Reactive Web Agents with Open Constraint Programming

Kenny Q. Zhu Wee-Yeh Tan Andrew E. Santosa

Roland H. C. Yap

Department of Computer Science

National University of Singapore

S16, 3 Science Drive 2, Singapore 117543

Email: {kzhu,tanwy,andrews,ryap}@comp.nus.edu.sg

August 15, 2000

Abstract

This paper discusses our current work on applying constraint programming systems as information back-ends for web agents. We present a conceptual framework as well as implementation based on Open Constraint Programming (OCP) to realize a highly reactive web-based system that can respond to high-order implications of the information that is supplied to the back-end.

Web agents are agents that serve clients across the World Wide Web. Such agents are able to reach out to more users than most traditional network agents. In our framework, instances of web agents talk to an OCP environment through a stub called the reactor. The reactor is responsible both for the communication between the web-agents and OCP, and for probing OCP when and if some constraint is satisfied (or unsatisfiable). This has several benefits: (1) Less programming effort due to that it is unnecessary to hard-code relationships between agents as in conventional approaches; (2) We can be sure about the state of all agents by consulting an OCP server; (3) Direct addressing of agents is not necessary.

In this paper we mainly discuss the implementation, as well as an example of a stock trading system.

Keywords: *web agents, multi-agent systems, constraint programming*.

1 Introduction

In the age of the Internet and electronic commerce, World Wide Web is playing a extremely crucial role. It is estimated more than 95% of all Internet users use WWW and the total number of internet users is growing at an approximate rate of 50% per annum. On the other hand, e-commerce is picking up its speed since 1998 and it is now on an explosive trend, with total consumer-based e-commerce revenue quadrupled to around 200 billion US dollars, and business to business revenue increased by almost tenfold to 1.25 trillion US dollars, by the year 2004 [10].

The potentially immense market and huge profit draw a lot of research attention into this area. Central to the e-commerce research is the idea of transaction between agents. Distributed agents are independent intelligent entities that are each loaded with some small programs to accomplish certain tasks [12]. Web agents are similar entities launched through web browsers or web servers, e.g. CGI scripts or Java applets. In particular, the agents query a common database or multiple databases, or submit a request that results in a change of the database. Each query or request as well as its reply from the data store is a transaction. A transaction could be an online purchase at an online store, or registration of a free email account, or checking the exchange rate at a bank's web site. Traditionally, transactions are sequential, which means the data store can only service one request at a time, and agents only interact with each other in a sequential manner. This is certainly inadequate with more sophisticated e-commerce applications. For example, in auctioning of goods online, a user may launch an agent that requests for some merchandise at a particular price. If the product is not available at the moment, the user would usually prefer to have the agent remain with the auction system and notify him or her when the bidding is completed, rather than submitting the agent repeatedly to check if the bidding is successful. Problems like this are reactive and concurrent in nature, and are not possible with traditional database transactions without heavy coding.

This paper presents a framework of reactive constraint programming through web agents, and the applications associated with it. Under this framework, users are able to submit their queries or constraint rules to a web interface concurrently, which in turn sends these requests to an *Open Constraint Programming Server* (*OCPS*). These concurrent requests interact with a single constraint store while preserving the atomicity and synchronization in the transactions. The key idea is that the *reactor agents* are spawned from a single web CGI program and establish socket communication with the OCPS. Communication channels are only existent during the submission of a query and the answering to a query, and not while the solving is taking place in the constraint server, hence it is reactive.

In this paper we first position our research among other work in Section 2 by providing comparisons. We then describe our framework in more detail in Section 3. and the corresponding implementation structure that supports it in Section 4. Several applications is discussed in Section 5, with main focus on a stock trading system. Before concluding our work in Section 7, we describe the current status of our research and some future works in Section 6.

2 Related Work

Much research in inter-agent communications focuses on language aspects, such as in *Agents Communication Languages* (*ACLs*) [8]. Our approach differs from theirs because the most fundamental of it deals with the contents of messages sent between agents. In this sense, one my find analogy between our research and the well-known contents language *Knowledge Interchange Format* (*KIF*) [2], however, what makes ours different is the embedding of *constraints* in the messages.

The closest sibling of the OCP project is the *Business Rules for Electronic Commerce (BREC)* [4]. The overall mission of BREC is to develop a reusable technology for business rules and rule-based intelligent agents. This technology is currently embodied as an extensible structured Java library, called *CommonRules*. Fundamental frameworks for business rules interoperability, their conflict handling, and procedural attachments are provided. These frameworks include an XML interlingua for communication and a declarative knowledge representation. There are several notable distinctions between BREC and our research:

1. BREC focuses on reuse of rules rather than constraint solving;

2. Courteous Logic Program deals primarily on conflicts of rules;

3. BREC does not propose an infra-structure for concurrent access.

In general, BREC addresses the multi-agent problem in a different angle, that is, the re-usability of rules, instead of openness and programmability as in the case of OCP.

In another work by Mikael Berndtsson [1], a *reactive object-oriented database* was described. It associates rules in database with events, so that a event will trigger the firing of a rule. The system, however, is not open and not meant for concurrent programming environment.

3 OCP as A Framework of Communicating Agents

Open Constraint Programming (OCP) is a programming model proposed by Jaffar, *et al*. [6]. This model is particularly useful for concurrent programming of highly reactive processes or agents. Reactive agents, which can be embedded in a piece of program code, are able to interact with each other through a constraint store or a solver. The main components of the OCP models are a shared constraint store, the notion of reactors, general use of agents and a trigger model. The main advantage of OCP over other Constraint Programming model, such as Concurrent Constraint Programming (CCP) [11], is it is open. The agents are completely detachable from the constraint store as their creation and termination are independent of the store. On the other hand, the agents are correlated with each other because one agent's constraints may only be solvable when the store has been altered by another agent.

Agent-based systems are most suitable for deployment in highly reactive environments where interactions between agents and their reactions best models the dynamism of the environment in which they operate. The advantages of software agents has been discussed in [7].

On the other hand, constraints are rules that specify relationships between different entities that must hold true. The rules, which usually describes real world constraints or constraints that model the problem, are specified by the programmer. The facts are things that we know are true. The application of the constraints to the supplied facts gives the constraint system the ability to reason about the implications of the given facts. This extends naturally as a means of specifying relationships between different cooperative agents in complex systems.

In this paper we mainly propose a system architecture in which constraints-based messages are used in communication. Our architecture employs constraint store as a common knowledge base. This architecture is the emphasis of this paper.

The OCP approach allows multiple agents to interact with each other through *reactors* to a shared constraint store. Using reactors, an agent may specify:

1. *watchdogs* to have the OCPS performs some actions or signals a set of agents when a condition is satisfied, or

2. *facts* which expresses the *current state* of the agent. The facts could be in the form of mathematical expression. The assertion of a fact could trigger some watchdogs whose conditions become true.

The use of constraints in the OCP framework will extend the use of software agents. The agents acquire ability to extrapolate from facts and rules available in the central store, within the domain of the constraint system used. This has several benefits:

Loop $\{$

If *waitcond*$_1$ $||$ *waitcond*$_2$ $||$ ... $||$ *waitcond*$_n$ $\Rightarrow$ **Do** *action*,

Watching *watchcond*$_1$ && *watchcond*$_2$ && ... && *watchcond*$_m$

$\}$

Figure 1: OCP Reactor Language

1. Without use of constraints, either the agents have to hard-code all the possible relationship or the central store has to assert all possible permutations of the relationships, most of which never used.

2. There is also an important benefit which has not received much attention from researchers: OCP allows one to be sure about the state of all agents. An administrator who has the privilege to change the constraint store may specify *invariants* that must be satisfied by all agents. In the analogy to the human society, these invariants act as the law. For example, the law must protect a store (e.g., an online store) such that nobody could take anything for free.

3. Explicit knowledge of agents that are target of invocation is not required. Invoking agents post facts about processing they have done and the target agents just pick them up from the constraint store.

Reactor language is shown in Figure 1. Here, *waitcond*$_1$ through *waitcond*$_n$ are the blocking conditions in disjunction; *action* is usually an assertion or a retraction operation that has an impact on the constraint store. *watchcond*$_1$ through *watchcond*$_m$ are watch conditions. *Action* is only executed if any blocking condition is true and no watch condition is true. The watch conditions, if fulfilled, cause the expulsion of the reactor from the OCP system, regardless whether *waitconds* are true or not. This simple reactor is already capable of handling large variety of problems involving constraints.

The users should be given a limited, non-destructive set of constraint language to start with. To

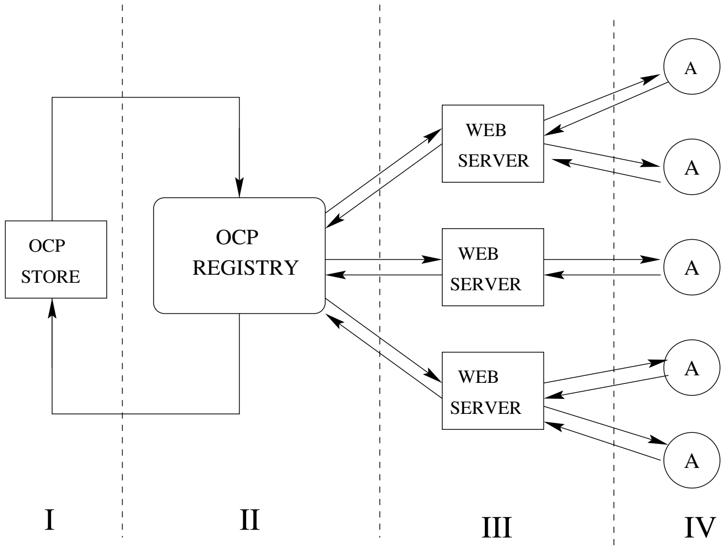

Figure 2: OCP's Four-Layered Architecture

the very least, users should not be allowed to remove system wide protected predicates from the shared memory, as this may cause inconsistency in constraint solving, or other undesirable consequences.

4 The OCP System Architecture

4.1 A Four-Layered Architecture

In this paper, we are presenting the Web-agent-based OCP system in a four-layered structure, illustrated in Figure 2.

- **Constraint store**. The core of the structure is the *constraint store*. This is the shared memory of program agents. The operations on a constraint store are simple transactions such as asserting or retracting a rule or fact, and queries. Practically, such constraint stores are implemented as constraint program with base facts and rules as in Prolog. Typically, we implement the system such that the program uses CLP(R) [5] syntax. Solving goals submitted by the agents relies on the constraint solver, which also serves as a logical database.

- **Registry**. The constraint solvers are usually sequential. The second layer, *registry*, remedies this limitation of the solvers by providing concurrency control. The registry is responsible for the following tasks.

  1. The registry unpacks the requests from the reactors and translates them from OCP reactor language to a form that is understandable to the constraint solvers.
  2. It coordinates these requests and queues them up according to certain priority policy, and then feeds the requests one by one to the solver.
  3. It maintains a *delayed queue* that contains all the requests that cannot be completed.
  4. The registry is equipped with a trigger mechanism so that when certain condition of the store takes place, a corresponding number of delayed requests can be woken up and resubmitted for evaluation at the solver.

  Figure 3 shows a schematic of the OCP registry.

- **Web server**. The third layer is web servers. It is a medium between the OCP infrastructure and the user agents. In this implementation, a web server gets requests from web agents, invokes a CGI script that spawns a web agent process and passes this process the request (reactor) in OCP language. The web agent, in turn, communicates with the OCP infrastructure at the specific OCP server and port number and transmits the reactors to the OCP server. The important issue of the reactor is that it adds some OCP specific features to the agent and make this addition transparent to the web users.

- **The user interface**. The outer-most layer of our OCP web agent system is the user interface. Here privileged users have the options to start or shutdown the OCP server at a host and a port, while normal users are able to submit their reactor agent in OCP language.

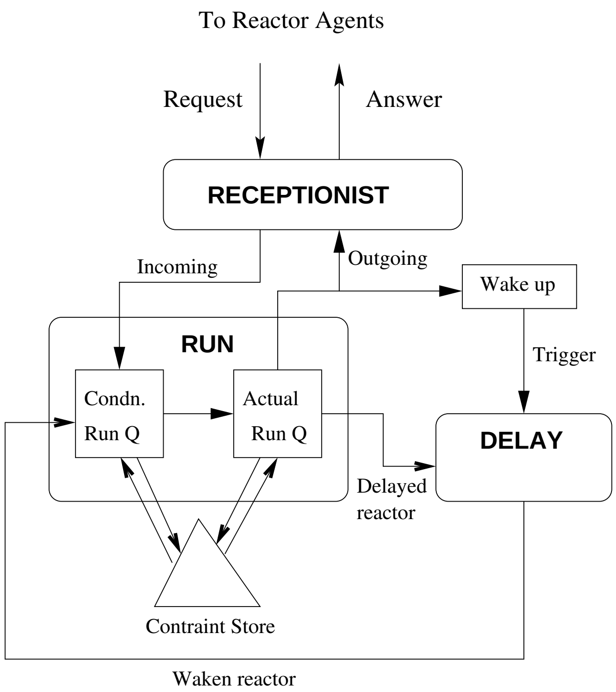

Figure 3: A Schematic of OCP Registry

With these four layers, we have the operational framework of OCP reactive system. Concurrent reactive agents can flow around in the system and make changes to the shared store if their wait conditions are satisfied. Such changes impact the behaviors of other agents, especially those delayed ones. Agents are terminated if their requests have been answered or when the watch conditions turn true and time them out.

4.2 The OCP Daemon

The OCP framework presented in Section 4.1 is functional only if the OCP server is up and running all the time. In a practical system, we certainly need some measures to control the startup and shutdown of the OCP server, and if possible, to provide a mechanism for privileged users to do administrative work on the OCP server. This gives rise to the idea of an OCP daemon. OCP daemon is a process that sits

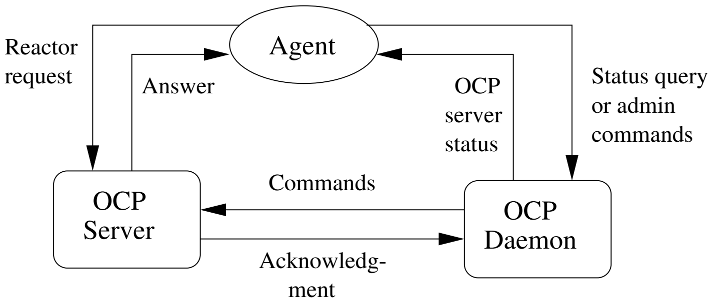

Figure 4: Meta-OCP Framework

in the background of the host and talks to the web server through socket connection. Special request of starting up, shutting down or checking for existence of the OCP server is sent to this daemon instead of the the OCP server. The daemon does the corresponding job if the authentication is passed. One important advantage of having an OCP daemon is now we can have multiple OCP servers running at different ports of same machine. The daemon manages these servers transparently from normal users. Reactive agents could then communicate with any of these independent shared stores interchangeably. The daemon itself is a very simple program which is generally fault tolerant, therefore in case one or more of the OCP servers crash, the users can comfortably rely on the daemon to get the servers back to work. A schematic of an OCP daemon in a meta-OCP scenario is exhibited in Figure 4.

5 Applications

In this section we illustrate some proposed examples. First, we discuss a simple stock trading system that we have implemented, and later discuss other web-based applications.

5.1 Stock Trading System

To demonstrate our proposal, we have implemented a simplified stock trading system. This system consists of a number of clients that are front-ends used by human stock traders, an OCPS, which is, in its current state, a structural wrapper to a CLP(R) interpreter, as we have explained, and an "external" stock market agent that keeps supplying the constraint store about changes in stock prices.

The human traders use the user interface of the clients to submit their trading strategies to OCP server. The server has the information on stock prices (supplied by the external stock market agents) and all traders' portfolio.

Number of stocks is the basic fact that corresponds to a trader. If the trader "Gekko" owns the stocks ABC, DEF, GHI, JKL, MNO, and PQR, then the OCP server will store these facts in the context of the trader. The CLP(R) syntax is as follows:

```
stock(gekko, abc, 30).
stock(gekko, def, 20).
stock(gekko, ghi, 25).
stock(gekko, jkl, 30).
stock(gekko, mno, 20).
stock(gekko, pqr, 250).
```

Using the above information and information on stock prices, traders are free to submit their trading strategies to the OCP server in the form of mathematical constraints. Watchdogs are used to notify traders when some supplied facts conflict with their strategies, or to do actions (e.g., buying or selling of stocks) when the supplied facts enable them to.

5.1.1 Bundles

We rarely look at isolated items when talking about stocks. In fact, we might want to group them according to their business focus, the countries they are incorporated in, or perhaps, even their risk factors. *Bundling* is a general mechanism for users to specify the grouping. The groupings in one's stock portfolio help an investor compare and contrast his or her holdings and diversify the investment.

For example, we may specify a *chip* bundle that collectively list all the companies that deals with micro-chips. We may also have bundle *finance* that refers to the companies in the financial sector. We can then argue about these stocks in our portfolio. A security manager, for instance, may want 50% of all his portfolio in the financial sector but does not want anything to do with the chip makers.

In the system, there are three types of bundles:

1. **System**: These bundles consist of attributes that are provided by the system as defaults. They include:

   - **All**: All listed stocks.
   - **Cash**: The cash value of the trader, viewed as stocks.
   - Each stock, which is a bundle of its own.

   System bundles only change when new stocks are introduced or when stocks are removed.

2. **Predefined**: These are also defined bundles but they are volatile bundles that can be added as the market trend shifts. For example, if there are sufficient demand for dot-com stocks, the administrator may add a bundle that contains all the listed dot-com companies. Predefined bundles are available only as a convenience to the users.

3. **User Defined**: These bundles are defined by the traders. This allows them to specify bundles for their own use.

The definitions of bundles are kept in the constraint store. Example definitions of bundles in CLP(*R*) syntax is:

```
rating(gekko, abc, lowrisk).
rating(gekko, def, lowrisk).
rating(gekko, ghi, lowrisk).
rating(gekko, jkl, lowrisk).
rating(gekko, mno, lowrisk).
rating(gekko, pqr, highrisk).
```

The above facts specify two non-intersecting user-defined stock bundles:

- Low-risk stocks, including the stocks with ticker symbols ABC, DEF, GHI, JKL, and MNO.
- A high-risk stock with the ticker symbol PQR.

Again, the client is free to change this information by submitting requests to the server.

5.1.2 Strategies

Having defined bundles, we may thus specify *strategies* that are to be applied on the bundles. Strategies are defined as mathematical constraints. A sample strategy is the one that is used to maintain a balance between high-risk and low-risk stocks. The constraints are as the followings:

```
100 <= port(lowrisk) (+/- 10%)

port(highrisk) = 1 * port(lowrisk) (+/- 10%)
```

Note that the function `port` computes the trader's portfolio of the stocks in a bundle.

The first constraint states that the portfolio of low-risk stocks should not be less than 100 dollars. A ten percent tolerance is specified to state that the rule can be violated by ten percent. For example, a portfolio of 90 dollars would be acceptable.

The second constraint says that the portfolio of high-risk stocks must balance the portfolio of low-risk stocks, again with ten percent tolerance. The use of tolerance is clearer in this case. Here we want both portfolios to be in balance instead of strictly having the same values.

As an example on how these constraints work, consider the time when a change in stock prices causes high-risk portfolio to be significantly greater than low-risk portfolio. A watchdog on the second constraint will notice this change, thus automatically perform the specified action, which could be the selling of high-risk stocks and/or the purchasing of low-risk stocks.

The determining of the stocks to purchase and/or sell can be guided by *objective function*, or be defined by traders. For example the "minimize transaction count" objective function will minimize the number of transactions to be performed. This function will most probably cause just single purchasing or selling being performed in restoring balance.

5.2 Other Applications

Apart from the stock trading system example we give earlier, OCP has many other interesting applications, especially in the e-commerce where the transactions are massive and volatile. These applications benefit from the constraint solving mechanism provided by OCP.

Online auction Auction is a process in which bidding price of an item keeps increasing at a potentially high frequency. Also the number of items on bid can be very large and one bidder is allowed to bid for or watch a number of offers at the same time. There is complex bid/offer relationship between the buyers and sellers. Most online auction sites including Ebay.com devised there auction rules to discourage users from overloading the system because they do not have a concurrent system to handle complex contraints. OCP, however, offers a handy way to control concurrent bidding, and allows users to specify their bidding strategy, such as "if an MP3 player's price doesn't exceed $120 and a SONY headphone cost less than $30, then I will bid for both and withdraw my bidding for the AIWA mini stereo." In addition, the OCP system can provide notification service when there is a merchandise in budget range.

The open market A market example is similar to auction with its unique features. Stock market is a special instance of open market. In a generic marketplace, there are products, price, rules and there are buyers and sellers, and their finances. Purchasing from a market can be complicated if these elements are inter-related or constrained. For example, buying from supplier X will be cheaper in bulk but with more defects than supplier Y. Also, the purchaser might have a choice to purchase either 12 PCs, or 5 to do the same job. There might also be supply problems such as a certain supplier is unable to deliver within some specified period. Such relationships often changes too rapidly to be hard-coded. This is a very typical optimization problem that a constraint based solution is most suitable for.

A possible automation of the above application in the OCP framework involves the suppliers launching agents to inform the central store about the current prices or perhaps even their pricing strategies. An agent can also be launched to watch the current prices available on the web, e.g. Priceline.com already has a simple version of that. The buyers, on the other hand, launch their respective agents to look for a purchasing solution.

Timetable schedulers Modern timetable schedulers are fairly basic. If one wishes to schedule an unusual event, the scheduler (possibly software agent) will launch an agent to look at everybody's schedule, and will return hopefully with a time slot that all participants of the event would agree to.

It is obvious that a later event might be adversely affected by this fixture even if a better solution exists. We really want to tell the event scheduling agents, the deadline, duration and list of participants for our event, and have them plan the optimal schedule for everybody. This is where the involvement of a constraint solver comes handy. Our proposed timetable scheduler will have a web-based front-end that collects data, both about a user's preference as well as the meeting specification. At any time, a page will show the users their updated schedule and the meeting chairman can learn about the time slots that are favored by the participants. Multiple meetings can be scheduled at the same time. The back-end is a constraint solver that tries to resolve all the constraints posted by the users. When a meeting is confirmed, the constraint solver updates the slot on the timetable of all the participants.

In conclusion, the aforementioned OCP applications have the following common properties:

- They consist of independent agents.
- The agents need to communicate in order to negotiate.
- There is need to reason from information provided by multiple agents.

6 Discussions

6.1 Current Status

A preliminary implementation of the OCP reactive system model has been realized. The core constraint solver is CLP(R) solver and the primary OCP language is based on CLP(R) language, too. The OCP daemon, however, was written in Perl. The server is currently hosted at guava.comp.nus.edu.sg (137.132.85.199). The OCP server , written in C++, is programmed to process many requests at the same time. In this implementation of OCP system, the registry does not send answers to every reactor automatically. Instead, the answer to the query is queued at the Receptionist awaiting to be polled by the original agent. This is generally acceptable as long as the polling frequency is not too high. The advantage of polling is that the internet socket does not have to be left open throughout the reactor processing time, thus the efficiency of the reception is improved. The trigger model is a simple one at the moment, under which all the delayed reactors are resubmitted to the solver whenever the constraint store is altered. However, to avoid too many reactors being woken up and crammed to the Run module, we brand our reactors as “impactive” and “non-impactive”. Only impactive reactors make changes to the solver database and triggers wake-up. The web interface[1] is simple and intuitive presently. One can submit one OCP reactor at a time, with a specification of whether the reactor is to be run once only or repeatedly. Repeating reactors are sent to delayed queue and wait for next evaluation. The system also maintains a log which documents all the system messages from *standard output* and *standard error*, for debugging purposes. At the moment, anyone can assert and retract anything to and from the store, so he or she can practically launch hostile agents that destroy the database. However, only users with a special password can bring down or restart the OCP server. We also have a working Perl reactor module with the capability to communicate with the OCP server without going though the web. Perl programmers are able to submit reactive agents directly from their concurrent programs using this module. This allows us to invoke large number of concurrent agents from different platforms to test the stability of the OCP system.

6.2 Future Work

Our future research will be focused on the improvement of the OCP system and the realization of practical applications. For OCP system, we have the following plans:

First, we need to include indexing mechanism for delayed reactors. The present practice of waking

[1]URL: http://guava.comp.nus.edu.sg/~kzhu/cgi-bin/react.cgi

up every delayed reactors in the queue is an intolerable system burden when the length of the queue grows bigger. Our experiment has indicated that a queue of 100 and above reactors already slows down the entire system remarkably. Therefore we need to index each delayed reactors and wake up only those reactors whose blocking conditions are likely to turn true when some changes happen to the database.

Second, most constraint logic solvers are compilers and not meant for large database purposes. Hence they are not particular strong in memory management of data. Some of them, e.g. CLP(R) does not garbage collection implemented hence there are intrinsic problem with code space starvation. We need to implement limited garbage collection with the existing solvers and we may have to incorporate relational database as our store.

Third, the system needs to be fault tolerant. If the OCP server crashes for some reason, it should be able to retrieve the CLP program and rewind to the state where it crashed. At the very least, it should prompt the user to re-submit their reactors if the reactors were lost in the crash. This can be achieved by periodically backup of the CLP program and logging the state of computing.

Finally, the security of OCP need to be strengthened. This comes in two-folds: the *system security* and the *language security*. In the system security level, we will make the OCP server only accessible to permissible users, and users will be categorized into varying levels of permission, like in Unix. Some of them, with administrative power, can start and close down the server and change certain system predicate; others can have either read or write access to the store. In the language security level, agents should only have access to a limited set of the predicate or grammar, depending on the owner's permission. Not only do we need to parse the agent's language to determine if it has the right to say it, we may also need to extend the OCP language to a more secured fashion to make the above security measures practical. In particular, we would add several system CLP(R) predicates to allow user agents define the permission level of their own data or predicates. In the first stage, users can at least protect their own predicates with CLP(R) "protect/1"[5]. In later stages, more sophisticated library that allows user and group permission will be introduced.

On the application side, we will experiment with our stock portfolio balancing application and further extend OCP to generic purchasing. Also we are seeing the possibility of introducing OCP into workflow applications and real-time traffic control.

7 Conclusion

We have introduced the structure of OCP framework. We have also described how this framework supports the use of a constraint logic store in a multi-agent environment, both as the infrastructure and as a synchronization mechanism, and how OCP can be extended to support multiple back-ends. The OCP framework in the context of web agents finds important applications in electronic commerce. A lot of large, dynamic and reactive problems can now be solved with the help of reactive databases (the constraint solvers) instead of relational databases. Traditional relational databases, such as Oracle, require immense efforts on tailor-made softwares to go with them when they are applied to differing applications. Constraint logic programs, however, can be easily composed or rectified to adapt to varying e-commerce scenarios. Our stock portfolio management application has proved OCP's capability of handling complex logic problems arising in multi-agent world.

Some of the pressing issues facing OCP research are reactor indexing, database memory management, solver fault tolerance and security. More topics will arise as our exploration goes deeper.

OCP allows web agents to exploit the declarative nature and the reasoning power of a constraint solver. We are most certain that OCP will be a thrusting technology for agent-based applications, especially those in the e-commerce, that are not realized at present now, either due to their complexity or that they are impractical to solve using existing methods. It will also allow existing applications to employ more complex inter-agent synchronization and expanding the horizon of existing constraint-based programs.

References

[1] Berndtsoon, Mikael, “Reactive Object-Oriented Databases and CIM”, the 5th International Conference on Database and Expert Systems Applications, p.769-778, 1994.

[2] Genesereth, M. R., “An Agent-Based Framework for Interoperability”, In Bradshaw, J. M. *Software Agents*, Chapter 15, pp. 317–345.

[3] Gray, J. S., “Interprocess Communications in UNIX”, Prentice Hall PTR, 1997.

[4] Grosof, B. N., Y. Labrou and H. Y. Chan. “A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML.” In Wellman, M. P. ed. *Proceedings of the 1st ACM Conference on Electronic Commerce (EC ’99), Denver, Colorado, USA, Nov. 3–5, 1999*. ACM Press, 1999.

[5] Heintze, N., *et al.*, “CLP(R) Programmer’s Manual Version 1.2”, September 1992.

[6] Jaffar, J. and R. Yap, “Open Constraint Programming, Invited Paper, *4th Intl. Conf on Principles and Practice of Constraint Programming (CP), Pisa*, Oct. 1998.

[7] Jennings, N.R. and M.J. Wooldridge, “Applications of Intelligent Agents.” *Agent Technology: Foundations, Applications and Markets*, p.3-28, 1998.

[8] Kone, M. T., A. Shimazu, and T. Nakajima. “The State of the Art in Agent Communication Languages”. In *Knowledge and Information Systems 2*(*3*), Aug. 2000.

[9] Marriott, K. and P. Stuckey, “Programming with Constraints. An Introduction”, MIT Press, 1998.

[10] “Newsweek International Edition”, September 20, 1999.

[11] Saraswat, V. A., “Concurrent Constraint Programming”. MIT Press, 1993.

[12] Wooldridge, M. and N. Jennings. “Intelligent agents: Theory and practice”, *Knowledge Engineering Review*, 10(2), 1995.