Coordination of Many Agents

Joxan Jaffar, Roland H.C. Yap, and Kenny Q. Zhu

School of Computing
National University of Singapore
Republic of Singapore
{joxan,ryap,kzhu}@comp.nus.edu.sg

Abstract. This paper presents a reactive programming and triggering framework for the coordination of a large number of distributed agents with shared knowledge. At the heart of this framework is a highly structured shared store in the form of a constraint logic program (CLP), which is used as a knowledge base and being reacted to by agents through the use of "reactors". The biggest challenge arising from such a reactive programming framework using CLP is to develop a trigger mechanism that allows efficient "wakeup" of blocked reactors. This paper addresses the architecture of this open framework, and discusses a general methodology for doing triggering of logic conditions using views and abstractions.

1 Introduction

In online applications such as an automated marketplace, many agents with shared knowledge need to interact and synchronize with each other by reacting to some conditions. The agents block when the condition they are waiting for is not satisfied and unblock when the condition becomes true some time later. Because the number of agents participating in these activities is very large, and the blocking conditions may be very complex, existing technologies such as blackboard architectures [10] and active databases [15] are inadequate.

This paper introduces a shared-store programming framework for interacting distributed agents which combines the power of Constraint Logic Program (CLP) [8] and the *triggering* of complex conditions.

The framework allows agents to *react* to their environments taking into account complex conditions, and allows for coordination with each other through a structured shared store, the *knowledge base*, represented by a CLP. The distributed agents interact with the shared CLP store via embedded program fragments called *reactors* programmed in a simple stylized concurrent language. The key features of this language include the use of CLP goals as *guards* and the use of *committed choice*. CLP goals as guards give a unique way of handling reactivity in distributed programs which leverages CLP's power of declarative semantics, databases and complex *views*.

Since reactors may block on complex logical conditions and there are many of them, the key technical challenge is how to identify efficiently a set of reactors, among all the blocked reactors, that need to be checked for "wakeup" given a state change to the CLP store, e.g. update of a base predicate. We call this process *triggering*. This is more

difficult than determining when a guard becomes entailed in Concurrent Constraint Programming (CCP) [12] as the CLP program itself can change and the test is not simply entailment.

The idea behind efficient triggering is to exploit a notion of *locality*. To explain a simple form of locality, consider the popular online chat client, MSN messenger. While there may be millions of users logged on at the same time, every user typically only has a small contact list. When a user logs on (which can be viewed as a state change), it is easy to exhaustively search the vicinity of this user (in this case, her contact list) to find out which contacts are online and notify (trigger) those people. However, there are more complex forms of locality for which no such efficient algorithms are readily at hand. For example, a marketplace has *frequent* and *diverse* updates; an individual trader, however, is typically interested in a small fraction of these. Given an update, it is difficult to identify the few interested traders.

In this paper, we deal with more complex forms of locality. We employ a general assumption that *any single state change in the CLP is unlikely to affect a large portion of the blocked reactor*, therefore it pays to design an index structure for the reactor conditions so that triggering can be done efficiently as a search in the index. To facilitate the indexing of complex conditions, the trigger framework exploits the semantics of CLP views in *abstracting* complex views to simple ones. It is also possible to develop analysis tools with CLP technology to verify the correctness of such abstraction.

Since this framework is completely *open*, in the sense that any agents can interact with the system at any time, and CLP goals are used in reactive conditions, we call this framework *Open Constraint Programming* or OCP. It is also an open system in the sense that the agents are language neutral. We argue that open, reactive, and concurrent systems with powerful modeling capabilities are useful for distributed coordinated applications such as those in E-commerce.

This paper addresses the design of the architecture and the reactor language, and focuses in particular on a triggering framework and methodology which are essential for managing large number of reactors. The main contribution of this paper is two-fold:

- the use of CLP as a knowledge base and for reactivity in an open programming framework; and
- a general triggering framework for re-enabling a small set of reactors, among a large number of blocked ones.

1.1 Related work

The OCP framework is related to shared-memory concurrent languages such as CCP [12] and blackboard architectures [10]. In the CCP framework (including GHC [14] and Oz [13]), processes communicate by interacting with a set of shared variables in a *store* on which they can either post ("tell") or test ("ask") for the presence of some constraint. These languages also use committed choice for non-determinism. However, the constraint store is monotonic in CCP, whereas in OCP, the store is a knowledge base and can be non-monotonic. Just as in a database or blackboards, the store needs to be non-monotonic as it is meant to be stateful.

Blackboard languages such as Linda [7] and ActorSpace [1] use a set of tuples or actors as medium of communication and synchronization. OCP can be thought of as generalizing Linda's tuple space, to use more powerful constructs and going from a structured shared memory store to richer store which uses CLP. By doing so, the primitive operations on the store can utilize complex reasoning to express how the agents interact and synchronize. OCP is also related to active databases [15] because the ECA rules can be treated as the special case of a basic kind of OCP reactors. Triggering is required in both active databases and OCP, though the active database has a much simpler trigger paradigm, which is on simple events such as insertion/deletion to the base tables. In particular, active databases do not address our problem of minimizing the cost of triggering.

Another class of reactive languages are the synchronous languages such as Esterel [2], Lustre [3] and SIGNAL [6]. These languages are designed for reactive systems in which reaction is *instantaneous*. These synchronous systems are also *deterministic*, while CCP and OCP are non-deterministic.

In what follows, we will discuss the architecture of OCP, which includes the reactor programming language and motivating examples, the triggering mechanism which is critical for the efficient implementation of a runtime system, and finally a description of an initial prototype system with some experimental results.

2 Architecture

In the basic OCP framework (Fig. 1), every software agent consists of a program written in a suitably convenient language. Small program fragments, which we call *reactors*, written in an OCP reactor language, are embedded in the agent program. This is similar to how one can embed SQL in a host program. When the agent wants to execute a reactor, this reactor is submitted to the OCP runtime system, in a similar fashion to a remote procedure call.

The purpose of the reactor is to perform some actions to the shared CLP store or knowledge base. In the rest of this paper, we will use the notation Δ to refer to the shared CLP store (this suggests it is stateful). The action may be guarded by some conditions defined in the reactor as well as other logic defined in Δ. We expect that some part of the requirements for the condition may be expressed in the submitted reactor and the rest of it could be reasoning expressed in the knowledge base itself. We use a rather general form of the condition which is any CLP goal expressed in the language of the knowledge base. A typical action may be to update or delete some data in the store and the condition could be some consistency condition, e.g. ensure minimum balance in a bank account. The action can only be performed if the knowledge base Δ is consistent with the guard. When that is not the case, the reactor blocks until some change in Δ makes the condition true, we say the reactor is *re-enabled* and can be executed as long as the condition is true. One can express concurrent alternatives in a reactor, committed choice is used to control the non-determinism arising from the alternatives. As a reactor behaves like a remote procedure call from the agent program, it only returns when the submitted reactor has completely finished executing its reactor program.

Since the knowledge base Δ is non-monotonic, we can think of it as consisting of some static predicates which do not change and some dynamic predicates which can be changed by a reactor. In this paper, we consider dynamic predicates to be ground facts, which we call *base predicates*.

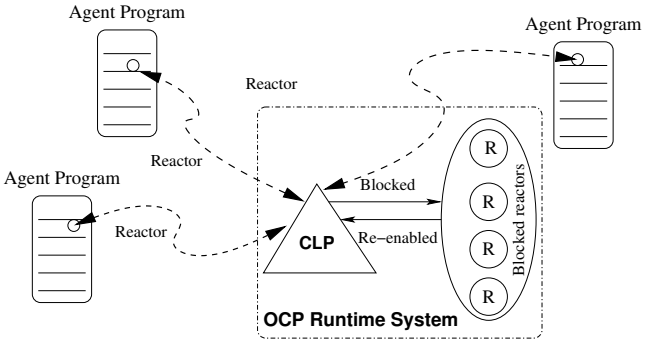

Fig. 1. The basic model of OCP

2.1 Syntax and semantics

We now give the inductive definition of a simple reactor language as follows. Let r be a reactor, then

r	$::=$	
	δ	atomic update to Δ
$\mid$	$r_1; r_2$	sequence
$\mid$	$r_1 \mid\mid r_2$	choice
$\mid$	`commit`	commit the enclosing choice
$\mid$	$c \Rightarrow \delta$	guarded actions

The reactors are intended to be embedded in agent programs. The sequence construct is self-explanatory, and we just focus on atomic update, choice/commit and guarded action. We define the operational semantics of these constructs using the reactor transition relations $r,\ \Delta \xrightarrow{\delta} \Delta',\ r'$, where δ is an atomic action that changes the store from Δ to Δ', and r progresses to r'.

– An atomic update is an action that consists of one or more of the following sub-actions in an *atomic* sequence: an insertion, a deletion or an update to a base predicate. Often the sequence is coded in a CLP goal to be executed atomically by the runtime system. While, in principle, these operations can be used on any predicate in Δ, in this architecture, we restrict the atomic updates to only base predicates. The rule for atomic update is

$$\delta, \Delta \xrightarrow{\delta} \delta(\Delta)$$

– The choice construct provides a form of early-committed non-determinism. The semantics of choice is that both branches execute concurrently until one of them makes an update to Δ or issues a `commit`. At that time, the other choice branch is aborted with no effect on Δ. The `commit` operation can be thought of as a special atomic update to Δ which has no effect, like a `noop`. Let ϵ denote an action that does not change the store, such a read; and let u denote an update that changes the store or a `commit`, the rules for the choice construct are

$$\frac{r_2, \Delta \xrightarrow{\epsilon} \Delta, r_2'}{r_1 \,||\, r_2, \Delta \xrightarrow{\epsilon} \Delta, r_1 \,||\, r_2'} \quad \frac{r_2, \Delta \xrightarrow{u} \Delta', r_2'}{r_1 \,||\, r_2, \Delta \xrightarrow{u} \Delta', r_2'} \quad \frac{r_2, \Delta \xrightarrow{u} \Delta'}{r_1 \,||\, r_2, \Delta \xrightarrow{u} \Delta'}$$

– Guarded actions are used for synchronization or for ensuring consistency conditions. c is called a *blocking condition* or simply a *condition*. A guarded atomic update, $c \Rightarrow \delta$, blocks until condition c is true, i.e. $\Delta \models c$, and then atomically performs the update δ. In particular, c is any CLP goal defined over Δ which is evaluated by the CLP system. Variables in condition c are in the same scope as the action and can be used to bind variables in the action. The rule for guarded actions is

$$\frac{\delta, \Delta \xrightarrow{\delta} \Delta'}{c \Rightarrow \delta, \Delta \xrightarrow{\delta} \Delta'} \quad \text{if } \Delta \models c$$

2.2 A motivating example

We use the following example of shipping marketplace as a motivating example for OCP throughout this paper. The agents interacting with the marketplace are clients who want to ship cargo and transportation companies which offer cargo ships of various load capacity and sailing schedules. The knowledge base is a CLP program which contains static facts of a distance table (map) among cities, and dynamic facts about the availability of the vessels, such as the following.

```
map(seoul, shanghai, 4668).
vessel('star', hongkong, shanghai, 20000, 0.012, 15, 18).
```

The `map` predicate records the distance between two cities, e.g. the distance between Seoul and Shanghai is 4668 km. The `vessel` predicate specifies a vessel named "star", which is scheduled to go from Hong Kong to Shanghai, with a load capacity of 20000 tons, and a shipping price of 1.2 cent per ton per km. It will depart at time 15 and arrive at time 18. The departure and arrival time along with the distance table implies travel speed of the vessel. We assume that the unit price for shipping is roughly proportional to the speed of travel.

A client wants to ship cargo from place A to B, either directly or via some other transit points, by a certain deadline and within budget. Constraints are on the load capacity of the vessel and the feasibility of arrival/departure times. The reactivity arises because it may not be possible to ship the cargo given the existing state of the store,

however, changes to the store may make the request feasible. Clients will update the capacity as they are committed to a particular vessel.

The relation (and a knowledge base) `deliverable` is used to specify blocking conditions c of the clients' reactors. It returns `Dep` and `Arr` as the departure and arrival times for tracking, and a list of vessel identifiers.

```
%base case for one segment
deliverable(A, B, Weight, Budget, Deadline, Dep, Arr, [ID]):-
  Budget>0, Weight>0,
  LoadCap>=Weight, Weight*Dist*Price<=Budget, Arr<=Deadline,
  map(A, B, Dist),
  vessel(ID, A, B, LoadCap, Price, Dep, Arr).

%base case for two segments: A-C and C-B
deliverable(A, B, Weight, Budget, Deadline, Dep, Arr, [ID1, ID2]):-
  Budget>0, Weight>0,
  LoadCap1>=Weight, LoadCap2>=Weight,
  Weight*(Dist1*Price1+Dist2*Price)<=Budget,
  Arr1<=Dep2, Arr2<=Deadline,
  map(A, C, Dist1), map(C, B, Dist2),
  vessel(ID1, A, C, LoadCap1, Price1, Dep1, Arr1),
  vessel(ID2, C, B, LoadCap2, Price2, Dep2, Arr2).

%recursive case: A...C-D...B
deliverable(A, B, Weight, Budget, Deadline, Dep, Arr, L):-
  LoadCap>=Weight, Weight*Dist*Price<Budget, Dep2>=Arr1,
  map(C, D, Dist),
  deliverable(A, C, Weight, Budget1, Dep1, Dep, _, L1),
  vessel(ID, C, D, LoadCap, Price, Dep1, Arr1),
  deliverable(D, B, Weight, Budget-Budget1-Weight*Dist*Price,
           Deadline, Dep2, Arr, L2),
  L=concat(L1, ID, L2).
```

There is more than one way to define `deliverable`. Here, we choose to define base cases of one segment and two segments, and then a recursive rule that consists of a path from A to C, a segment C to D and another path from D to B. The reason for this set-up is to have more efficient triggering which will become clear in the next section. When the `deliverable` condition is satisfied, the cargo can be shipped and the `do_ship` action will update the corresponding capacities along the route.

A client who wants to ship a cargo weighing 100 tons from Singapore to Seoul by time 25 and with maximum $5000, can submit the following reactor to the OCP system:

$$\texttt{deliverable}(singapore,\ seoul,\ 100,\ 5000,\ 25,\ D,\ A,\ IDs) \Rightarrow \\ \texttt{do\_ship}(singapore,\ seoul,\ 100,\ IDs)$$

From the above example, we argue that there exists a large class of applications like the shipping marketplace where the use of a CLP program as a knowledge base and for reactivity is not only elegant but, we believe, also essential. The recursion and constraint solving capability of a CLP offers an extremely concise but expressive way to specify

general logical rules, such as `deliverable`, to be used by many different reactors from different agents with their own instantiations or possible additional constraints.

2.3 The runtime system

This section describes the design of the runtime system architecture. Central to the system design is the notion of *triggering*. A trigger model determines which blocked reactors to *fire* given an update δ to the knowledge base Δ. When a reactor is fired, it is re-enabled and starts execution by re-evaluating the blocking condition which failed earlier, and if it succeeds, proceeds to executing the action.

To manage the execution, blocking and wakeup of the reactors, the runtime system employs a registry that wraps around the CLP system and the blocked reactors. The registry is composed of the following elements (see Figure 2):

1. a CLP program loaded in a CLP system;
2. a receptionist that handles the I/O of reactors; and
3. a trigger unit for triggering blocked reactors.

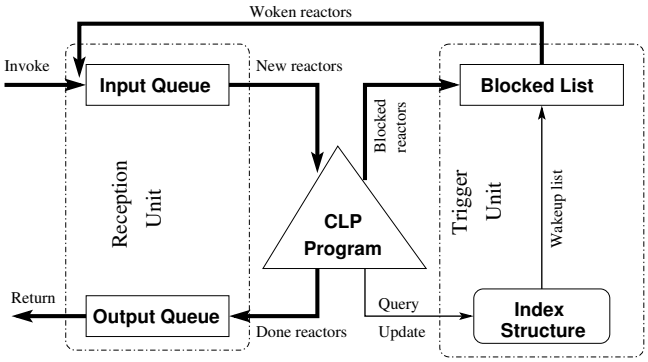

Fig. 2. The registry

The dark arrows in the figure represent the flow of reactors in the system. A reactor is executed against the CLP store once it enters the registry. The blocking conditions and the actions in the reactors are implemented as CLP goals which are executed by a CLP system. If the blocking condition is satisfied, the action is executed, probably updates the CLP and exits the system; otherwise, the reactor is blocked in the blocked list, until the condition becomes true in the future.

We suggest that this architecture is extremely versatile because CLP can integrate databases with logic programs, constraints and concurrency. At one end of the spectrum, a CLP program can be reduced to just ground facts which is similar to Linda. At the other end, it can be a full-fledged knowledge base complete with a reasoning system and constraint solvers.

3 Triggering

The problem of triggering is present in many applications and scenarios. Consider the popular online application MSN messenger. One of its features is when a user Jane logs in, all her contacts who are currently online must be notified, or *triggered*. As we know, MSN messenger has millions of users online at any time, certainly we don't want to test every online user to see if he or she is a friend of Jane. In this particular case, a simple hash-based triggering can be used as any user's contact list is typically small, in fact, bounded. But in general, the problem of determining just which agents are to be triggered by an often-occurring event is intractable.

In this section, we discuss the triggering problem for OCP, and present a methodology for dealing with it.

3.1 Views and blocking conditions

The basic problem can be defined as follows: given an update δ to a base predicate of the CLP knowledge base, and given a set of blocked conditions $\mathcal{C}$ which are currently false, efficiently return a subset of $\mathcal{C}$ which become true as a result of the update.

To facilitate discussion below, let us first define:

Definition 1 (View). *A* view *is simply a rule defining a distinguished set of non-base predicates. It has the general form:*

$$p(\tilde{X}_0) :- q_1(\tilde{X}_1),\ q_2(\tilde{X}_2),\ \ldots,\ q_n(\tilde{X}_n),\ \Psi(\tilde{X}_0, \cdots, \tilde{X}_n). \tag{1}$$

where p is not a base predicate. We say that this view is basic *if the $q_i, 1 \leq i \leq n$, are all base predicates. Otherwise, we say that the view is* composite.

Note that not all CLP predicates provide views. Views are essentially interface predicates for the agents to interact with the CLP program. The blocking conditions of reactors are defined based on views.

Definition 2 (Blocking Condition). *A blocking condition c is of the form:*

$$p(\tilde{X}),\ \Psi(\tilde{X}),$$

where p is a view on variables $\tilde{X}$, and $\Psi(\tilde{X})$ is a constraint. We say a blocking condition is basic (composite) if the view it refers to is basic (composite).

Typically, $\Psi(\tilde{X})$ specifies a value or a range for some of the variables in $\tilde{X}$, such as $c ::= p(X, Y),\ X = 5,\ 0 \leq Y \leq 5$.

Definition 3 (Induced View). *Let p be a view of the form $p(\tilde{X}) :- Body$. Let c be a blocking condition $p(\tilde{X}),\ \Psi(\tilde{X})$. The view of p* induced *by c is the rule*

$$p(\tilde{X}) :- Body, \Psi(\tilde{X}).$$

To determine if a blocking condition c on a view p is enabled by an update δ is in general an undecidable problem. Naively, one can execute c as a goal against the newly updated CLP knowledge base. This is tantamount to testing if the induced view of c has any solutions.

Running induced views is, unfortunately, unacceptable if c is a composite condition that depends on complex views whose resolution is very expensive, e.g. when recursive joins are involved. Preferably, we could discover some constraints, from the definitions of both c and δ, which could answer this question directly. This is clearly more desirable, and this optimization represents our first objective.

However, if the total number of blocked reactors is very large, even this optimization is insufficient, because having to consider every blocking condition is prohibitively expensive (recall the MSN example). We therefore seek to build an *index* for the blocked conditions so that large number of conditions can be excluded from an update without testing any one of them. Constructing this index thus becomes our second optimization objective.

In what follows, we will first show how to index basic blocking conditions by a spatial index structure called the *RC-tree*. We then show how to reduce composite views to basic ones so that the RC-tree can be used.

3.2 The RC-tree

This section considers the problem of indexing multi-dimensional geometrical objects. There is a wealth of publications in the the area of spatial databases [5] and computational geometry [11]. However, these spatial indexes, especially those used for geographic information systems applications, assume little or no overlapping among the objects, and when the objects are large, static segmentation is used to reduce large objects to many small rectangles, which increases the space and insertion cost. In addition, they index the Minimum Bounding Rectangles (MBRs) of the original objects, rather than the objects themselves. The original shapes of the objects are thus lost and not made use of in such approximation.

We propose a new spatial index structure, called RC-tree, which is better suited for indexing dynamic, overlapping regions. RC-tree is a clipping-based spatial index which combines some features of the kd-tree and the R^+-tree. Every intermediate node of a RC-tree is a hyper-plane that partitions the space assigned to this node. The space is thus divided into two sub-spaces. All objects entirely contained in the left half-space will be stored in the left sub-tree at the node; and all objects contained in the right half-space go into the right sub-tree. If an object intersects the hyperplane, it is clipped and the two resulting clipped objects go into respective subtrees where they belong. The root node is assigned the entire space.

The novelty of the RC-tree is that instead of indexing MBRs of the objects, it indexes the actual shape of the objects, and dynamically clips the objects on demand when there is need to discriminate a number of them. This enables the RC-tree to index objects of large extension and with heavy overlapping.

RC-tree's dynamic clipping can be seen as doing the segmentation dynamically and on demand. A very important technique used in the RC-tree is *domain reduction* which dynamically updates the MBRs of clipped objects such that insertion and search costs

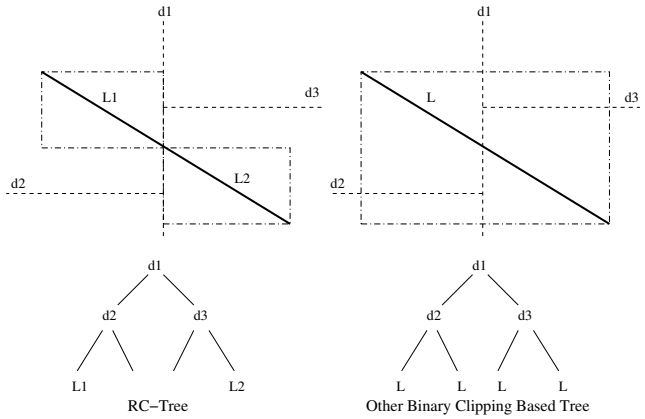

Fig. 3. Advantage of clipping and domain reduction in insertion

as well as space requirements are reduced. In the left part of Fig. 3, domain reduction strategy creates a tree with only two items ("L1" and "L2") in the leaves. In the right part of the figure, the other tree, which is similar to the R^+-tree, has four items (denoted by "L"). In addition, during the insertion, the RC-tree clips object "L" and inserts the two sub-objects "L1" and "L2" into the tree while the other strategy inserts four times.

We have conducted a range of traditional and synthetic benchmarks on the RC-tree, and have observed an amortized $\log(n)$ insertion time, amortized $\log(n)$ point query time, where n is the number of objects to be indexed. Our experiments also show that RC-tree performs much better in query performance than other R-tree variants and the quad-tree.

3.3 Basic views

The RC-tree can be used to efficiently index multi-dimensional shapes and to search using a query of the same dimensionality.

Let $\Psi(\tilde{X})$ be a constraint and q a base predicate. Then the basic condition $q(\tilde{X}), \Psi(\tilde{X})$ can be treated as a geometrical shape in an RC-tree for $\tilde{X}$. Accordingly, the update on $q(\tilde{X})$ where $\tilde{X}$ has been grounded can be treated as a query to that RC-tree. Therefore, reactor conditions on a single base predicate can be indexed and triggered using the RC-tree in a straightforward manner.

Consider an example of such a basic blocking condition:

```
vessel(_,A,B,C,P,_,_), A=beijing, B=taipei, C>=500, P<=0.02.
```

One can construct a 4-dimensional shape on (A, B, C, P) such that

$$(A =' beijing') \wedge (B =' taipei') \wedge (C \geq 500) \wedge (P \leq 0.02)$$

and index shapes like this in a 4-d RC-tree on variables (A, B, C, P). When a new vessel becomes available or an existing vessel changes, variables (A, B, C, P) get updated simultaneously. The ground values (A, B, C, P) can then be used as a point query to the RC-tree index. For example, an update of

```
vessel('dragon', beijing, taipei, 10000, 0.015, 23, 40)
```

is one of such updates that would enable the above blocking condition.

Another type of basic condition is of the form $p(\tilde{X}_0), \Psi(\tilde{X}_0)$, where p is a basic view, which means

$$p(\tilde{X}_0) \text{:-} q_1(\tilde{X}_1),\ \ldots,\ q_n(\tilde{X}_n),\ \Psi_0(\tilde{X}_0, \cdots, \tilde{X}_n),$$

where q_1 through q_n are all base predicates. Of course, one can immediately replace $p(\tilde{X}_0), \Psi(\tilde{X}_0)$ by

$$q_1(\tilde{X}_1),\ \ldots,\ q_n(\tilde{X}_n), \Psi'(\tilde{X}_0, \cdots, \tilde{X}_n),$$

where

$$\Psi'(\tilde{X}_0, \cdots, \tilde{X}_n) = \Psi_0(\tilde{X}_0, \cdots, \tilde{X}_n) \wedge \Psi(\tilde{X}_0).$$

For example,

$$c ::= q_1(X),\ q_2(Y),\ X + Y = 10,\ X \geq 0,\ X \leq 5.$$

The condition c can be formulated as a shape in the (X, Y) space, and an RC-tree can be built for shapes in the (X, Y) space. The problem is, updates are only on `q1/1` and `q2/1` separately, which means either X or Y is updated at a time, but not both. Therefore, we cannot construct a complete query of (X, Y), but instead we have either $(x,\ *)$ or $(*,\ y)$, where * denotes unknown values. There are two possible ways to solve this problem. The first and the "default" method is to construct a range query using a wildcard for the variable that is not instantiated. For example, if `q1(5)` is written to the CLP, a query $(5,\ *)$, which is essentially an infinite range query to the RC-tree, can be produced to query the index tree. This method is applicable but not always effective. Suppose the blocked conditions are all binary constraints on X and Y, and there is no bound on X, then $(5,\ *)$ will not be able to discriminate any shapes in the index, and hence all conditions will be triggered.

The second way is to instantiate or constrain some of the unknown variables at the time when an update occurs. This is possible if there exists in the CLP a constraint or functional relationship between the value of the known variable (X) and the value of unknown (Y). For example, if the following rule exists in the CLP:

```
q2(Y):- q1(X), Y = 2*X+1.
```

Then given $X = 5$, the system can infer by the above rule that $Y = 11$, and thus produce a complete query $(5, 11)$.

Alternatively, if there exists a constraint between X and Y such as,

```
q2(Y):- q1(X), Y<=X.
```

then a finite range query can be produced: $((5,\ Y) :\ Y \leq 5)$, which is more specific and effective than querying with $(5,\ *)$.

We conclude this subsection with a few comments on the issues of *aggregation* and *materialization*. Aggregation is a concept that originates from the relational databases. An aggregate is a function of some tuples in the same relation. Common aggregation

functions such as *min*, *max*, *average* can be computed incrementally and are included in some versions of CLP as system predicates or as meta-level predicates. In the shipping example, the prices and speeds of vessels vary over time, but their ranges can be defined as aggregates of the `vessel` predicate using the *min*/*max* functions.

When a blocking condition contains a view that uses aggregation, how do we deal with it? One way to handle aggregation is to materialize the value of aggregates if they are not changed often, such as price ranges of all vessels. Once the aggregate values are materialized, they can be treated as constants and used in the basic views. However, when the aggregate value *does* change later, the views constructed based on the materialized values must be updated. This may involve deleting of the corresponding shape from the index, reconstructing it and then re-inserting it into the index.

3.4 Composite views

Having shown how to index and query basic conditions in the last section, this section considers a methodology to reduce composite conditions to basic conditions so that they can be handled like in Section 3.3.

The essence of our method is to translate the definition of the composite view p at hand into a basic view. There are two ways, which can be repeatedly interleaved, to progress towards this.

The first and obvious way is to perform an *unfolding* of the definition of p. Clearly unfolding alone cannot, in general, obtain a basic view, because of recursion.

The alternative way, which represents the main contribution of this section, is to replace the remaining non-base predicates in the definition by an *abstraction*, that is, a sequence of other predicates and constraints, in such a way the resulting definition of p is *at least as general* as the original definition. Though this step is seemingly difficult, it may be the case in applications that the abstraction is in fact evident from the domain. We shall demonstrate this below; meanwhile, we shall call this methodology an *application-based abstraction*.

For example, for a view $p(\tilde{X})$ whose recursive definition refers to a base predicate q, it is possible to unfold $p(\tilde{X})$ a number of times such that q is *exposed* along with some subgoals of p:

$$p(\tilde{X}) :\!- p(\tilde{X}'),\ q_1(\tilde{X}_1),\ p(\tilde{X}'),\ \Psi(\tilde{X}_1).$$

Now if one can replace the two $p(\tilde{X}')$'s with a constraint and combine it with $\Psi(\tilde{X}_1)$ to obtain:

$$p'(\tilde{X}) :\!- q_1(\tilde{X}_1),\ \Psi'(\tilde{X})$$

Then $p'(\tilde{X})$ is an abstraction of the recursive view $p(\tilde{X})$.

We now give a more concrete example of abstracting the view `deliverable` in the shipping example of Sec 2.2. We shall however simplify the relation `vessel/7` to three arguments: source, destination and cost. We further assume that the value of cost, for each (source,destination) pair, is precisely the distance between them according to `map/3`. We also simplify `deliverable/8` to three arguments: source, destination and budget. Fig. 4 shows the simplified definition of `deliverable`.

```
deliverable(A, B, Budget):-
  vessel(A, B, Cost),
  0<Cost<=Budget.
deliverable(A, B, Budget):-
  vessel(A, C, Cost1),
  vessel(C, B, Cost2),
  0<Cost1+Cost2<=Budget.
deliverable(A, B, Budget):-
  deliverable(A, C, Cost1),
  vessel(C, D, Cost2),
  deliverable(D, B, Cost3),
  Cost1+Cost2+Cost3<=Budget.
```

Fig. 4. Simplified `deliverable`

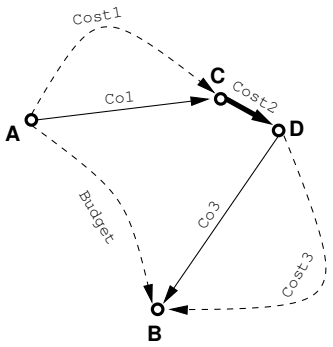

Fig. 5. Abstraction of `deliverable`

By inspecting the third rule of `deliverable`, one can see that `deliverable` is the transitive closure of `vessel` and thus the cost of a direct vessel from point A to C and from D to B is no bigger than cost of the transitive closure. In other words, given `map(A, C, Co1)` and `map(D, B, Co3)`, `Co1` $\leq$ `Cost1` and `Co3` $\leq$ `Cost3`, where `Cost1` and `Cost3` are defined in Fig. 4. We illustrate this scenario in Fig. 5, where the dark arrow refers to an actual vessel, the light arrows refer to straight distances (map), and the dashed arrows are the transitive closure of vessels (deliverable). Therefore we can abstract `deliverable` to a basic view `abs_deliverable` as follows.

```
abs_deliverable(A, B, Budget):-
  Co1+Cost2+Co3<=Budget,
  map(A, C, Co1), vessel(C, D, Cost2), map(D, B, Co3).
```

The above abstraction captures the intuition that if a new vessel segment (from C to D) is too far away from both the source (A) and destination (B) of a reactor, then this reactor should be excluded from triggering. In other words, we are exploiting "locality" of the source and destination in this example.

We summarize our methodology of abstraction as follows.

- it reduces composite views to basic views so that direct indexing can be done on the basic conditions;
- it provides a *conservative* estimate of the original view, ie. the set of facts satisfying the abstraction is a *superset* of that of the original view. Thus an update that does not trigger an abstracted condition does not trigger the original blocking condition. In other words, exclusion of reactors by abstraction is *safe*.

It is, of course, undecidable in general to replace a composite view with an equivalent one, while trivial to replace it with an abstraction. The challenge is to find a *useful* abstraction. While it is not possible to characterize this condition formally, we suggest that if the application intuitively satisfies the condition that an individual reactor is not likely to be triggered by an average update, then it is likely that a desired abstraction can be discovered without great effort. We have tried to indicate this with the example above.

4 Implementation and evaluation

We have implemented a prototype OCP system. Rather than using a tailor-made CLP system, our prototype for simplicity integrates the CLP($\mathcal{R}$) system [9] with a server registry explained in Section 2.3 that manages a collection of reactors especially to handle triggering and communicates with external agents. The multi-threaded registry was implemented in C++. The OCP registry and the CLP($\mathcal{R}$) system communicate through Unix message queues.

Agents are written using the language Python which is a relatively rich and extensible scripting language. A special Python reactor library handles the submission of reactors to the OCP system.

4.1 Trigger efficiency

To evaluate the effectiveness of triggering using RC-tree, we conduct the following experiments on a Pentium 4 2.4GHz PC running Linux 2.4.20. We implement the shipping marketplace example in Section 2.2 for transporting cargo between a set of 7 Asian and 6 North American cities. The distance matrix among these cities is approximated by the flight distances between them [4].

We identify two types of reactor blocking conditions and two types of vessel updates: *intra-continental*, *inter-continental*. For instance, a reactor waiting to ship a cargo from an Asian city to an American city is inter-continental, whereas a reactor waiting to ship within two Asian cities is an intra-continental reactor. Similar definition applies to the available vessels. We thus created three sets of reactors (1000 reactors in each set): intra-continental only, inter-continental only, and mixed; and similarly three sets of vessel updates (1000 updates in each set): intra-continental only, inter-continental only, and mixed. We use the abstraction in Section 3.4 for triggering the reactors. We did 9 experiments, corresponding to the 9 possible combinations of sets of reactors and updates. The experimental results are given in Table 1. The first number in each data cell is the average percentage of reactors being triggered out of 1000 blocked reactors in each scenario. The second number in parentheses is the time for the triggering mechanism to determine which reactors to be fired with a sequence of 1000 vessel updates. All times are measured in milliseconds.

	reactors(intra)	reactors(inter)	reactors(mixed)
vessels(intra)	**10.176%** (12.6ms)	37.743% (13.2ms)	23.964% (12.9ms)
vessels(inter)	**0%** (8.7ms)	33.893% (13.1ms)	16.937% (12.7ms)
vessels(mixed)	**4.364%** (12.4ms)	33.694% (13.2ms)	19.025% (12.8ms)

Table 1. Hit rate and average trigger time

From Table 1, we see that for intra-continental reactors, the best case for triggering excludes all reactors from wakeup (intra-column, row 2). This is intuitive because long-

haul voyages are more expensive and take longer and thus do not affect short range shipping needs. For inter-continental reactors, triggering excludes about two thirds of the reactors. This is simply because for inter-continental reactors, the budgets are larger and deadlines are later, and thus short range vessels are more likely to affect these reactors. The experimental results demonstrate that indexing and abstraction are effective optimizations: (a) the triggering mechanism is effective in avoiding the wakeup of a substantial number of blocked reactors; and (b) the triggering mechanism is itself relatively fast.

5 Concluding Remarks

This paper has presented a new distributed programming framework which allows distributed program agents to react to a CLP program like a shared common store. Agents modify the CLP program through the use of reactors which are guarded on logic conditions with respect to the CLP. The key challenge is then how to efficiently manage these reactors to allow blocking and wake-up. We detailed a triggering framework which incorporates a novel spatial index structure to solve this problem.

Some of the future work includes the development of an automatic verification tool for application-based abstraction used in the triggering, and the classification of various kinds of advanced views as well as abstraction recommendations for these classes. We are also enhancing OCP with a more generalized version of committed choice in which commit can happen anywhere in the choice branch.

References

1. G. Agha and C. J. Callsen. Actorspace: an open distributed programming paradigm. In *ACM PPoPP*, 23–32. 1993.
2. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
3. P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *ACM POPL*, 178–188. 1987.
4. Distances between 325 cities in the world. `http://www.etn.nl/distanc4.htm`.
5. V. Gaede. Multidimensional access methods. *ACM Computing Survey*, 30(2):170–231, 1998.
6. T. Gautier and P. L. Guernic. SIGNAL: A declarative language for synchronous programming of real-time systems. In *FPCA*, pages 257–277. Springer, 1987.
7. D. Gelernter. Generative communication in Linda. In *ACM TOPLAS*, 7(1):80–112, 1985.
8. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *ACM POPL*, 111–119, 1987.
9. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP(R) language and system. In *ACM TOPLAS*, 14(3):339–395, 1992.
10. H. P. Nii. *Blackboard Systems*. Addison Wesley, 1989.
11. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.
12. V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
13. G. Smolka. The Oz programming model. In *Computer Science Today*, pages 324–343. Springer, 1995.
14. K. Ueda. Guarded horn clauses. In *4th Logic Programming '85*, LNCS 221, 168–179, 1986.
15. J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, Inc., 1996.