a major point in this kind of application since web servers often have to deal with hundreds of users at the same time. Linden proposes in [8] a framework for planning air travel using constraints which has a lot of similarities with our work, but all processing takes place on the server and is thus not scaleable.

The paper is organized as follows. The next section describes briefly our technology called *SmartClient* and an example of how to apply this technology to the travel domain. Section 3 describes the use of Constraint Satisfaction Problems for *SmartClients* and the main architecture of our technology. In section 5 we describe a prototype called the *Air Travel Planning System* which uses *SmartClient* concepts. Concluding remarks can be found in section 5.

2. Motivation

SmartClient technology is based on constraint satisfaction techniques. In this section we briefly introduce the basis of Constraint Satisfaction Problems (CSPs) and present the *SmartClient* concept. Next, we describe how to apply *SmartClient* to the travel domain and to more general networked information systems.

2.1. Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSPs) are ubiquitous in applications like configuration [12, 11], planning [14], resource allocation [2, 13], scheduling [4] 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 for this application are the following:

- It offers a general framework for stating many real world problems in a succinct and compact way.
- Solutions can be found using simple and formal algorithms.

- There are numerous methods for resolving conflicts, visualizing solutions, characterizing the solution space, and transforming CSP into other forms.

A finite, discrete Constraint Satisfaction Problem (CSP) is defined by a tuple $P = (X, D, C)$ 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 ([17]).

2.2. SmartClients

An important 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 implement complex behaviors with reasonable efficiency, and
- constraints allow representing complex information in a compact form.

As a result, *SmartClients* 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 constraint satisfaction problem and solved by incremental constraint posting. However, one can imagine similar *SmartClients* for designing and planning applications.

2.3. Travel Planning

Arranging a trip when we have to meet a variety of people in different places involves numerous constraints from many different information servers: people's agendas, airline schedules, server for fares, etc. Let us consider for the moment only constraints for a single server, the airline schedules server.

In the current state of affairs, schedule information can only be obtained by queries made by 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 Air Travel Planning (ATP) system is a kind of personal assistant designed to facilitate arranging this kind of trips using the concept of smart agents. The prototype is described in detail in Section 4.

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 constraints 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 on the dates which are specified in the main query. Thus, a system considering all the possibilities from the example described above will access the flight information for the initial query as follows:

1. 1^{st} leg from Bern to Princeton: flights from `ZRH/BSL/GVA` to `JFK/EWR/PHL` on the dates from 1^{st} to 10^{th} February,

2. 2^{nd} leg from Princeton to London: flights from `JFK/EWR/PHL` to `LGW/LHR/LCY` on the dates from 4^{th} to 12^{th} February, and

3. 3^{rd} leg from London to Bern: flights from `LGW/LHR/LCY` to `ZRH/BSL/GVA` on the dates from 6^{st} to 14^{th} February.

Considering only 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 analyze 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 using conventional approach versus *SmartClient* agents.

Table 1. Conventional systems vs. SmartClient systems. The data of the table is calculated considering that the user is interested in all possible combinations of flights that match the initial query of the example mentioned before.

	Conventional Information Systems	Smart Client Agents
Server access	447039	1
Size of 1 transfer	60Kb	500 Kb
Size of total transfers	26.82 Gb	500 Kb

2.3.1. 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, simply allows the user 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 and finally put together a solution by hand - not a very satisfactory way of planning such a trip.

Fortunately, tools such as Travelocity [16] 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.

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

First of all, the customer specifies initial constraints about the trip: the departure and arrival airports, departure dates, and some general preferences. The *SmartClient* then collects information about *all* flights which could be part of a solution in one single server access. Since the information is encoded as a Constraint Satisfaction Problem (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 ($795 * 80 bytes$). Added to this the size of the **J**ava **C**onstraint **L**ibrary (100 Kbytes) and the graphical interface (300 Kbytes), the complete *SmartClient* is no larger than 500 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 through a standard modem.

The CSP in the *SmartClient* implicitly contains all 4,470,396 possible solutions. Because it runs locally on the customer's computer, the best combinations (according to some user's preferences, as described in Section 3.5) can be searched using advanced constraint satisfaction techniques.

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

2.4. Networked Information Systems

Often, problem solving requires information from many different servers. For example, in travel planning I 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.

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. Figure 1 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 the CSP 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

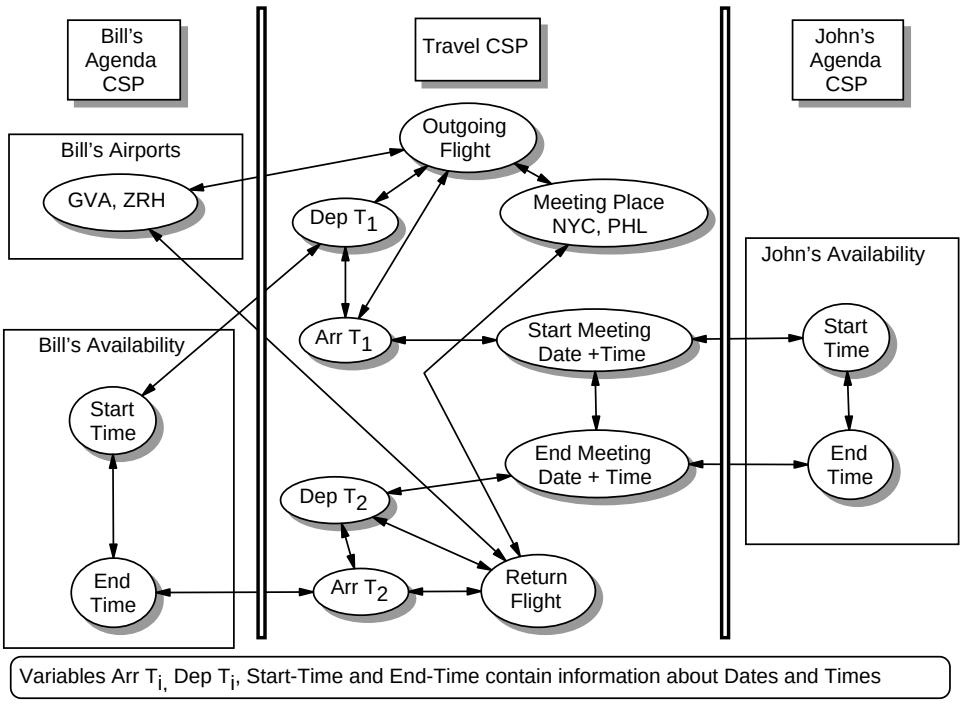

Figure 1. A CSP for a complete travel planner combining information from several servers. Ovals represent the variables and the arcs the constraints of the CSP. Note that Bill is living in `Bern` (Switzerland), so he can leave from both `Geneva` or `Zürich` airports. On the other side, Bill can arrive at `New York` or `Philadelphia` because both are equally convenient for reaching `Princeton`.

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 *SmartClient* 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.

3. Constraint Satisfaction for SmartClients

In this section, first we describe formally how we model the travel planning problem as a CSP. Then, the architecture for supporting *SmartClients* is described. Subsection 3.3 focuses on the **J**ava **C**onstraint **L**ibrary. The rest of the section is dedicated to explaining how the user deals with the *SmartClient* by posting constraints in order to get good solutions.

3.1. Modeling the travel planning problem as a CSP

Let us formalize 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: $itinerary = \{leg_0, leg_1, \ldots, leg_{n-1}\}$. Then, the CSP encoding the travel planning problem with n legs is defined by a tuple $P = (X, D, C)$ where:

- $X = \{DT_0, \ldots, DT_{n-1}, AT_0, \ldots, AT_{n-1}, Airports_0, \ldots, Airports_n, Flights_0, \ldots, Flights_{n-1}, AirCrafts, Fares, Airlines, \ldots\}$ is a set of variables. There are several kind of variables:
  - DT_i and AT_i represent the dates and times on which the traveler could depart and arrive respectively.
  - $Airports_i$ represents the possible airports near the departure for leg_i of the itinerary.
  - $Flights_i$ stands for the possible flights in between the airports of $Airports_i$ and $Airports_{i+1}$.
  - 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... These constraints can involve the whole itinerary or only a specific leg of the itinerary.
- $D = \{D_1, \ldots, D_n\}$ is the set of domains. There are several kinds 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 aircrafts, 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 other variables.

In Figure 2, we show the constraint graph representing the example described above.

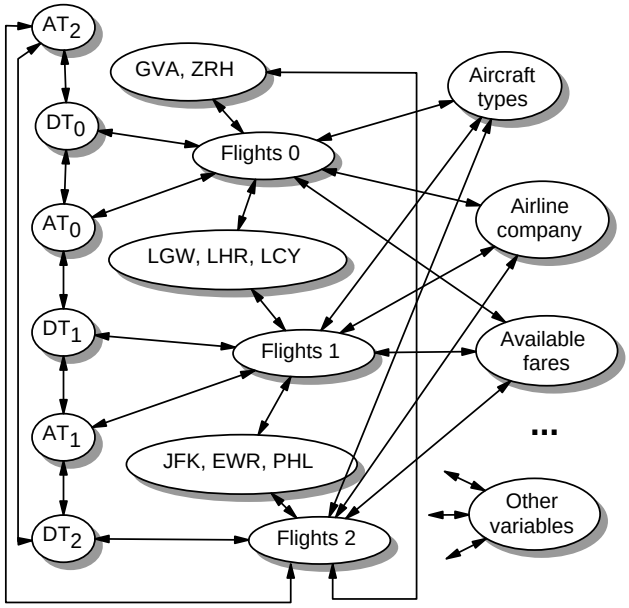

Figure 2. The constraint graph representing all the possible solutions for the given example. Nodes represent variables and the edges represent the constraints. The constraints guarantee that the user's preferences are taken into consideration and the flight schedules are satisfied.

3.2. Architecture[1]

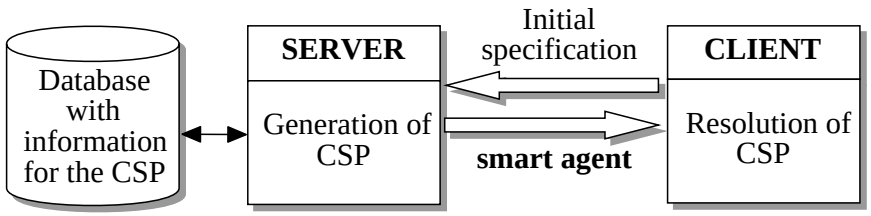

Figure 3. SmartClient architecture. The CSP is generated on the server side from the initial user query. Then, the CSP is sent back to the client side where the user's computer can solve the problem locally without further need for data from the server.

The architecture supporting smart-agent methodology is shown in Figure 3. The client sends a request containing the user constraints to the server. The server

accesses 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, the *SmartClient*, 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 variables of the CSP depend on how the problem has been modeled. The domains of variables must be searched for in a database or an information system. In our architecture, the exact queries to the databases depend on the query of the user. For example, in the domain of air travel planning, if the user wants to flight from `Geneva` to `London` on the 2^{nd} or the 3^{rd} of July, then we will search for flights on these dates going from `Geneva` to `London`. Our prototype for planning flights uses the Galileo reservation system which is used by Swissair and other major airlines. The advantage of building the CSP on the server side and sending it to the client side is that the CSP is a compact representation of all solutions that the problem can have given the initial restrictions of the user. So, the CSP can be transferred quickly.

- 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 it is no longer necessary to access the server, so the agent is completely autonomous

*3.3. **J**ava **C**onstraint **L**ibrary (JCL)*[2]

We implemented the **J**ava **C**onstraint **L**ibrary (JCL [15]), which allows us to package constraint satisfaction problems and their solvers in compact autonomous agents suitable for transmission on the Internet. It provides services for:

- creating and managing discrete and binary CSPs, and

- applying preprocessing and search algorithms to CSPs.

The JCL can be used either in a stand-alone Java application or in an applet (an *applet* is an application designed to be transmited over the Internet and executed by a Java-compatible Web browser). The purpose of the JCL is to provide a framework for easily building agents that solve CSPs on the Web. The JCL allows the

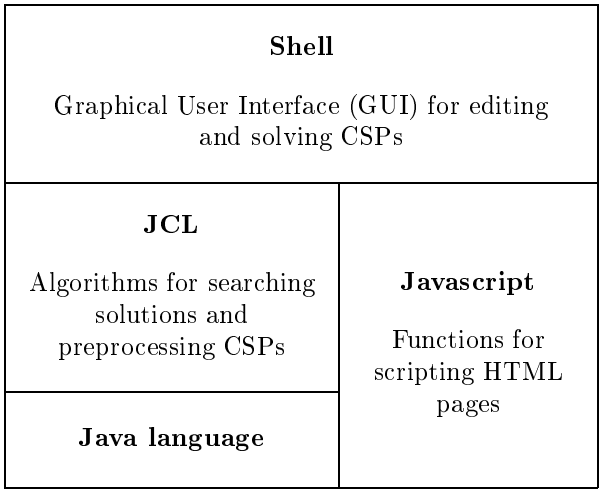

Figure 4. The components of the JCL environment. The JCL is composed by two main elements: the Constraint Library and the Graphical User Interface. The Constraint Library is implemented in Java being able to be used in applets. The Shell is a GUI for the library that combines Java with Javascript allowing the system to present results in HTML pages.

development of portable applications and applets using the constraint mechanisms. It can be downloaded from `http://liawww.epfl.ch/~torrens/JCL`.

Basically, the JCL is divided into two parts: a Constraint Library available on the Web and a Shell with a Graphical User Interface built on the top of this library (see Figure 4).

3.3.1. The Constraint Library provides methods for building CSPs from scratch and solving them. The user only has to model the problem as a CSP and then use the methods from the JCL to build the CSP. Once the CSP is built, any of the algorithms implemented in the JCL can be used for solving the CSP. These algorithms can search for *one*, *all* or n solutions depending on one parameter that the user can set. Once the solving algorithm finishes, the JCL reports an enumeration of the solutions found and some statistics on the search. Statistics on the search includes the number of backtracks, the number of consistency checks, the number of instantiations, and so forth. The library contains both search and preprocessing algorithms. The search algorithms allow us to find solutions of a CSP, while the preprocessing algorithms are used to simplify a CSP by eliminating values and compound labels that do not affect its solutions. Several search algorithms are implemented in the JCL. There are three main algorithms derived from *Chronological Backtracking* (BT) that are: *Backmarking* (BM), *Backjumping* (BJ) and *Forward Checking* (FC). Combinations of these algorithms yield other algorithms (see [7] for more details on these combined algorithms) which are implemented in the li-

brary: *Constraint-directed Backjumping*, *Graph-based Backjumping*, *Backmarking with Backjumping*, *Backmarking with constraint-directed Backjumping*, *Backmarking with graph-based Backjumping* , *Forward Checking with Backjumping*, *Forward Checking with constraint-directed Backjumping* , *Forward Checking with graph-based Backjumping* and *MAC (Forward checking using arc consistency)*. Some combinations of them are implemented in CSPLib [18] and just adapted in the JCL.

Two preprocessing algorithms are implemented in the JCL: *Arc-consistency* (AC) and *Path-consistency* (PC) [9].

The JCL has been extended with methods for dealing with exclusion constraints (unary constraints) and with binary relations like $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$. In this way, the JCL supports the **C**onstraint **C**hoice **L**anguage which is an agent *content language* designed to explicitly support agent communication about CSPs [19]. Such language could be useful for *SmartClients* that combine several information systems. In the travel domain, we can imagine several *SmartClient* agents for configuring different parts of a travel (flights, hotels, events, etc). The communication among this agents could be done using CCL, and the solving process would be carried out by the JCL.

3.3.2. The User Graphical Interface (GUI) has been implemented for easing the integration of the Constraint Library into Java applications and applets where the results of the algorithms must be displayed graphically. We have also implemented an application and an applet for editing and solving general CSPs. The user can build variables, domains and constraints by means of a graphic interface. CSPs can also be saved to a text file following a concrete syntax, so they can be edited in any text editor. The functionality of the GUI allows the user to connect applets to HTML pages as well by means of Javascript.

The main goals of the GUI for the JCL are:

- test and demonstrate the library graphically and intuitively, and
- provide classes in order to easily allow us to integrate the solving algorithms with graphical applications or applets.

3.4. Representing solution spaces

Compacting solution spaces is a key point for SmartClient technology where solutions spaces are sent through the Web to the user's computer. In this subsection, the issue of how CSPs can help compacting solution spaces is discussed.

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 enumeration of all solutions would require $n \cdot d^n$ 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.

3.5. 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 can be formulated on any attribute 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 system at any moment, thus searching new solutions according to the new posted constraints.

The *SmartClient* can thus implement approaches similar to that in [8], where sample solutions are proposed to the customer and stimulate formulation of additional constraints to rule out undesirable features. However, with *SmartClients* this mechanism can be much more powerful, since it is not limited by the need to communicate with the central server.

3.6. Configuration constraints, user's preferences and criteria for optimality

User's preferences can be set by posting constraints on the attributes of any proposed solution. In our framework, user's preferences are modeled as soft constraints. Normally, there are also constraints that are predefined for optimizing some criteria. For example, users always prefer inexpensive products, thus the price could

be a criterion to be optimized. Configuration constraints are also needed in order to guarantee the correctness of the solution. The following types of constraints are identified:

- *Constraints for configuration*: constraints that never can be violated. These constraints guarantee the feasibility and the correctness of a solution. For example, if we want to go from `Geneva` to `Paris` for one day, the return flight must be taken after the outgoing flight has arrived to `Paris`.
- *Constraints for preferences*: constraints for expressing user's preferences. By mean of these constraints we express the explicit user's preferences, for instance, the user could prefer to fly with `SwissAir` rather than with `Lufthansa`.
- *Constraints for optimization*: constraints for determining the quality or optimality of a (partial) solution. Some optimality criteria which are valid in general can be predefined. For example, users don't like to stop in intermediate airports or buyers want to pay as less as possible for a concrete good.

The next subsection describes the algorithms which allow us to find solutions where the CSP is overconstrained by minimizing the conflicting constraints.

3.7. Dealing with overconstrained situations

Through constraint posting, the user can quickly arrive at a situation where there is no longer any solution satisfying all constraints. Such situations have been extensively studied within the constraint satisfaction framework, and there are several alternatives for dealing with them:

- in *partial constraint satisfaction* ([5]), we find solutions where some variable assignments violate constraints. The advantage of this framework is that it is simple to implement and gives intuitive results. The disadvantage is that we might also obtain inconsistent solutions, violating for example the constraint that we cannot leave a city before having arrived there.
- many frameworks have been proposed for separating *soft* and *hard* constraints. For example, Borning ([1]) orders constraints in a hierarchy of importance; the best solutions are those which satisfy all constraints up to a maximal level in this hierarchy. This framework lets us avoid inconsistencies, but the solutions are less intuitive to users: we would, for example, prefer to violate a constraint on the preferred airline 10 times rather than violate two different preferences on departure times. Also, it is difficult to establish an ordering of constraints which reflects the user's preferences.
- the user can be asked to *retract* certain constraints. Such retraction can be chronological, or it can be supported by an explicit analysis to find minimal conflicting sets of constraints. This approach is the most satisfying one, as the user can get an explanation such as "you cannot leave later than 8 am and travel with your preferred airline". However, it involves numerous difficulties: the

number of conflict sets can grow exponentially with the number of constraints, it is computationally expensive to compute then, constraints and conflict sets need to be displayed and chosen from, etc.

In light of this discussion, our prototype performs partial constraint satisfaction taking into account the penalties of the violated constraints, and also allows chronological retraction of constraints by the user when (s)he is not satisfied with the solutions obtained in this way.

4. The Air Travel Planning (ATP) prototype[3] [10]

We have implemented a prototype for Air Travel Planning (ATP) using SmartClient technology. In most current travel e-commerce sites, the traveler needs to enter dates, times and destinations and is then directly led to a small choice of flights. In our approach, the buyer initially specifies only a set of possible destinations and ranges of dates. The catalog then proposes a large set of possible products, using three different displays:

- an overview display which allows comparing the entire range of solutions according to selected criteria,
- specific example products to elicit further constraints on their attributes, and
- a visualization comparing small sets of alternatives in their attributes.

In all these displays, buyers identify their needs (constraints) and evaluate them constantly until they reach an optimal solution. Flexible interaction sequences are supported through the use of constraint satisfaction as a basic selection mechanism (as described in previous sections). This allows us to model a buyer's criteria accurately and explicitly. Constraints can be posted and retracted in any order, resulting in a flexible conversation that rapidly leads the buyer to refine his needs. Constraint satisfaction paradigm provides support for visualizing and comparing the entire space of possibilities, thus ending up with a solution that appears to be the best deal and that she/he is thus ready to buy.

4.1. Defining initial needs

Buyers enter the interaction with often very vague ideas of what their actual needs are. In air travel, they usually know where and roughly when they want to travel, but they do not think of many other secondary criteria, such as departure times, airports and airlines they like to avoid, etc.

We use a world map as a metaphor for defining origination and destination airports. As a user zooms in, detailed information appears, such as each country's contours and the names of available airports. Clicking any name will enter the corresponding airport into the itinerary definition panel located on the upper-left corner (see Figure 5). The system automatically selects airports which are really close to the airports entered by the user.

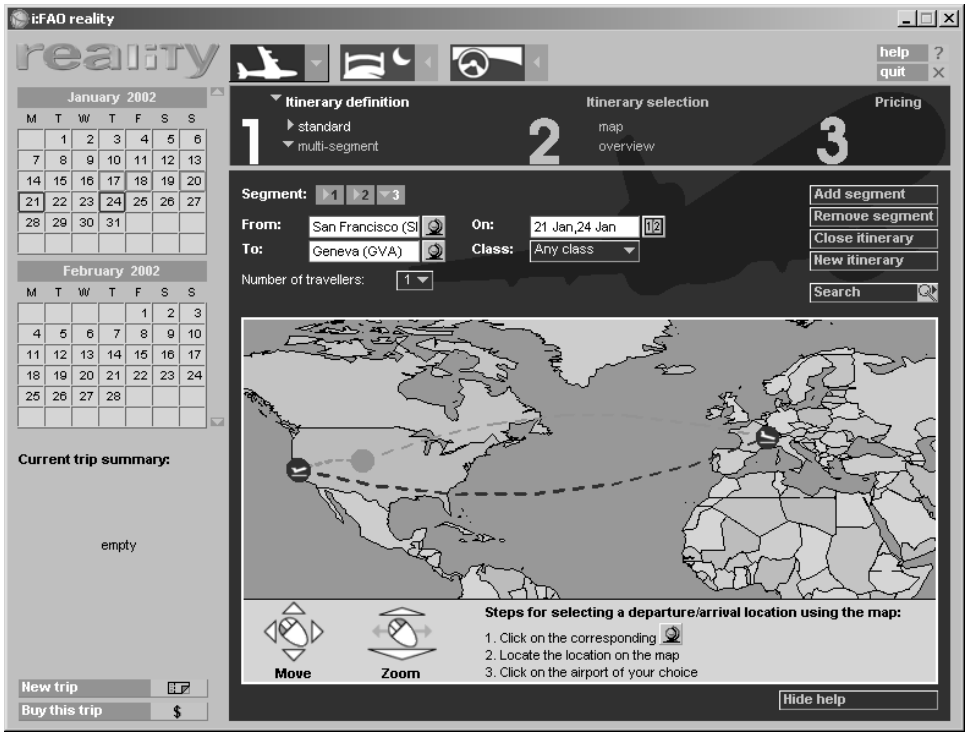

Figure 5. Query display with world map in the initial panel. The user can input his/her initial preferences about the travel, basically the destinations and the dates.

The *Search trips* button generates a solution space which is then shipped to the customer's side, whose constraints and preferences are used as guidelines to define an initial focus on the solution space.

4.2. Getting an overview of the available products

Often, the range of choices that a buyer might consider is bewildering. When there are many competing and possibly conflicting evaluation criteria, there can be a huge number of relevant choices, each optimal for some of the criteria and suboptimal for others. For example, the cheapest flight may require three plane changes. In trip planning, the complexity and richness of such decision problems are further compounded by the conditional nature of users' criteria. That is for certain flights, they would prefer the cheapest, while for others, they would prefer non-stop feature.

Such multi-criteria analysis can be performed in the overview display (see Figure 6), showing a scatter-plot of a sample set of solutions according to total time (horizontal axis) and departure time for segment 2 (vertical axis). In the shown ex-

ample, the scatter-plot is useful to see that if the user wants to leave in the morning for the segment two, he has basically two options:

- to leave on the 20th of January, or

- to leave on the 18th of January but take a solution that takes about 3 hours more than the others.

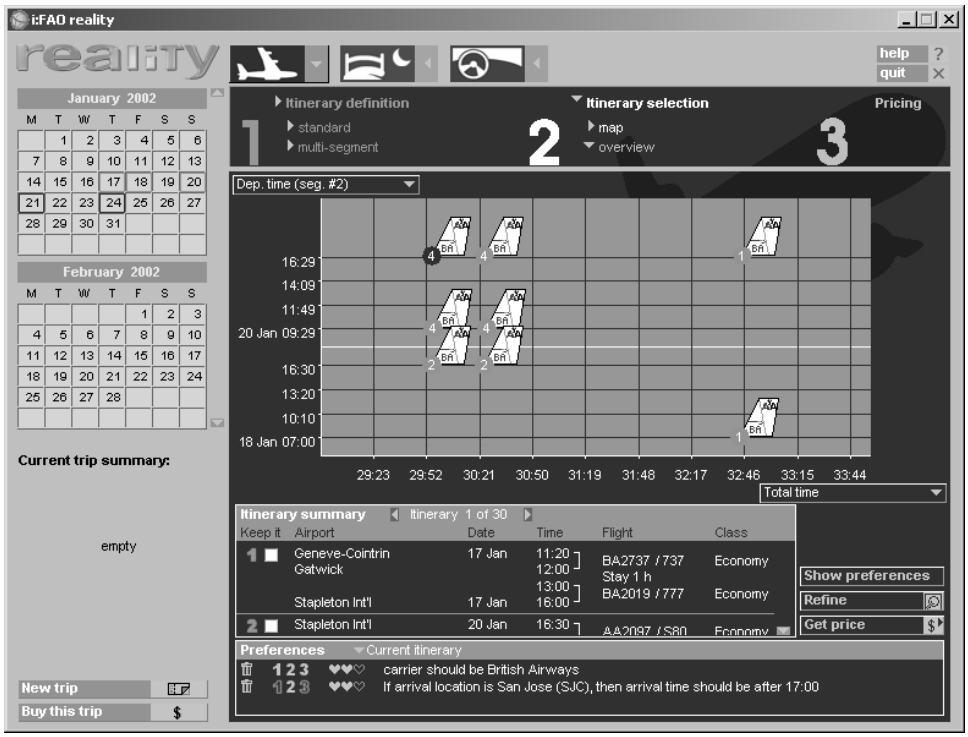

Figure 6. Tradeoff between departure time for segment 2 and total flying time in the *overview* panel. It is possible to inspect each of the possibilities in the *Overview* panel, and use this to make an initial choice which we then further inspect in the other views or panels. Selected flights are displayed in detail on the *flight table* below the graphical plot.

Solutions can be selected and are then stored in the *Current trip summary* panel.

At any moment during the solution space navigation and browsing, users can go to the *Overview* panel to further compare trips. Overviews can be provided for any combination of price, total flying time, number of intermediate stops, departure/arrival times for any segment, and so on.

4.3. Eliciting further needs and constraints

A typical buyer has many constraints that are not stated up front. The user becomes aware of these only when solutions are proposed that violate them. The *Edit preferences* panel, shown in Figure 7, allows us to post constraints on any item in the display. In the textual display, they can be posted by clicking on the respective cell and thus activating a menu, as shown in the figure. Similarly, it is possible to post constraints on any of the following solution attributes:

- price,
- airlines,
- aircraft types,
- departure and arrival dates,
- departure and arrival times,
- intermediate airports, and
- direct of non-direct flights.

When constraints are posted in this way, they are automatically restricted to the context in which they were posted. For example, if the user posts a constraint on a departure time, it will by default be applied to flights for that particular leg and leaving from that particular airport only. Applicability can be further restricted by selecting cells as a context, for example only when leaving from `San Francisco` airport because of the longer driving time.

Constraints can also be posted using sliders in the graphical tracer display above the textual display, which is discussed in more detail later.

Posting constraints in this manner eliminates one major difficulty with conversational interfaces: it makes it impossible for the user to input constraints that cannot be understood. Since the display does not show attributes or values that do not exist, it is not possible to post constraints on them.

At any time, the user can request the system to compute a new set of solutions that satisfy all constraints posted so far, and will usually obtain immediate response.

4.4. Comparing alternatives in detail

The tracer display (see Figure 7) shows each solution as a trace through the set of flight attributes comprising a trip itinerary. For each attribute there is one vertical bar with its possible values. A solution is a trace that links the values of the different attributes. The tracer display quickly makes apparent the differences among a set of possibilities. The tracer display also allows us to post additional constraints by using sliders on each attribute. This gives users who prefer to work with more graphical abstractions another way of declaring needs and criteria to the system. Sliders on attribute bars can further provide rapid specification of ranges

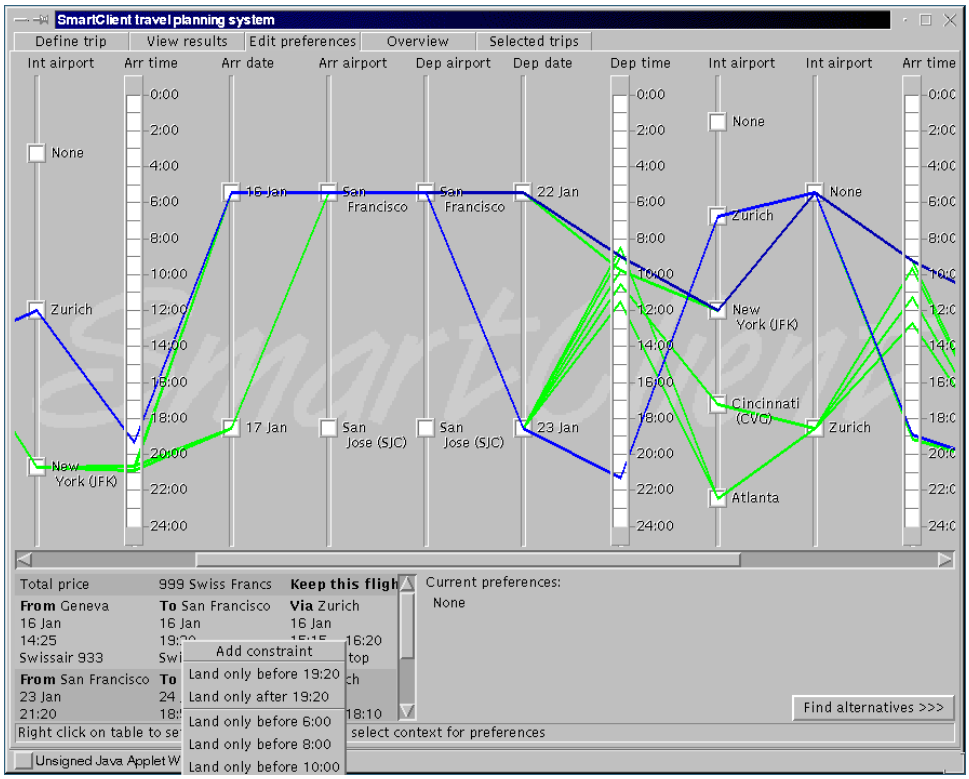

Figure 7. Edit preferences display allows the user to post preferences on any attribute of any displayed solution. Each solution is shown as a trace through the set of flight attributes. Constraints can be posted by using sliders and check boxes on the attributes.

of dates and time. Clicking on the other hand allows easy interaction for choices, such as whether someone wants to stop in a specific intermediate airport or not.

As an alternative way, this interaction design is particularly useful when travel is less constrained, allowing buyers to quickly decide that they can only expect small differences in cost, but potentially large gains in travel time. This mechanism can help users define criteria to find the "good deal".

5. 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 present here 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 since constraint satisfaction is extreme in its compactness and simplicity. In networked environments where the size of applications becomes important, this is another interesting feature which can be exploited in many other applications.

SmartClients are a new way of applying constraint satisfaction techniques. Rather than focusing on efficient solvers, we exploit the possibilities of representing spaces of solutions as constraint satisfaction problems. This possibility is important for information systems, where problem solving is based on comparing or synthesizing large numbers of solutions from a set given by an information system server.

Acknowledgments

We thank the Swiss National Science Foundation for funding the initial phase of the SmartClient project. i:FAO Switzerland SA (formerly Iconomic Systems SA) developed the second generation software architecture and user interface, and is currently commercializing the software.

We would like to thank all the i:FAO Switzerland SA team members for the great implementation of the commercial travel planner using SmartClient concepts: Christian Frei, Sebastian Gerlach, Patrick Hertzog, Denis Lalanne, David Nemeshazy and Loic Samson. Networkers Interacrtive SA (www.networkers.ch) designed the layout of the software.

Notes

1. Note that SmartClient architecture is protected by a pending patent.
2. The JCL is distributed in open source software under the terms of the LGPL (Lesser General Public License Version 2).
3. A long version of this chapter can be found in [10].

References

1. A. Borning, R. duisberg, B. Feldman-Benson, A. Kramer, and M. Woolf. Constraint hierarchies. In *Proceedings 1987 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 48–60, Orlando, FL, 1987.

2. Berthe Y. Choueiry. *Abstraction Methods for Resource Allocation.* PhD thesis, Swiss Federal Institute of Technology in Lausanne, 1994.
3. FIPA. *Foundation of Intelligent Physical Agents. http://www.fipa.org*, 1998.
4. Mark Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Morgan Kaufmann Publishers, Inc., Pitman, London, 1987.
5. Eugene C. Freuder and Richard J. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58(1):21–70, 1992.
6. Jango. *Excite Product Finder. http://www.jango.com*, 1998.
7. Grzegorz Kondrak and Peter van Beek. A Theoretical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence*, 89:365–387, 1997.
8. 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.
9. Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.
10. Pearl Pu and Boi Faltings. Enriching buyers' experiences: the SmartClient approach. In *CHI: Conference on Human Factors in Computing Systems*, The Hague, The Netherlands, 2000.
11. 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.
12. Felix Freyman Sanjay Mittal. Towards a generic model of configuration tasks. In *Proceedings of the 11^{th} IJCAI*, pages 1395–1401, Detroit, MI, 1989.
13. 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.
14. Mark Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16(2):111–140, 1981.
15. Marc Torrens, Rainer Weigel, and Boi Faltings. Java Constraint Library: bringing constraints technology on the Internet using the Java language. In *Working Notes of the Workshop on Constraints and Agents, Tehnical Report WS-97-05, AAAI-97*, Providence, Rhode Island, USA, 1997.
16. Travelocity. *http://www.travelocity.com*, 1998.
17. Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London, UK, 1993.
18. Peter van Beek. *CSPLib : a CSP library written in C language.* ftp://ftp.cs.ualberta.ca/pub/vanbeek/software, 1995.
19. Steve Willmott, Monique Calisti, Boi Faltings, Santiago Macho Gonzalez, Omar Belakdhar, and Marc Torrens. CCL: Expressions of Choice in Agent Communication. In *The Fourth International Conference on MultiAgent Systems (ICMAS-2000)*, Boston, USA, 2000.