OPEN CONSTRAINT PROGRAMMING

KENNY QILI ZHU

B.Eng.(Hons), NUS

A THESIS

SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE
2005

To my parents.

Acknowledgments

Back in the summer of 1999, I graduated from Department of Electrical Engineering, National University of Singapore, with a bachelor's degree. An incidental browse of the newly established School of Computing of NUS web page drew my attention to a project lead by the then Associate Professor Joxan Jaffar and Assistant Professor Roland Yap. The project is about building a concurrent programming system and it requires a research assistant. And that was the start of my 6-year endeavor with Joxan and Roland on the Open Constraint Programming.

OCP is an interesting project because we are developing a new idea. It is also very challenging because, firstly the idea was rudimentary so it took a lot of trial-and-error fumbling in the dark; secondly, this project involves the development of a big concurrent programming system which requires knowledge and efforts in many seemingly orthogonal areas, such as operating systems, constraint programming, database systems, indexing; thirdly, in this project, we have suggested a nice technical solution but all the time we have been search for the problem for which the solution applies. In fact, it was not until a few months ago when we finally convinced ourselves that we have found a *killer app* for the OCP framework. I cannot imagine myself completing this project and coming up with this thesis without the smart and patient guidance and advice from my two advisors, Joxan and Roland.

Joxan, despite being the Dean of the School with tons of administrative work to do, makes tremendous effort to meet up with me and Roland on almost daily basis to keep track with my progress and to moot new ideas. His unique insight and acute intuition has guided me throughout this project. In fact, OCP is Joxan's brainchild originally, because he was looking for ways to generalize constraint logic programming (CLP) to work in distributed environment. And since CLP was one of his earlier "babies", Joxan knows CLP very well, and knows exactly what he wants from OCP. Therefore, he has done a great job in focusing my research on the right track and keeping me on that track. His signature humor never fails to brighten our day at the end of lengthy technical discussions. Joxan is more than an advisor to my PhD study, he

is also a great mentor of life and a great friend to me. I could never forget how many times he has sent me home after our late night meetings and how we would chat happily about travel, sports and other interesting topics; I could never forget the way he patiently suggested to me how to respond to questions when I was about to attend important job interviews; I could never forget how much I benefit when I turned to him for advice on career choices, family and even love life! I am deeply grateful to have Joxan not only as my academic mentor but also as a lifetime friend.

If Joxan has provided me with strategic guidance at a higher level in this project, then Roland has helped me with much of the ground work. Roland is one of the most dedicated researchers I have ever met. He has wide area of interests and is constantly aware of new technologies and developments. He also has many great ideas, though some of them cannot be easily accepted by others due to their pioneering nature. Roland is also a very fast learner who's very receptive to new ideas. That makes our discussion often very efficient and productive. Because Roland is so knowledgeable in many areas in computer science, he is very good at connecting the dots, so very often, when he told me to look at certain article in a seemingly unrelated area, I would often realize later that there was some implicit connection with our project which I didn't see earlier. As far as I know, most of the PhD advisors often only advise their students at a very high level. But Roland doesn't mind getting his hand "dirty." He contributed with writing papers when the deadlines approached, and even went as far as helping me debug my programs. He has also relentlessly proofread and corrected errors in this thesis to make it look the way it is today. Throughout those years, we have built up a strong teacher-student relationship as well as a long-lasting friendship. All those happy memories will stay with me for years to come.

My gratitude also goes out to my wonderful colleague and dear friend Andrew E. Santosa, a.k.a. Dido. Dido has worked on the OCP project with us for the first 3.5 years before embarking on his own project and he has been one of my co-authors for a number of papers. He is a very hardworking guy. Together we have solved many detailed problems and enjoyed many nights of chit-chat. Even after Dido has retired from this project, I have still received occasional help from him which was always very valuable to my research. Thank you so much, Dido, for everything!

My love and thanks also goes to my dear parents. I probably can't pull through this six years of hard-work in good shape without the excellent care by my parents. Whenever I'm sick, baffled or frustrated, my parents are always there to comfort me, care for me and give me

courage. That is why this thesis is specially dedicated to them.

And last but not least, I would like to thank a number of friends and colleagues who have given me advises and suggestions along those years and in the making of this thesis. They are: Yuanlin Zhang, Hui Wu, Razvan Voicu, Shaofa Yang, Professor Seif Haridi, Professor H. V. Jagadish, among many other wonderful people. Without you, this thesis could never be in this shape!

June 30, 2005

In National University of Singapore.

Table of Contents

Summary

Open Constraint Programming (OCP) is a new programming framework that offers shared-store distributed programming for reactive applications. The basic feature of this paradigm is that it allows multiple agent programs to react and synchronize with a highly structured shared store through the use of embedded program fragments called *reactors*. Such shared store takes the form of a knowledge base which can be realized by a constraint logic program. The use of a knowledge base makes the framework very flexible because a knowledge base covers a rich spectrum from a repository of raw data items to a complex system of constraints and logic rules. In this sense, OCP generalizes the Linda coordination model. The paradigm gives rise to reactivity with a set of stylized reactor language constructs including a unique *sustain* construct and non-deterministic committed choice construct. The advanced feature of the paradigm is the generalization of committed choice (which also exists in most concurrent languages) to allow commits to happen arbitrarily "late" in the choices, and thus enables the agents to "speculate" different possibility in future. This powerful feature opens new areas of applications for which traditional early committed choice or long transactions cannot handle well. The OCP framework is particularly useful in the areas of e-business, multi-agent systems, workflow systems, gaming and robotics, etc. In summary, this thesis presents:

1. the theory of OCP which includes a basic model and a speculative model;

2. some key implementation concepts including the triggering framework that uses a novel RC-tree spatial index structure and the optimization of the multi-world data structure in the speculative model;

3. the first OCP prototype system based on the basic model and some applications developed using the prototype system.

List of Tables

List of Figures

List of Abbreviations

cLP Concurrent Logic Program(ming)

CCP Concurrent Constraint Program(ming)

CC-NUMA Cache-Coherent Nonuniform Memory Access

CLP Constraint Logic Program(ming)

CP Common Property

CV Conjunctive View

DB Data Base

DFV Differential View

DV Disjunctive View

ECA Event-Condition-Action

ECC Early Committed Choice

GCC Generalized Committed Choice

GHz Giga Hertz

GIS Geographical Information System

LAN Local Area Network

MAS Multiagent System

MBR Minimum Bounding Rectangle

MCV Materialized Conjunctive View

OCP Open Constraint Programming

PDA Personal Digital Assistant

QT Quadtree

RPC Remote Procedure Call

RTDVRP Real Time Dynamic Vehicle Routing Problem

RVR Reactive Vehicle Routing

RVRS Reactive Vehicle Routing System

SC Space Control

SMP Symmetric Multiprocessor

TS Tuple Space

VRP Vehicle Routing Problem

VRPTW Vehicle Routing Problem with Time Windows

w.r.t. with respect to

Part I

Background

Chapter 1

Introduction

This chapter provides the general background and an overview of the content of this thesis. We start with the motivation of the Open Constraint Programming (OCP) framework, which is distributed and concurrent computing where many agents or processes are involved, and suggest OCP as a solution to the problem of coordinating many agents. Then we outline the basic challenges faced by the proposed OCP framework and briefly describe how we tackle these challenges. Finally, we conclude the chapter with the organization of the thesis.

1.1 Coordinating Many Agents

Prior to the advent of the World Wide Web (WWW) in 1980's, computers were largely operated independently and in detached mode. We call computing with a single CPU computer *monolithic computing*. In fact, Thomas Watson, Chairman of IBM, remarked in 1943, "I think there is a world market for maybe five computers", and of course, all those computers were presumably disconnected.

Since the inception of the Internet in the late 80's, more and more mainframe computers, personal computers, and other smaller computing devices are connected to the Internet. More recently, mobile computing has joined the big network, and many mobile devices such as cell phones and PDAs are now also hooked up to the net. In addition, companies and organizations have also increasingly maintained their local area networks (LAN) which consist of sizeable number of interconnected computers. These local networks are referred to as the *intranets*. With so much collective computing power on the Internet and the intranets, there is great amount of interest in developing new applications that take advantage of the interconnection of computing resources, and also designing new computing/programming models and paradigms to support these new applications. These applications can be loosely classified as "distributed

applications". This form of computing which involves multiple computers and other resources connected in a network is called *distributed computing*. A subset of distributed computing involves the use of software "agents" that would interact with each other directly or through a central medium.

An agent is a computational entity as a component of a software or a hardware what is capable of perceiving and acting upon its environment autonomously[Wei99]. Distributed and concurrent agent applications include e-commerce applications such as trading and auctions, *Just-In-Time* (JIT) manufacturing management, supply chain management, distributed simulation, online gaming and so on. In these applications, traders, producers, consumers, vehicles, ships, game players, etc can all act as or deploy agents to interact (either compete or cooperate) with each other.

Many programming models for distributed computing have been proposed and some have been implemented to handle the interaction and coordination of distributed agents, most prominently: message passing model, client/server model, and coordination model.

Message passing has been extensively used in distributed and parallel computing. Some examples of the message passing paradigm include socket programming, the MPI [HPY+93] and the PVM [Sun90] system. More recently, message passing has also evolved into agent communication languages (ACLs) [KSN00]. The message passing paradigm is very efficient because programmers has very low level control over almost every aspect of the agent communications. However, programmability suffers since it is the programmer's burden to ensure every detail of the data distribution and that the task allocation is correct. Hence using message passing in big applications tends to be tedious and error-prone.

The client/server model is the most common distributed computing paradigm today. Typically, a server offers a service that is used by clients across the network. Its main advantage is that it improves the structure of the programs and thus supports collaborative software development. Another advantage is its flexibility, in the sense that different client may implement different functionalities. However, most of the time, client/server model is used to implement service based applications such as databases, file server, printer server and web services, and not so much of a model for coordinating distributed agents. In addition, the programming effort of client/server model can be significant, too.

The coordination model describes the interaction among autonomous agents and deals with the communication, synchronization and task management. It separates the computation per-

formed in a single process from the cooperation (competition) activities performed with a number of processes. Like the client/server model, coordination model benefits structuring programs, because of the separation of computation and coordination. The separation also improves the heterogeneity of the system since the coordination part of the system has a "black-box" view of the processes or agents and does not assume any particular language or implementation of those processes or agents. As such, this model is very suitable for implementing *open systems*. An open system, as being considered in this thesis, is one whose structure is capable of dynamic changes. The components of an open system is not known *a priori*, and can change over time. Designing of open systems presents a great deal of challenge because of their heterogeneity and dynamism. The most prominent example of an open system is the omnipresent Internet, which incidentally is not designed on purpose.

One of the simplest coordination models is Linda [Gel85, CG86a], proposed by Gelernter et al. in 1985. Linda used a collection of tuples which is known as a *tuple space* as a medium of communication and synchronization point of the distributed agents. Agents synchronize with each other by blocking wait on a request of a tuple, and being awakened when the requested tuple becomes available. Linda model is known for its simplicity and relatively high level programming. A number of variants have been implemented, usually combined with other languages such as C, Fortran, and Prolog. However, because the only synchronization condition is the existence of certain type of tuples, applications which require more sophisticated synchronization conditions can be hard to program in the Linda model. This has led us to the investigation of a new model for coordinating agents in an open systems, and our solution to solve this problem is the proposed Open Constraint Programming framework. The following example illustrates the kind of applications for which the Linda model is inadequate and demonstrates why the new model is needed.

1.2 A Flight Reservation Example

Suppose Bob wants to go from New York (NY) to San Francisco (SF) while Jill wants to go from New York (NY) to Los Angeles (LA). Figure 1.1 depicts the two possible choices of routes for Bob (and similarly for Jill): one is a direct flight NY-SF or via a midpoint Washington DC (DC) with the 2 flights NY-DC and DC-SF. Currently there are no available seats on any of the routes. But seats will arrive into the system due to return of tickets, etc, at unexpected times

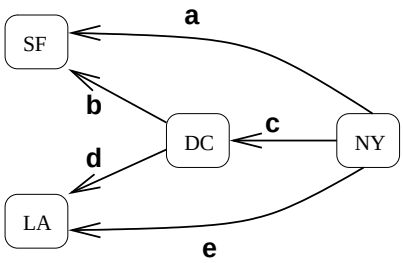


Figure 1.1: A Flight Network

in future.

Both Bob and Jill want to get the tickets that enable them to complete their trips if and when situation that allows them to do so arises. Therefore their bidding for the tickets has to be *reactive* to the availability of the tickets and subject to the maximum prices Bob and Jill can afford and timing constraints. Further, neither may commit to a partial route. If we have an additional constraint that none of them has money to pay for more than one route, Bob and Jill have to *speculate* on which route to buy, rather than actually purchase the tickets. It is also interesting to know that since DC is a common midpoint for Bob and Jill, there may be competition between them on the **c** route.

The basic elements of this example is complex *reactivity* and *speculation*. A solution of the problem here (i.e. the successful purchase of a trip) relies on complex logical conditions on tickets availability, prices, timing. For example, the departure time of segment **b** must be after the arrival time of segment **c**, and the combined price must not exceed certain limit. These conditions cannot be easily expressed in the Linda model or any other existing coordination models.

Because neither Bob nor Jill has enough money to pay for more than one trip, so traditionally they would have to use a "bet-and-risk-it" strategy and wait on just one trip's availability. This is not good because certainly one can make a mistake with the gamble and lose solutions that may have come by. Thus it is highly desirable to let Bob and Jill try both of their choices simultaneously in a speculative computation. To date, none of the distributed computing models allows such speculative computation.

1.3 Open Constraint Programming

The proposed programming framework for the coordination of distributed agents, Open Constraint Programming, consists of two models: the *basic model* and the *speculative model.*

The basic model can be characterized as a number of program agents interacting with a shared, non-monotonic, common *store.* The common store is implemented by an updateable knowledge base which represents the agents' shared environments for the purpose of cooperation and coordination. There is no direct communication between the agents, and all communications are done via the common knowledge base. The highly structured knowledge base can be realized by a constraint logic program (CLP) [JL87] which contains logic rules, constraints and dynamic facts. Agent programs communicate with the store through the use of embedded program fragments known as the "reactors" that are transported to and executed in the store, similar to the execution of Remote Procedure Calls (RPCs). The reactor language offers a set of concurrent and reactive programming constructs that enable the software agents to synchronize on complex constraint and logical conditions, and to make a form of *committed choices.* This model is called "reactive" because reactors can react to the change of their environment, which is the state of the store. They are blocked when their synchronization condition is not true and they are "awakened" when the condition becomes true later. Note that this is more general than the "guards" provided by most concurrent programming languages. The main technical challenge of the basic model is to provide a trigger framework so that allows the efficient wakeup of the minimum number of reactors which *should* be awakened, given an update to the store. In OCP, we suggest a multi-level view approach as well as *abstraction* to reduce complex blocking conditions to simpler, flat ones, and then index those simpler conditions as multi-dimensional spatial objects in a novel index structure called RC-tree.

The speculative model of OCP enhances the basic model to go beyond committed choices. It generalizes the committed choice to obtain a new constructed called GCC, so that agents can speculate against the future conditions by *actually* executing different alternatives in a generalized committed choice (GCC) construct, and not select one choice to run like in the non-determinism provided by many other concurrent languages or frameworks. Speculative OCP is a very powerful programming model that is useful in application that involves decision making in face of future uncertainties. The key challenges here are the semantics of speculation as embedded in the GCC and the implementation difficulty of dealing with possibly exponential space

as a result of multiple possibilities from many agents. We propose a number of optimizations to help contain this space blow-up.

1.4 Organization of the Thesis

This thesis gives an overview of the OCP theoretical models, important implementation techniques and some experiments with a prototype reactive OCP system, and it is structured into 11 chapters which are divided into five parts.

Part I provides the background information to the thesis which includes this chapter and a survey in chapter 2. The survey reviews some of the well-known shared-memory programming models and systems for concurrent or reactive systems. Some of these systems, like Linda and Orca, can be categorized as DSM; while others like Cilk and Treadmarks are more of a parallel language breed. OCP is related to cLP and CCP because these two frameworks also feature a shared store. Some of the synchronous programming languages are also reviewed as they are popular in developing reactive systems, although the programs react to mostly signals rather than complex conditions in the OCP framework. Active databases are introduced because reactivity and non-determinism exist in these systems and some triggering is required in active DB just like in OCP.

Part II represents the theoretical foundations of the thesis. It consists of two chapters. Chapter 3 presents the basic model. We give the definition of a store and give examples of how such a store can be realized. Chapter 3 also presents the full set of language constructs that can be used in an reactor and a number of examples. Finally the semantics of the basic model and a trigger framework is presented.

Chapter 4 presents the advanced speculative model which replaces the "early" committed choice with GCC. We have introduced the semantics of GCC with the help of a data structure call "multi-world" that represents parallel multiple worlds as a result of the speculation. This speculative model, at its current state, is largely conceptual: we have yet to implement a real speculative system.

Part III contains the main technical contributions toward the efficient implementation of an OCP system. Chapter 5 presents the RC-tree, as an underlying technology for implementing the basic model. The RC-tree is a novel spatial indexing technique for objects with large extent and possibly heavy overlapping. It allows dynamic insertion, deletion of geometrical shapes, and

point and range queries. The chapter contains the algorithms, analysis and the experimental evaluation of the RC-tree structure. Because of the general applicability of this index structure, the chapter can also be read independently from other chapters.

The key technical challenge in the implementation of the basic model is to design and implement a triggering mechanism so that a small subset of blocked reactors can be "awaken" or re-enabled efficiently without checking through all blocked reactors which can be in very large amount. Chapter 6 presents a trigger framework to handle reactor conditions that are CLP goals. We start with indexing basic (level-0) views which are derived from simple reactor conditions, and further discuss the methodology to reduce composite (level-n) views to basic ones by *abstraction*. All basic views can be indexed by the RC-tree.

Chapter 7 discusses key implementation concepts behind a practical speculative OCP system. We focus on methods of optimizing the size of the multi-world as well as the number of concurrently running program instances. The goal is to reduce the space and time complexity of such a system.

Part IV describes the experiment experiences with a prototype OCP system and an extension of the basic model. For this thesis, we have only implemented a system based on the basic model. We do not have a working implementation for the speculative model at present. Chapter 8 presents the architecture and the implementation of the prototype system. This system includes a C++ implementation of a server with a CLP($\mathcal{R}$) system as its store, and a library for the Python front-end. Chapter 9 deal with a number of example applications, namely a real-time stock trading system, a distributed constraint search solver, multi-agent simulation for computerized war games, real-time dynamic vehicle routing and a meeting scheduling system. Chapter 10 attempts to distribute the store in the basic model in a preliminary bid to come up with a distributed OCP system.

Finally in Part V, chapter 11 concludes the entire thesis and discusses some future directions of this research.

Chapter 2

Survey

In this chapter, we will briefly survey a number of programming languages and systems which are related to OCP in one way or another. Because OCP is a coordination-based distributed programming model that resembles shared memory programming, we divide this chapter into three sections. We first present "shared-memory-like" programming languages/systems in section 2.1, and then synchronous languages for reactive systems in section 2.2. Finally we discuss the active database systems which represent some form of reactive programming. For extended readings on concurrent/parallel/distributed programming languages and systems, the reader is referred to [BST89, Sha89, HS96, ST98, Has00, Leo01].

2.1 Shared Memory Programming

Shared memory programming is a task-parallel model characterized by the availability of a shared memory. Such shared memory can exist physically (like SMP), or be implemented logically. The basic challenge of shared memory programming is *synchronization* and *locality*. Synchronization is needed when a task or process must coordinate its action with other tasks. Locality is required since data required in the computation maybe distributed, such as caching, and consistency needs to be maintained. The useful features of shared memory programming has led to the development of *distributed shared memory* (DSM). DSM provides an abstraction of shared memory on physically distributed memory architecture such as CC-NUMA, clusters, grids.

In this section, we compare several popular concurrent/parallel/distributed languages with shared or global memory for the purpose of coordination or computation. The first three languages, namely Linda, JavaSpaces and Orca are *blackboard* programming languages whose communication model very much resemble that of OCP. Here the blackboard is the shared space

or store, and programs act like agents that communicate not to each other, but through the shared data as a medium. All these languages assume logically shared memory but possibly physically distributed data structure. Cilk is a general-purpose parallel programming language based on C, and it does rely on physically shared memory. TreadMarks is a distributed shared memory system. Finally we introduce concurrent logic programming (cLP), and concurrent constraint language (CCP) such as Oz.

2.1.1 Linda

Linda [Gel85, CG86a] is parallel programming model (system) originally developed by David Gelernter and his colleagues at Yale University for the SBN network computer [GB82]. Linda features *generative communication*, which Gelernter argues to be a new model of concurrent programming beyond the existing ones: monitors (shared variables), message passing and remote operations (RPCs). Generative communication forms the basis of parallel computing in the Linda model.

A Linda program is a collection of ordered tuples of two types: active tuples which consist of executable code, and passive tuples that represent data only. And the programming model of Linda is based on a pool of tuples called *Tuple Space (TS)*, which serves as a medium of communication and a means of synchronization among concurrent processes. If process A wants to send data to process B, A generates tuples and put them in the TS; B withdraws them by pattern matching. Synchronization is possible because operations of withdrawal is blocking. This mode of communication is named "generative" because until it is explicitly withdrawn from the TS, a tuple generated by A has an independent existence in the TS. That is, it is equally accessible to all processes and bound to none. The generative communication model is thus called a blackboard model.

There are only three basic language primitives in Linda: `out()`, `in()` and `read()`. `out()` adds a tuple to the TS, `in()` withdraws one from the TS, and `read()` reads one tuple without withdrawing it. For example, remote procedure calls can be implemented as follows:

remote procedure call:

out(P, me, *out-actuals*);

in(me, *in-formals*)

remote procedure:

in(P, who:name, *in-formals*) $\Rightarrow$

{*body of procedure P;*

out*(who, out-actuals)*

}

The original goal of Linda is to achieve high speedups in the parallel setting for real life problems. Computational intensive applications such as DNA sequencing, database search, VLSI simulation and Traveling Salesman Problem have been written in Linda. Linda is a very abstract system with a very high level interface. This makes coordination between distributed processes very straightforward, and unlike most other systems does not require all programs to be compiled together nor does it make any requirements about the languages used by the client programs. More recently, it has been used extensively as a coordination language in the multi-agent systems and other distributed settings.

2.1.2 JavaSpaces

Sun Microsystems' JavaSpaces [Mic98] is derived almost entirely from Linda. The purpose of the JavaSpaces is to provide a simple service for cooperative computing in the Java environment. Like Linda, JavaSpaces have no message passing directly between processes, and there is no invocation of methods on remote objects. Instead, it provides a shared, network-accessible repository for objects which is used by the processes as a persistent object storage and exchange mechanism. It also provides loosely coupled communication among processes through its simple primitives. Almost every concept in Linda has a corresponding name in the JavaSpaces. For example, the Tuple Space in Linda is called JavaSpaces; tuple becomes *entry*. A JavaSpace is the global name space (a shared memory space) which can be shared between computers. Once a JavaSpace is created, objects can be inserted into and retrieved from the space. Coordination of objects (entries) in the JavaSpaces is done by JavaSpace servers. All objects in the JavaSpace have self describing tags. When objects are inserted into the JavaSpace, a JavaSpace servers matches the tags with CPU's looking for work.

JavaSpaces have four simple operations (primitives):

- **write:** put an entry into the space
- **read:** return a matching entry from a space
- **take:** remove a matching entry from a space

- **notify:** send event if matching entry is written

The JavaSpace model can be considered a circulation model, or a "flow of objects" [Mic98]. Objects circulate through the space and while they circulate, other objects examine their tags, and use or execute them. This "flow of objects" model breaks the client/server model. An object can circulate through the JavaSpace after its computation is done, at which time it represents a result. That result is not simply returned to the client. Any objects in the JavaSpace interested in the result may use it. Distribution of objects is done through serialization. All objects which can be inserted into a JavaSpace are instances of a serializable object class called `jive.JavaSpace.Entry`. Because these objects are serializable, they can be persistent: they can be transmitted across the network and they retain their behavior and state on other machines. Below is an example code in JavaSpaces.

```
// "Message" Entry
public class Message implements Entry{
  public String content;
  public Message(){
  }
}

// Create a "msg" entry
Message msg = new Message();
msg.content = "Hello World";

//Store it in space
JavaSpace space = getJavaSpaceServiceViaLookup();
space.write(msg, null, Lease.FOREVER);
```

A drawback to the JavaSpaces approach is that it may not provide a familiar computing environment. In other words, users are forced to adopt the JavaSpace computing model which is a master/worker model. A master spins off a number of workers, which go into the JavaSpace, perform their work, and the Master checks back for results.

2.1.3 Orca

Bal et. al. developed another shared memory parallel language Orca [BKT92], which is intended for application programming rather than system programming for distributed systems. Therefore the language is designed to be simple, expressive and efficient with a clean semantics. Processes in Orca can communicate with each other through shared data. Here the shared data are different than the physically or distributed shared memory, but mainly *logically shared.*

Therefore access of the shared data is through high level operations (like methods in object-oriented programs). The data sharing is achieved through a broadcasting protocol [BK93a]. Another important distinction of Orca with other distributed or parallel programming language is that Orca is a complete language rather than just a few programming constructs built on top of existing sequential languages. Orca is a procedural, strongly typed language with data structure such as records, unions, dynamic arrays, sets, bags and graphs. Its syntax is simple and similar to conventional languages and pseudo-codes. Abstract data types can be created by an object-oriented style specification of the object type followed by implementation of the object methods (known as *operations* in Orca). The high level features and expressive power allow programmers outside computer science discipline to program their applications with ease. For example, pointers are deliberately omitted for safety's sake. For syntax of Orca and a complete example of parallel implementation of Traveling Salesman Problem, please refer to [BKT92] as well as [Bal91].

Orca has been used for the implementation of several applications including parallel branch-and-bound, computer chess and graph algorithms.

2.1.4 TreadMarks

Andrew Tannenbaum has classified both Linda and Orca as object-based DSM [Tan95]. Other two categories of DSM are variable-based and page-based DSMs. In this section, we introduce a page-based DSM, TreadMarks. TreadMarks [KCDZ94, ACD+96] was developed at Rice University, and it supports parallel computing on a network of computers. TreadMarks has a few novel features such as release consistency semantics and multiple-writer protocols.

In TreadMarks, every processor has a view of a shared address space. In the implementation, the shared memory in the TreadMarks system is a linear array of bytes. The memory model employed is a relaxed one called *release consistency*, which means updates to the shared data are not immediately visible but until some synchronization access has occurred. For example, process p's update on a shared variable x only becomes available to process q until q acquires the lock. TreadMarks uses *multiple-writer protocol* from the Munin system [CBZ91] to allow multiple writes to local data before merging them at a synchronization point, and thus reduces the effect of *false sharing*.

TreadMarks provides a simple yet powerful C language application programming interface (API). Shared memory allocation is done through `Tmk_malloc()`. There are three synchroniza-

tion primitives: `Tmk_barrier()`, `Tmk_lock_acquire` and `Tmk_lock_release`. When the first primitive is called, the calling process is stalled until all processes reach that same barrier. The last two primitives acquire and release a lock respectively.

2.1.5 Cilk

Cilk (pronounced "*silk*") [BJL+95, BL94] from Leiserson's group at MIT is C-based runtime system for multi-threaded computation on shared memory parallel computers. Cilk offers a few programming constructs, most notably, `spawn` and `sync`, to express dynamic, highly asynchronous parallelism which can be difficult to write in data-parallel or message passing environments. Cilk employs a provably efficient scheduler based on the *work-stealing* algorithm.

Below is a sample Cilk implementation of the calculation of the Fibonacci numbers.

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}
```

Cilk is a faithful extension of C, in the sense that if we remove the Cilk-specific keywords `cilk`, `spawn` and `sync` from the program, we get a completely valid C code.

The Cilk model of multi-threaded computation can be represented by an directed acyclic graph (DAG). Every Cilk program consists of a number of *procedures*, each of which is made up of a sequence of *threads*. Each thread is a non-blocking C function, which forms a vertex in the DAG. When a thread *spawns* a child thread, a new procedure is created. At the same time, a *successor* thread is also spawned to receive the results after the child returns. In the DAG, there are two kinds of edges: one is *continuation* that connects sequential successors; the other is *data dependency* between threads. The runtime of any Cilk program is determined by the amount of *work* T_1 and the *critical path* T_∞ of the DAG. T_1 is the execution time of a program by one processor, and T_∞ is the execution time by infinite number of processors. The main

theoretical result of Cilk is that the execution time of a Cilk program on P processors is

$$T_P \leq T_1/P + T_\infty,$$

and the space usage is

$$S_P \leq S_1 P.$$

A number of successful chess programs such as *Socrates and Cilkchess [DL99] have been written in Cilk. The problem with Cilk is that it relies on a shared-memory architecture. Though attempts have been made to distribute Cilk, the project was not very successful probably due to the fact that the theoretical guarantee of time and space does not hold any more on distributed memory machines.

2.1.6 Concurrent Logic Programming (cLP)

In this section, we introduce concurrent logic programming (cLP) languages as another class of concurrent programming languages. Traditional sequential logic languages and even the parallelizable ones are used to program *closed* or *transformational* systems, which means given an input, transforming it to some result and outputting it. This framework is not suitable for programming *open* or *reactive* systems, in which a constant interaction between continuous streams of input and agents takes place. To address this new challenge, a new programming paradigm, cLP, has been developed with the augmentation of the sequential logic programming with concurrency theories [Dij75]. Examples of cLP languages include Guarded Horn Clauses, or GHC [Ued86], PARLOG [CG86b], and Concurrent Prolog [ST83]. While the implementation details and features differ among these languages, they are unified by the following main concepts behind all concurrent logic languages:

- *Process interpretation* replaces the traditional procedural interpretation. Every goal in the program is treated as a process instead of a procedure. AND-connected goals are run concurrently, which is known as the AND-parallelism.

- *Communication* is achieved through shared variables (which are not the same as a shared memory). Goals/processes communicate with each other through these variables in an elegant way (similar to the blackboard languages presented earlier).

- *Synchronization* is accomplished by a producer/consumer relationship. A process may be blocked until a shared logical variable has been sufficiently instantiated by another process, which is comparable to the Linda approach.

Some of the new features in cLP languages include:

- The concept of *guard* which is a number of goals to be executed before the body of the goal is attempted. This acts as a synchronization mechanism.
- The *don't know* non-determinism in sequential logic programs are replaced with the *don't care* non-determinism (also known as *indeterminism*). In sequential logic programs, at every resolution step, all alternatives are attempted because we don't know which will lead to success; in concurrent logic programs, only one choice is picked as we don't care if it will lead to success. In other words, in don't care non-determinism, failure is a likelihood and partial solutions or results are possible. Note, however, that it is still possible to program don't know non-determinism in cLP.

For a comprehensive survey on concurrent logic programming, the reader is referred to [Sha89] for a detailed presentation of this language paradigm and comparison of the most languages in this area.

2.1.7 Concurrent Constraint Programming (CCP)

Inspired by the success of cLP, Saraswat [Sar93] developed the general concurrent constraint programming (CCP) framework. In this framework, agents communicate by interacting with a set of shared variables on which they can either post ("tell") or test ("ask") for the presence of some constraint. CCP can be viewed as unifying framework of constraint logic programming (CLP) [JL87] and concurrent logic programming because CCP programs without "asks" (and with don't know non-determinism) can be treated as CLPs; while CCP programs with constraints restricted to term equations are essentially cLPs. The strength of constraint programming such as in CCP and also CLP is that the constraint store does not explicitly represent data, but rather a "closure" of the information at present.

Examples of languages in the CCP category are AKL [HJ90] and Oz [Smo95, RBD+03]. The Andorra Kernel Language (AKL) by Haridi, et. al. is almost a super set of Prolog and GHC, except AKL does not provide Prolog cuts. Most of the propagation techniques

of AKL are inherited from the Andorra model [BG89], which essentially leads to computing the deterministic steps first. Another technique unique to AKL is *local execution*, where two AND-connected subgoals can be evaluated independently until one of them fails in some way.

The Oz project took off in 1991, and is based on the ideas of AKL. Oz is a multi-paradigm language that supports logic, functional, constraint and object-oriented programming, just to name a few. It is a lexically scoped language with first class procedures, state and encapsulated search. It is designed for applications that require complex symbolic computations. The *computation space* in Oz hosts a number of *tasks* connected to an Oz *store*. An Oz store comprises a thread store, a monotonic constraint store, a mutable store which consists of references to the constraint store, and a trigger store. The mutable store is non-monotonic as it contains references to the constraint store variables which may change, and the inclusion of such references, also known as *cells*, gives rise to object oriented programming in Oz. The threads in the thread store can only refer to shared variables in the constraint store and not elsewhere. A thread is runnable if arguments needed in the next step of its current statement are bound. Since the constraint store is monotonic, a thread that is runnable will remain runnable until it proceeds one step in its statement. Computation advances by *task reductions*. More recently, there has been various efforts to enable Oz for distributed computing in the current Oz 3 language [HRS97, AR00, RHB+97]. The distribution is transparent to the programmer who sees no change to the semantics when the program is distributed. In this implementation, the top level space is distributed over multiple processes while the child computation space is not due to the communication overhead.

2.1.8 Summary

Languages	shared data	DSM	update mechanism	nondeterminism
Linda	tuple	yes	unrestricted	no
JavaSpaces	entry	yes	unrestricted	no
Orca	object	yes	unrestricted	guards
Cilk	thread	no	unrestricted	no
TreadMarks	page	yes	unrestricted	no
cLP	variable	no	monotonic	don't care (ECC)
CCP	constraint	no	monotonic [1]	don't care (ECC)
OCP	knowledge base	no	non-monotonic	don't care/know (GCC)

Table 2.1: An Overall Comparison of Shared-memory Languages

[1] CCP updates are monotonic except for Oz in which updates to its mutable cell store is non-monotonic.

Table 2.1 compares and contrasts the shared memory languages or paradigms surveyed in this section with OCP by a number of characteristics. The second column records the different types of data being shared by these languages. The third column indicates whether the system is distributed shared memory. Though OCP, at its current state, is not distributed, we will present some preliminary thoughts about distributed OCP in chapter 10. The column under "update mechanism" summarizes how the shared data are updated in a given language. The first five languages uses relatively primitive, unstructured data, and the operations defined on them are rather unrestricted, in the sense that anything can be done to those data. cLP and CCP allow only monotonic updates to their shared constraint store, in the sense that the meaning of the store becomes increasingly *refined*. Finally, the OCP store allows non-monotonic updates to its highly structure data. The last column gives the kind of non-determinism present in a language. OCP offers don't-care non-determinism like cLP and CCP, but but cLP and CCP both use early committed choice (ECC) for non-determinism, whereas OCP's generalized committed choice (GCC) allows a spectrum of early to late commits and hence more powerful than the other two. In this sense, GCC offers both don't-care and don't-know non-determinism. More details on GCC will be presented in chapter 4.

2.2 Synchronous Languages

In this section, we introduce synchronous languages, another class of languages designed for programming reactive systems. Difference between synchronous languages and the concurrent languages in the last section is, as its name implies, the systems written with synchronous languages have synchronous behavior. It is assumed that each reaction is *instantaneous*. That amounts to saying all "overhead" such as process handling, instruction scheduling and inter-process communication take *no time at all*. In consequence, synchronous reactive systems are *deterministic*, in the sense that identical input into the system always yields identical output, despite the fact that computation can be done concurrently in different processes. In what follows, we briefly compare three synchronous languages, Esterel, Lustre and Statecharts. These languages essentially have the same power, and are only different by programming style.

However that is an object-oriented feature.

2.2.1 Esterel

Esterel [BG92] adopts a classical imperative programming style. It handles assignable variables that are local to concurrent statements and cannot be shared, or *signals* which carries a status of presence or absence and can be shared among concurrent processes. Signals are used for communication among processes or with the environment. There are two types of Esterel statements:

- Standard statements like assignment, signal emission, sequencing, conditional, loops, and explicit concurrency.
- Temporal statements like triggers (`wait` *event* `do ...`), watchdogs (`do ... watching` *event*), or temporal loops (`loop ... each` *event*).

One of the most classic examples in Esterel is a digital wristwatch. Below is a small piece of the wristwatch code that checks the current time and sends a StartBeeping signal if the current time is equal to preset alarm time and the watch is being set.

```
every Time do
   present WatchBeingSet else
      if Equal(?Time, ?AlarmTime) then
         emit StartBeeping
      end if
   end present
end every
```

2.2.2 Lustre

Lustre [CPHP87] distinguishes itself from Esterel as a declarative dataflow synchronous language. Another language in the same group is SIGNAL [GG87]. The main challenge in these languages are to provide synchronous behavior or the notion of time in dataflow model which traditionally is considered as asynchronous only. In Lustre, every variable and expression denotes a *flow*, e.g. a pair made up of

- a possibly infinite sequence of values of a given type
- a *clock* which represent a sequence of times.

A flow takes n^{th} value at the n^{th} time. There is a concept of *basic clock time* which defines the minimum cycle within which a program cannot discriminate external events. Therefore this clock time may not be physical real time. Other "slower" clocks can be defined based on the basic clock time. Because Lustre has a declarative nature, it can also be used as a specification language and lends itself easily to program verification.

As an example, consider the second order filter:

$$H(z) = \frac{az^2 + bz + c}{z^2 + dz + e}$$

The output $y = H(z)x$ can be written as:

$$y = ax + (bx - dy)z^{-1} + (cx - ey)z^{-2}$$

and the corresponding Lustre program is:

```
const a,b,c,d,e: real.


node SECOND_ORDER(x: real) returns (y: real);
var u,v: real;
let
   y = a*x + (0.->pre(u));
   u = b*x - d*y + (0.->pre(v));
   v = c*x - e*y;
tel.
```

Here, `->` denotes "followed by", and `pre(E)` acts as a memory of all previous values of expression E.

2.2.3 Statecharts

David Harel's Statechart [Har87] is an extension to the classical flat state diagram to address the challenge of a visual specification for describing concurrency and reactivity. The statechart augments the conventional state diagrams by stratified multi-levels AND/OR decompositions. The OR decomposition represents the *clustering* and *refinement* of statements; the AND decomposition gives the notion of *independence* or *concurrency* between states. Broadcast mechanisms

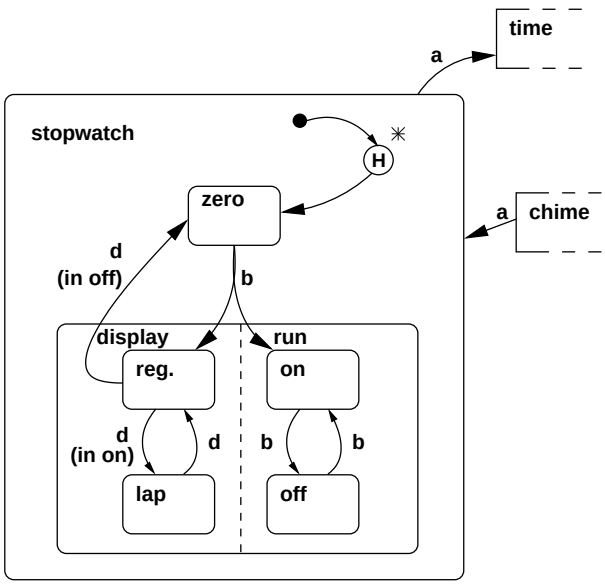


Figure 2.1: Stopwatch Display State in Statechart

are used among the concurrent states. In a pinch, one can say:

$$\text{statecharts} = \text{state-diagrams} + \text{depth} + \text{orthogonality} + \text{broadcast}$$

The running example used in [Har87] is again a digital wristwatch. Figure 2.1 depicts the display state of Harel's Citizen Multi-Alarm III watch (in 1987) in a statechart. Note there is a 2-level hierarchy in this statechart, and the dashed line defines independence between the display and the run state.

2.3 Active Databases

Work in active database area also deals with reactivity [WC96, VB98]. Traditional database systems are *passive* in the sense that data is created, retrieved, modified and deleted only in response to operations issued by users or applications. An active database performs certain operations by itself automatically in response to certain conditions in the database. This makes active databases more powerful because certain frequently used functions can be coded efficiently in the database systems instead of being coded individually in the applications, and such active databases facilitate special applications not possible with conventional databases.

The central notion of an active database is *rules*. These rules are specified by the database administrators, users or applications. An active database rule consists of three parts:

Event: causes the rule to be triggered [DHL90]

Condition: is checked when the rule is triggered

Action: is performed when the rule is triggered and the condition is true

Such rules are also called Event-Condition-Action rules or ECA rules. Other than ECA rules, some active database systems utilize more declarative data-driven rules which are similar to production rules. There have also been efforts of integrating ECA rules to deductive databases [Har93, LLM98].

While the ECA rules provide some degree of reactivity to the users, non-determinism is only achieved with early committed choice of which rule to fire first. In addition, the actions carried out in ECA rules are either simple database read/write operations or user-defined procedure calls. Rules can be processed either sequentially or concurrently. In sequential execution, only one rule is executed at a time, and a conflict resolution mechanism can be used to choose which rule to execute first when multiple rules are triggered. In concurrent execution, all triggered rules are executed concurrently, which avoids the use of conflict resolution but introduces the problem of concurrency control. There is usually subtle differences between the behavior of sequential and concurrent rule executions.

Part II

Foundations

Chapter 3

Basic Model

The OCP framework consists of a *basic model* and an enhanced *speculative model*. This chapter presents the basic model, which offers high-level distributed/reactive programming with shared data. The basic model describes a general method for multiple processes or distributed agents to react to and coordinate with each other using a highly structured shared data called a *store* or *knowledge base*. Section 3.1 gives a general introduction of the basic model and compares it with its related work, CCP. Sections 3.2 and 3.3 discuss the two key components of model, namely the shared store and the reactors. The semantics of the basic model is detailed by the calculus in section 3.4.

3.1 Introduction

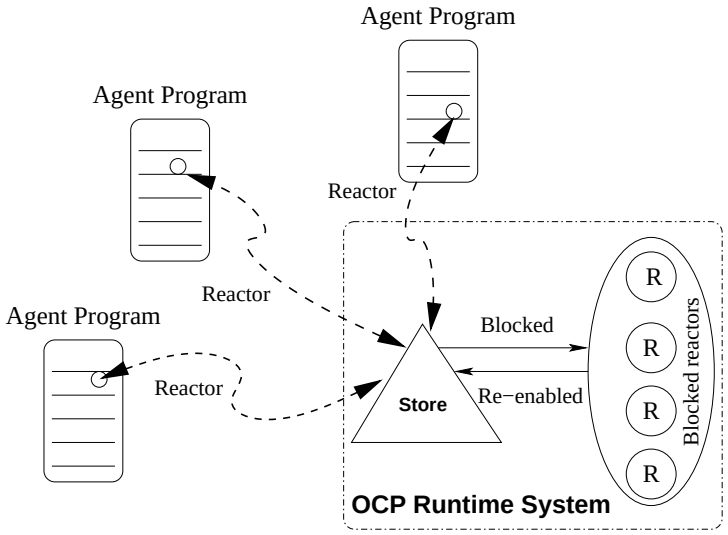


Figure 3.1: The Basic Model

Traditional distributed agent communication models like Actors [Hew77] and ACLs such as

KQML [FFMM94] are based on message passing. While the message passing models give rise to autonomy and distribution, programming the such agents, especially the communication and synchronization of the agents, is very complex. In addition, because there is no central control in these models, different agents may have to be coded with the same functionality independently and repeatedly, causing a great deal of software redundancy. The basic model is an agent-based reactive programming framework which enables the communication and synchronization of multiple agents through the use of a highly structured shared data, and allows constraints and conditions to govern how the shared data is used. The use of the structured shared data as medium of communication improves the centralized reasoning/coordination power and reduces the programming effort at the agent side. The basic model is characterized by the following elements (see figure 3.1):

- a highly structured shared piece of data, known as the *store* and implemented as a *knowledge base*;
- a notion of *reactor*, which is a program fragment written in a number of stylized reactive programming constructs;
- a general use of agents (distributed or concurrent programs) that contain and invoke reactors; and
- a trigger mechanism for *re-enabling* blocked reactors

Concurrent programming against a shared store is not new. We have already reviewed a number of shared memory programming systems and techniques in chapter 2. Depending on the structure of the shared data, on one end of the spectrum, we have page-based shared memory programming such as TreadMarks (section 2.1.4) which shares array of bytes. In the middle, there is blackboard style programming systems such as Linda/JavaSpaces (section 2.1.1 and 2.1.2) which share simple, unstructured data types such as a tuple. And at the other end, more structured shared store is present (in the form of constraints) in cLP (section 2.1.6) and CCP (section 2.1.7) programming paradigms. The basic model is designed to provide a common basis for reactive programming across the entire spectrum, although its strength is best exemplified where the stores are more structured, such as with a constraint logic program (CLP) [JL87].

We postulate that high level reactive programming, such as that in CCP where concurrent agents can share knowledge and synchronize using information in the store is a highly useful

programming paradigm given the popularity of networks and abundance of distributed computing resources. While basic model is closely related to CCP, it is an improvement over CCP and addresses the following limitations of CCP:

1. In CCP, entailment is defined on a monotonic store and often the underlying store represents some logical state. Persistent representation of a state is difficult. OCP moves away from monotonic stores by introducing a the notion of *non-monotonically updateable* store which is analogous to databases, and the state of the store is persistent.

2. The underlying guarded rules with committed-choice operation in CCP are used both for application specific part of the programs and the reactive part. As a result, the granularity of the processes interleaving is often finer than desired. It is, instead, more flexible to use a generic and standard programming language for application development and use some embedded feature for reactivity and concurrency, which is the case in the basic model.

3. CCP defines a closed system where the interaction of processes are pre-determined by the program; OCP is designed for open systems where creation, introduction and termination of agents are all external to the store and thus the interaction of agents is unknown *a priori*.

4. CCP only provides atomic *ask* and *tell* operations. OCP employs more general notions of atomicity and transactions including some of those from the databases.

The store in OCP is formalized as a theory which can be updated. The update functions come from external agent programs. The program fragments which issue updates and synchronize with the store are call *reactors*. Reactors are embedded in user application programs which are coded in any standard programming languages. In the remaining sections, we focus on the store, the reactors syntax and semantics, as well as the triggering mechanism used in OCP.

3.2 Store

Central to the basic model is the notion of a store, representing a shared memory and knowledge as well as medium of communication and synchronization for the reactive agents.

Definition 1 (Store) *A store Δ is a set of formulas in a first order language $\mathcal{L}$ w.r.t. a base theory $\mathcal{T}_0$ and the intended interpretation of $\mathcal{T}_0$. A store offers two kinds of basic operations:*

- *entailment of* Δ *which gives whether* $\Delta \models c$ *is true, where* c *is another formula in* $\mathcal{L}$ *, and*
- *update* δ *which is a transition function that maps one store* Δ *to another store* Δ'.

A store can be a shared memory, a relational database, a set of tuples like in Linda, or a set of rules in CLP. and when this notion is used, the store is also referred to as an *knowledge base*. Depending on δ, the theory can be arbitrarily updated, hence the knowledge base is *non-monotonically updateable*. In the case of a CLP store, the entailment is defined as a predicate c being implied from any of the rules in the CLP program.

A store can also be a system of constraints which can be updated either *monotonically* or *non-monotonically*. For example, using the interpretation of linear integer arithemetic on an appropriate base theory of arithmetic, a constraint store could be:

$$
\begin{aligned}
& 1 \leq X_1 \leq 4 \wedge 1 \leq X_2 \leq 4 \wedge 1 \leq X_3 \leq 4 \wedge 1 \leq X_4 \leq 4 \wedge \\
& X_1 \neq X_2 \wedge X_1 \neq X_3 \wedge X_1 \neq X_4 \wedge X_2 \neq X_3 \wedge X_2 \neq X_4 \wedge X_3 \neq X_4 \wedge \\
& X_1 - X_2 \neq 1 \wedge X_2 - X_1 \neq 1 \wedge \\
& X_1 - X_3 \neq 2 \wedge X_3 - X_1 \neq 2 \wedge \cdots
\end{aligned}
$$

which can be used to represent constraints on a four-queen search problem over the variables $X_1, \ldots, X_4$. An example store update is the conjoining of this formula with $X_1 = 1$, which places the first queen (of the first column) on the first row. A subsequent update conjoining $X_2 = 1$ would lead to an inconsistent store, while conjoining $X_2 = 4$ gives a consistent store.

We will abuse the notation Δ hereinafter to denote both the name of a store as well as the one of its states, because we are usually only interested in the current state of the store as we are referring to it. We write $\Delta \xrightarrow{\delta} \Delta'$ to denote an *update* of the store via δ. We may also write $\mathcal{C} = \Delta \xrightarrow{\delta} \Delta'$, to denote that $\mathcal{C}$ is the *result* of the update δ, also in the language $\mathcal{L}$. δ is said to be *passive* if $\Delta = \Delta'$, e.g. when δ is an `ask` of the store; otherwise δ is said to be *active*.

We now give several more examples of valid stores.

EXAMPLE 1

(One-bit) This trivial store contains just one-bit variables, which are either `true` or `false`. It can be thought of having just two possible constraints $X = 0$ and $X = 1$. Possible updates whose meanings are obvious are:

```
is_zero(X), is_one(X), set_zero(X), set_one(X), flip(X)
```

Here, the first two updates are passive while the last three are not. Other more sophisticated data structures such as sets, arrays, trees can also be modeled in a similar fashion. □

EXAMPLE 2

(Arithmetic) The constraints of interest in this Δ are conjunctions of linear arithmetic atomic formulas $\mathcal{C}$ open on a global decision variable X, such as $-2 \leq X \wedge X < 8$. The meaning of the store is characterized by the polyhedron corresponding to the solution space of the conjunction. Some example updates are:

- `tell`($\mathcal{C}$). E.g. `tell`($X = 10$) fails if $\Delta \wedge X = 10$ is inconsistent, and returns `false`. Otherwise, $\Delta' = \Delta \wedge X = 10$, and returns `true`.
- `softtell`($\mathcal{C}$). E.g. `softtell`($X = 10$) is passive and return false if $\Delta \wedge X = 10$ is inconsistent but does not change Δ. Otherwise it returns true and updates Δ to $\Delta' = \Delta \wedge X = 10$.
- `ask`($\mathcal{C}$). E.g. `ask`($X < 5$) is a passive update. It returns `true` if $\Delta \models X < 5$ or `false` otherwise. [1] There is no delay in return.
- `is_ground`(X). This update is passive and it returns $X = n$ if $\Delta \models X = n$ for some number n, and `false` otherwise.

□

EXAMPLE 3

(Constraint Database) We use a PROLOG program P with ground facts [Llo87] here, though any CLP program, treated as a constraint database [Rev02], could be used by a straightforward extension of the following notions. There are variables of the form X_p which represents a sequence of rules defining the relation p. There are also variables of the form X_G representing allowed goals G to P . The meaning of a store is characterized by the least model of P (see [Llo87]). For example, given the program P:

```
p(X) :- q(X), r(X).
q(a).
q(b).
r(b).
```

[1] If Δ has $X = 3$, then $\Delta \models X < 5$; if Δ is updated to $X = 10$, then $\Delta \not\models X < 5$.

The decision variables of interest are X_p, X_q and X_r, whose values are constant in the store which is also the program itself. The meaning of this store is that a subset of the Herbrand universe has been assigned values variables X_p, X_q and X_r. In particular, $X_p = \{p(b)\}$, $X_q = \{q(a),\ q(b)\}$, and $X_r = \{r(b)\}$.

Let $\mathcal{R}_p$ denote a rule for the relation p, and let $\mathcal{G}$ denote the body of a rule. We can define the following updates: `insert`(X_p, $\mathcal{R}_p$), `delete`(X_p, $\mathcal{R}_p$), and `goal`($\mathcal{G}$). For example, if we apply the update `insert`(X_r, `r(a)`) to the store above, we will obtain the new store:

```
p(X) :- q(X), r(X).
q(a).
q(b).
r(b).
r(a).
```

Accordingly, the meaning of the store becomes: $X_p = \{p(a),\ p(b)\}$, $X_q = \{q(a),\ q(b)\}$, and $X_r = \{r(a),\ r(b)\}$. Note that `p(a)` can now be implied from the store, even though it is not added to the store explicitly.

Now consider the update `goal`($\mathcal{G}$). This update is passive and returns a nonempty list of (ground) substitutions, and fails if there is no such list. For the example program above, `goal(p(X))` returns a sequence of two substitutions $\theta_1 = \{X \mapsto a\}$ and $\theta_2 = \{X \mapsto b\}$. This syntax is to be regarded as a shorthand for the formula $\mathcal{C} = p(X)\theta_1 \wedge p(X)\theta_2$.

□

EXAMPLE 4

(Shipping Marketplace) 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 store 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).
map(singapore, quebec, 14633).
map(hongkong, washington, 13117).
vessel('star', hongkong, shanghai, 20000, 0.012, 15, 18).
vessel('lion', singapore, beijing, 10000, 0.01, 10, 18).
```

The `map` predicate records the distance between two cities, e.g. the distance between Seoul and Shanghai is 4668 km. The first `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 a 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 when they are committed to a particular vessel.

The relation below 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 chapter 6. When the `deliverable` condition is satisfied, the cargo can be shipped and the `do_ship` action will update the corresponding capacities along the route.

The following rules specify actions to be performed when a client's cargo can be shipped:

```
do_ship(A, B, Weight, [ID]):-
  ub(vessel(ID, A, B, Cap, Price, Dep, Arr),
     vessel(ID, A, C, Cap-Weight, Price, Dep, Arr)).

do_ship(A, B, Weight, [ID|L]):-
  ub(vessel(ID, A, C, Cap, Price, Dep, Arr),
     vessel(ID, A, C, Cap-Weight, Price, Dep, Arr)),
  do_ship(C, B, Weight, L).
```

□

To conclude, in the rest of this thesis, unless otherwise stated, *we assume the store to be a CLP knowledge base and the updates are either insert a predicate, delete a predicate, or update a predicate.* [2] For all practical purposes, a CLP program is sufficiently powerful for the applications we are considering in this thesis. For example, relational data and Linda tuple space can both be represented in a CLP program.

3.3 Reactors

A *reactor* is a program *embedded* in the agent program, invoked by the agent, for the sole purpose of interacting with the store, *at some proper time.* Our model is such that when an agent invokes a reactor, the reactor is submitted to the store, and the agent program blocks and waits until the reactor is executed and terminated (this is similar to the mechanism of remote procedure calls). It is called a reactor, because this program is designed to *react* to events and conditions of its environment, in this case, the store.

A reactor in the OCP framework is written using special OCP reactive language constructs.

[2] This definition allows the update on an arbitrary predicate, though in the rest of this thesis, we are only concerned with updating of the base predicates.

Its operations are defined upon the existence of some conditions in the store. Because the reactors are executed in an asynchronous and detached manner from the agent program, in principle they may never terminate. In practice, one can provide timeout in the reactors to ensure a proper termination. In what follows, we will present the syntax of OCP reactive language, and give an example some examples of how reactors can be written.

We give the inductive definition of a reactor r as follows. Parentheses can be used for normal scoping.

r	::=	
	δ	atomic update
	$r_1; r_2$	sequence
|	r_1 & r_2	interleaving
|	$r_1 \mid\mid r_2$	early-committed choice
|	`commit`	commit this choice
|	$c \Rightarrow r$	sustain
|	$\langle \delta_1; \delta_2; \ldots; \delta_n \rangle$	atomic transaction
|	`repeat` r `until` c	uninterruptible loop
|	`repeat` r `watching` c	interruptible loop

Figure 3.2: OCP Reactor Constructs

Below we give an informal explanation of the constructs to facilitate the examples and delay the formal semantics till section 3.4.

A δ is an atomic update action to the store Δ (defined in Definition 1) to be executed (eventually).

The *sequence* only assumes r_1 is executed and finished before r_2, but r_1 and r_2 can be arbitrarily interleaved with other reactors.

The *interleaving* construct r_1 & r_2 corresponds to the concurrent execution of r_1 and r_2.

The *choice* construct provides a form of early-committed non-determinism. The semantics of this *early committed choice* is that both branches execute concurrently until one of them makes an active update to Δ (i.e. $\Delta \xrightarrow{\delta} \Delta'$ and $\Delta \neq \Delta'$) 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`. The committed choice construct is comparable to the guards in some CCP languages such as GHC [Ued86], where the commit in OCP is analogous to the guard symbol in CCP. Without loss of generality, in this thesis, we assume only two possibilities within a choice construct.

We also define the following syntactic equivalences of the choice construct:

Definition 2 (Choice equivalence)

$$
\begin{aligned}
r_1 \mid\mid r_2 &\equiv r_2 \mid\mid r_1 \\
r_1 \mid\mid (r_2 \mid\mid r_3) &\equiv (r_1 \mid\mid r_2) \mid\mid r_3 \\
(r_1 \mid\mid r_2); r_3 &\equiv (r_1; r_3) \mid\mid (r_2; r_3) \\
r_1; (r_2 \mid\mid r_3) &\equiv (r_1; r_2) \mid\mid (r_1; r_3)
\end{aligned}
$$

`commit` is a special kind of δ. It is only used with in the scope of a choice. `commit` expresses the intention to *commit* to this branch of a choice and to remove the other branch as if it did not exist. `commit` refers to the innermost choice which surrounds the these operations. If `commit` is used outside the scope of a choice, they have no effect. Furthermore, only the first use of `commit` has any effect within a choice branch. We require that every choice branch must have a commit operation, the intent is that a choice must eventually be "committed". This can be achieved by adding `commit` to the end of every choice branch.

The *sustain* $c \Rightarrow r$ is the most important construct which gives rise to reactivity in this language. $c \Rightarrow r$ means to execute r as long as c is true before every δ is executed in r. Execution of r suspends if c becomes false and resumes if c is true again. The idea of sustain is that it provides an alternative to a traditional database transaction. Instead of putting a sequence of atomic updates in a transaction, a sustain construct produces a sequence of *sustained atomic updates* which can be interleaved. A sustained atomic update or $c \Rightarrow \delta$ is also known as a *guard*. Since the reactor can execute for a long time and possibly not even terminate or commit, we do not want a series of updates to be in a database-like transaction and to block other reactors from accessing the store while the transaction is taking place. A similar idea also exists in the related work of *transactional memory* [HM93].

The *atomic transaction* or *transaction* in short, groups a sequences of updates including commits into one atomic action against Δ. Δ becomes private to this reactor once the execution starts, and becomes public again when the reactor terminates. Notice the definition of transaction in OCP is *different* than that of a normal database system. Atomic transactions are provided to complement sustain in situations when absolute atomicity is required. Because we have already provided sustain as a "relaxed" transaction model, we do not provide inter-

leaving and serializability in the atomic transaction as normal database transactions do. The OCP transaction construct is only valid for a sequence of updates and not for general reactors, because if we have $\langle r \rangle$ with an arbitrary r, r may be blocked due to a sustain inside and consequently block the entire system.

The *uninterruptible loop* executes r repeatedly until $\Delta \models c$, when the system *finishes* r and terminates the loop. The *interruptible loop* behavior similarly except, when $\Delta \models c$, the system aborts r immediate, leaving possible partial effects. Notice that consecutive runs of r can be interleaved with other reactors.

We now first consider the following simple example:

"As long as δ_1 is true, repeatedly execute δ_2, stopping only when ϵ_1 or ϵ_2 is true. Then execute δ_3 unless ϵ_2 is true."

Figure 3.3 shows how this example can be coded in CCP (a), Esterel (b) and StateMate, a Statechart system (c) [HLN+90]. [3] In the case of StateMate, there are three states: A (start), B (running), and C (halt), and the notion ϵ/δ denotes that the update is enabled upon $\Delta \models \epsilon$ and δ is applied.

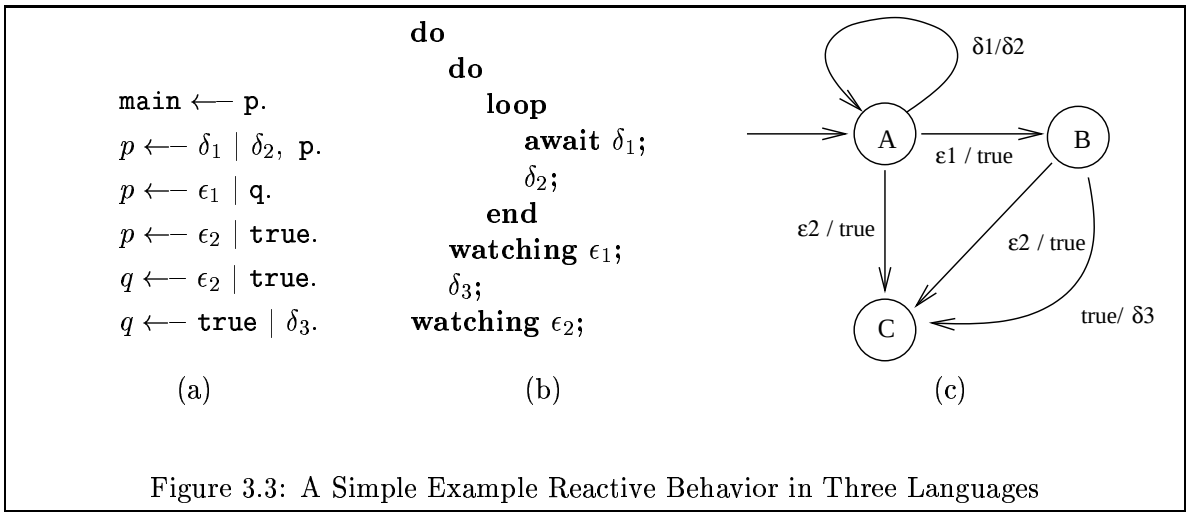


Figure 3.3: A Simple Example Reactive Behavior in Three Languages

If we are to write an OCP reactor to exhibit the same behavior, it will be:

$$\texttt{repeat}(\texttt{repeat}\ \delta_1 \Rightarrow \delta_2\ \texttt{watching}\ \epsilon_1)\ \texttt{watching}\ \epsilon_2);\ (\epsilon_2 \Rightarrow \texttt{commit} \mid \delta_3)$$

We conclude this section with a few more examples. First, let us define a special predicate $\mathcal{T}time(t)$ which holds when t is the current time according to a given wall clock. We assume the

[3] The exact behavior of the three programs are not exactly the same due to detailed considerations such as synchronous, asynchronous execution, concurrency control and timing of the operations.

the availability of functions that extract year, month, day, hour, minute, second, etc, which are represented by $\mathcal{T}year(t)$, $\mathcal{T}month(t)$, etc. The first two examples are time-related applications written as OCP reactors.

EXAMPLE 5

(Alarm clock) An alarm clock is an action r that should be executed when time t is between certain minutes t_1 and t_2 of the day, provided some condition c is true. Typically action r could be just "set off the ringing". Such an alarm clock can be expressed as:

$$\mathcal{T}time(t) \wedge t_1 \leq \mathcal{T}minute(t) \wedge \mathcal{T}minute(t) \leq t_2 \wedge c \Rightarrow r$$

□

EXAMPLE 6

(Timeout) As we have discussed in the last section, to ensure termination, one should specify a timeout for the reactor r. This notion of timeout can be implemented in the language as:

$$\mathcal{T}time(t_0) \Rightarrow (r \mid\mid (\mathcal{T}time(t) \wedge t - t_0 > x \Rightarrow \texttt{commit}))$$

This reactor first records the start time of t_0, and then executes the intended action r concurrent with the timeout choice. The timeout branch commits when the time elapsed is more than x time units, and effectively kills the other branch in which r is being executed.

A more general approach would be to provide an alternative action r', when timeout happens:

$$\mathcal{T}time(t_0) \Rightarrow (r \mid\mid (\mathcal{T}time(t) \wedge t - t_0 > x \Rightarrow r'))$$

□

EXAMPLE 7

(Dijkstra's guards) Dijkstra's guards [Dij76] can be expressed as:

$$G_1 \rightarrow S_1 \ [] \ G_2 \rightarrow S_2 \ [] \ \cdots \ [] \ G_n \rightarrow S_n$$

where G_i is a logical expression known as the *guard*, and S_i is a list of statements.

This can be implemented in OCP by sustained commits in a choice construct:

$$(G_1 \Rightarrow \texttt{commit}; S_1) \;||\; (G_2 \Rightarrow \texttt{commit}; S_2) \;||\; \cdots \;||\; (G_n \Rightarrow \texttt{commit}; S_n)$$

□

EXAMPLE 8

(Recursion) Let $edge(x,\ y)$ be true if there is an edge between node x and node y. The following reactor creates a relation $path(x,\ y)$, if there is a path from node x to node y, given a database of edge relations as the store.

```
repeat
    edge(x, y) ∧ ¬path(x, y) ⇒ +path(x, y)
until ∀x, y. edge(x, y) ∧ path(x, y);
repeat
    path(x, z) ∧ path(z, y) ∧ ¬path(x, y) ⇒ +path(x, y)
until ∀x, y. edge(x, y) ∧ path(x, y)
```

Note $+path(x,\ y)$ denotes the update of inserting a new relation into the database. □

Finally, let us consider an active relational database system which allows certain trigger rules in the ECA form: ON *event* IF *condition* THEN *action*. There are some connections between an OCP system and an active relational database system.

EXAMPLE 9

(Active Database) Let c be a Boolean expression and δ be a simple deletion or insertion on a table. Then a reactor such as $c \Rightarrow \delta$ can be implemented in an active relational database system as a series of trigger rules like this: For each table mentioned in c and for each possible kind of update event e on this table, add the rule ON e IF c THEN δ to the active relational database system. Hence a simple reactor can explode into a large number of trigger rules. □

In commercial relational databases, some restrictions are imposed on ECA trigger rules. The action in an ECA trigger rule is a combination of simple deletions from and insertions into relational tables; the condition in an ECA trigger rule is a Boolean combination; and recursive

firing of ECA trigger rules is disallowed. With these restrictions, ECA rules cannot be used for general computation purposes, and hence are not as powerful as the OCP framework.

3.4 Semantics

In this section, we present the operational semantics of a basic OCP system in terms of a number of transition rules. The basic system is composed of a store state Δ and a finite set of reactors R and their *states*. For the purpose of the discussion in this section, it suffices to think of Δ as a logical theory representing the current store which supports the notion of logical entailment on the test of formulas c appearing in the sustains. Because reactors are inductively defined, which mean a reactor r can itself be composed of a number of "sub-reactors", e.g. reactor "$r_1; r_2 \parallel r_3$" can be thought of having three sub-reactors. The state of the reactor (or sub-reactor) is essentially the *continuation* of the program, which changes as the program is being executed. We write r' to denote the next state of r after some transition. Notice here we are abusing the notation r to indicate the state of reactor.

The dynamic behavior of the basic system is defined by the system transition relation $R,\ \Delta \longrightarrow \Delta',\ R'$, where Δ' is the new store state and R' is the new set of reactors (states) after the transition. These system transitions are defined, in turn, by the reactor transition relation $r,\ \Delta \longrightarrow \Delta',\ r'$, where r and r' are reactor states before and after the transition. The syntax of the transitions are meant to highlight the changes of Δ to Δ' and r to r'.

3.4.1 Basic semantics

Let us first define the system transition behavior below.

$$\frac{}{R, \Delta \longrightarrow \Delta, R \cup \{r\}} \quad \textbf{*** birth ***}$$

$$\frac{r, \Delta \xrightarrow{\delta} \Delta', r'}{R \cup \{r\},\ \Delta \longrightarrow \Delta,\ R \cup \{r'\}} \quad \textbf{*** progress ***}$$

$$\frac{r, \Delta \xrightarrow{\delta} \Delta'}{R \cup \{r\},\ \Delta \longrightarrow \Delta,\ R} \quad \textbf{*** death ***}$$

The "birth" rule allows the arbitrary birth of a reactor as it is invoked from the embedding agent program, which may interact with the store any time. The "progress" rule captures the changes made to the store by an update δ from a reactor r, as r changes its own state to r'. The "death" rule depicts the completion of a reactor r and that it is deleted from the basic system.

Next, we define the reactor transition relations, $r, \Delta \xrightarrow{\delta} \Delta', r'$ and $r, \Delta \xrightarrow{\delta} \Delta'$. Note that the update δ may or may not change Δ. We refer to the a passive update as ϵ, and an active

update as u. `commit` can be treated as a special kind of active update. Let us write $r \rightsquigarrow r'$ if there are Δ , Δ', and $\xrightarrow{\delta}$ such that $r, \Delta \xrightarrow{\delta} \Delta', r'$. Let $\rightsquigarrow^*$ be the reflexive transitive closure of $\rightsquigarrow$. The reactor transitions are defined with the following rules.

$$\frac{r_1, \Delta \xrightarrow{\delta} \Delta', r_1'}{r_1;\ r_2, \Delta \xrightarrow{\delta} \Delta', r_1';\ r_2} \qquad \frac{r_1, \Delta \xrightarrow{\delta} \Delta'}{r_1;\ r_2, \Delta \xrightarrow{\delta} \Delta', r_2}$$

***** sequence *****

$$\frac{r_1, \Delta \xrightarrow{\delta} \Delta', r_1'}{r_1 \ \&\ r_2, \Delta \xrightarrow{\delta} \Delta', r_1' \ \&\ r_2} \qquad \frac{r_1, \Delta \xrightarrow{\delta} \Delta'}{r_1 \ \&\ r_2, \Delta \xrightarrow{\delta} \Delta', r_2}$$

$$\frac{r_2, \Delta \xrightarrow{\delta} \Delta', r_2'}{r_1 \ \&\ r_2, \Delta \xrightarrow{\delta} \Delta', r_1 \ \&\ r_2'} \qquad \frac{r_2, \Delta \xrightarrow{\delta} \Delta'}{r_1 \ \&\ r_2, \Delta \xrightarrow{\delta} \Delta', r_1}$$

***** interleaving *****

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

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

***** committed choice *****

$$\frac{r \rightsquigarrow^* r' \qquad r', \Delta \xrightarrow{\delta} \Delta'}{(r';\ \texttt{repeat}\ r\ \texttt{until}\ c), \Delta \xrightarrow{\delta} \Delta', \texttt{repeat}\ r\ \texttt{until}\ c} \text{ if } \Delta \not\models c$$

$$\frac{r \rightsquigarrow^* r' \qquad r', \Delta \xrightarrow{\delta} \Delta'}{(r';\ \texttt{repeat}\ r\ \texttt{until}\ c), \Delta \xrightarrow{\delta} \Delta'} \text{ if } \Delta \models c$$

***** uninterruptible loop *****

$$\frac{r \rightsquigarrow^* r' \qquad r', \Delta \xrightarrow{\delta} \Delta', r"}{(r';\ \texttt{repeat}\ r\ \texttt{watching}\ c), \Delta \xrightarrow{\delta} \Delta', (r";\ \texttt{repeat}\ r\ \texttt{watching}\ c)} \text{ if } \Delta \not\models c$$

$$\frac{r \rightsquigarrow^* r' \qquad r', \Delta \xrightarrow{\delta} \Delta', r"}{(r';\ \texttt{repeat}\ r\ \texttt{watching}\ c), \Delta \xrightarrow{\delta} \Delta'} \text{ if } \Delta \models c$$

***** interruptible loop *****

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

***** sustain *****

$$\frac{(\delta_1; \ldots; \delta_n), \Delta \xrightarrow{\delta_1} \Delta'(\delta_2; \ldots; \delta_n) \ \cdots \ \delta_n, \Delta^{(n-1)} \xrightarrow{\delta_n} \Delta^{(n)}}{\langle \delta_1; \ldots; \delta_n \rangle\ , \Delta \xrightarrow{\delta} \Delta^{(n)}} \text{ where } \delta = \delta_1; \delta_2; \ldots; \delta_n$$

***** atomic transaction *****

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

***** perform update *****

Most of the above rules are straightforward and formalizes meaning of the constructs we have informally introduced in section 3.3.

The sustain rules require some discussion. The use of the sustain condition c here is like a scheduling condition on execution. The reactor within a sustain can only make transitions as long as c is entailed by the current Δ. Whenever the sustain condition is false, the reactor is *suspended*. And when the condition becomes true again, the reactor is *re-enabled*. It is in this sense that this notion of sustain only makes minimal requirements on the system. We do not require that all sustained reactors to constantly check the store, but rather, before they execute an update to the store, the condition should be satisfied. Reactors can cooperate in enabling or suspending a sustained reactor by changing the entailment of c. We further state the following properties on the model.

In the "atomic transaction" rule, we lump all updates in the transaction together as if they were one single atomic update δ and apply it to Δ. Notice that if the sequence of updates include a `commit`, δ will be interpreted as an active update u, which will trigger one of the "committed choice" rules involving u, and the commit semantics will apply.

The result of performing update δ on Δ is a new store $\delta(\Delta)$ as indicated in the last rule.

Proposition 1 $\rightsquigarrow^*$ *is a well-founded partial ordering.*

PROOF. (Sketch) It suffices to show that $\rightsquigarrow^*$ is anti-symmetric. We define the size of a reactors to be the number of symbols in that reactor, Then anti-symmetry follows from the fact that $\xrightarrow{\delta}$ strictly decreases the size of reactors. □

Proposition 2

- $c \Rightarrow (r_1;\ r_2) \equiv c \Rightarrow r_1;\ c \Rightarrow r_2$
- $c \Rightarrow (r_1\ \&\ r_2) \equiv (c \Rightarrow r_1)\ \&\ (c \Rightarrow r_2)$
- $c \Rightarrow (r_1\ ||\ r_2) \equiv (c \Rightarrow r_1)\ ||\ (c \Rightarrow r_2)$

PROOF. We will prove only the first item by induction on reactors using the well-founded partial ordering $\rightsquigarrow^*$. The other items can be similarly proven.

Consider the sequence $r, \Delta_1 \xrightarrow{\delta_1} \Delta_1', r_1;\ r_1, \Delta_2 \xrightarrow{\delta_2} \Delta_2', r_2;\ \ldots$; and let r be $c \Rightarrow (r_a; r_b)$ and r' be $(c \Rightarrow r_a);\ (c \Rightarrow r_b)$. We need to show that there are r_1', r_2', ..., such that $r', \Delta_1 \xrightarrow{\delta_1} \Delta_1', r_1'; r_1', \Delta_2 \xrightarrow{\delta_2} \Delta_2', r_2';\ \ldots$. That is, r' gives the same sequence of updates and store states.

The base case is that $c \Rightarrow (r_a;\ r_b), \Delta_1 \xrightarrow{\delta_1} \Delta_1', c \Rightarrow r_b$ is the first step in the sequence. According to the first sustain rule and the second sequence rule, it must be the case that $\Delta_1 \models c$ and $r_a, \Delta_1 \xrightarrow{\delta_1} \Delta_1'$. This gives rise to $c \Rightarrow r_a, \Delta_1 \xrightarrow{\delta_1} \Delta_1'$, according to the second sustain rule. Therefore, due to the first sequence rule, $(c \Rightarrow r_a);\ (c \Rightarrow r_b), \Delta_1 \xrightarrow{\delta_1} \Delta_1', c \Rightarrow r_b$. Thus we have $r_1 = r_1' = (c \Rightarrow r_b)$. Therefore $c \Rightarrow (r_a;\ r_b) \equiv c \Rightarrow r_a;\ c \Rightarrow r_b$ as desired. This proves the base case.

The induction case is $c \Rightarrow (r_a;\ r_b), \Delta_1 \xrightarrow{\delta_1} \Delta_1', c \Rightarrow (r_a';\ r_b)$ is the first step in the sequence. Since $c \Rightarrow (r_a;\ r_b) \rightsquigarrow^* c \Rightarrow (r_a';\ r_b)$, it follows by hypothesis that $c \Rightarrow (r_a';\ r_b) \equiv (c \Rightarrow r_a');\ (c \Rightarrow r_b)$. That is, we have $r_1 = c \Rightarrow (r_a';\ r_b) \equiv (c \Rightarrow r_a'); (c \Rightarrow r_b) = r_1'$. From the induction case above and by the sustain rules and the sequence rules we have $r_a, \Delta_1 \xrightarrow{\delta_1} \Delta_1', r_a'$. This gives $c \Rightarrow r_a, \Delta_1 \xrightarrow{\delta_1} \Delta_1', c \Rightarrow r_a'$, which further gives $(c \Rightarrow r_a); (c \Rightarrow r_b), \Delta_1 \xrightarrow{\delta_1} \Delta_1', (c \Rightarrow r_a'); (c \Rightarrow r_b)$. In other words, $r', \Delta_1 \xrightarrow{\delta_1} \Delta_1', r_1'$. And thus $c \Rightarrow (r_1;\ r_2) \equiv (c \Rightarrow r_1);\ (c \Rightarrow r_2)$ as desired. □

From the above, sustain can be decomposed into a series of *sustained atomic updates* of the form $c \Rightarrow \delta$, where the reactor is blocked until $\Delta \models c$, at which point δ is applied to Δ atomically. We refer to sustained atomic updates simply as *guards*.

Proposition 3 *The sustain conditions are compoundable:*

$$c_1 \Rightarrow (c_2 \Rightarrow r) \equiv (c_1 \wedge c_2) \Rightarrow r$$

PROOF. It follows from the sustain rules. □

Because sustains are such important construct in the OCP framework, suspending and re-enabling the sustained reactors are one of the central issues of the basic model. We conclude this section with the presentation of some additional rules on the trigger semantics for the sustains, which serves as a refinement of the basic semantics presented above. This refinement is designed to be at the minimal level of detail in the semantics which would be required for a practical implementation.

3.4.2 Refined trigger semantics

We now refine the definition of the state of the store Δ to contain two components: first component is the theory representing the current state of the store, denoted as THEORY(Δ); second component is a trigger table, BLOCKED(Δ), which records explicitly all the suspended reactors in the form of $i : c_i \Rightarrow r_i$, where i is a handle that uniquely identify a reactor in the trigger table. The intention of the trigger table is to be implemented with an efficient indexing mechanism (as we will see in chapter 5 and chapter 6) so that given a change in the entailment of the store, reactors whose condition c_i that are affected by the change can be quickly identified and re-enabled for execution. We define a new constant variable $blocked_i$, which is a status variable that replaces a reactor r in the set R, when r is a sustain and is being suspended.

Definition 3 (Trigger process) *A trigger process is defined as: given a set of blocked reactors B, and a new update δ to the store Δ, repeatedly apply rules in the refined trigger model to remove re-enabled reactors from the trigger table and add them to the set of available reactors R.*

The following are the refined rules for sustains. Note that $\xrightarrow{\tau}$ is a no-op transition which does not change the theory of the store, THEORY(Δ).

$$\frac{\begin{array}{c}\textsc{theory}(\Delta) \not\models c \\ [i : c \Rightarrow r] \notin \textsc{blocked}(\Delta) \\ \textsc{theory}(\Delta') = \textsc{theory}(\Delta) \\ \textsc{blocked}(\Delta') = \textsc{blocked}(\Delta) \cup \{[i : c \Rightarrow r]\}\end{array}}{c \Rightarrow r, \Delta \xrightarrow{\tau} \Delta', blocked_i}$$

$$\frac{\begin{array}{c}\textsc{theory}(\Delta) \models c \\ [i : c \Rightarrow r] \in \textsc{blocked}(\Delta) \\ \textsc{theory}(\Delta') = \textsc{theory}(\Delta) \\ \textsc{blocked}(\Delta') = \textsc{blocked}(\Delta) \setminus \{[i : c \Rightarrow r]\}\end{array}}{blocked_i, \Delta \xrightarrow{\tau} \Delta', c \Rightarrow r} \qquad \frac{\begin{array}{c}\textsc{theory}(\Delta) \not\models c \\ [i : c \Rightarrow r] \in \textsc{blocked}(\Delta) \\ \textsc{theory}(\Delta') = \textsc{theory}(\Delta) \\ \textsc{blocked}(\Delta') = \textsc{blocked}(\Delta)\end{array}}{blocked_i, \Delta \xrightarrow{\tau} \Delta', blocked_i}$$

The top rule above specifies how a fresh sustain is blocked when its condition is not satisfied by the store. The bottom-left rule depicts the wake up process of a blocked sustain when its condition is now satisfied by the store. The bottom-right rule applies to blocked sustains which do not get triggered by a condition update. The second and third rules should be attempted repeatedly for all blocked reactors after every update to Δ.

Given a set of reactor R and a state Δ , let $R\Delta$ denote the basic system obtained by replacing each r in R by $r\Delta$, where $r\Delta$ is obtained by repeatedly carrying out the following replacements: if $[i : c_i \Rightarrow r_i] \in \textsc{blocked}(\Delta)$, then replace each occurrence of $blocked_i$ in r by $c_i \Rightarrow r_i$.

We are now going to state the following soundness property which says every transition according to the refined trigger rules for has a corresponding transition in the basic semantics.

Proposition 4 (Soundness of trigger semantics)

If $r, \Delta \xrightarrow{\tau} \Delta', r'$, *then* $r\Delta = r'\Delta'$ *and* $\textsc{theory}(\Delta) = \textsc{theory}(\Delta')$ *in the basic semantics.*

Proof. (Sketch) By structural induction on the rules and the fact that update function $\xrightarrow{\tau}$ does not affect the trigger table. □

3.5 Conclusion

This chapter gives an overview of the basic model, which is also the core model of the proposed open constraint programming framework. The model allows multiple agents or processes to react to a non-monotonic shared store which can be realized by highly structured data such

as a CLP program. We present a simple stylized language for writing reactors which can be embedded in agent programs to interact with the store. We also give the semantics of the reactor language as well as as a refined trigger model which gives rise to reactivity in OCP.

Chapter 4

Speculative Model

This chapter extends (and enhances) the basic model of OCP in the previous chapter with a generalized committed choice construct in place of the early committed choice we have introduced in the reactor syntax earlier in section 3.3. We call the extension a *speculative model*. The generalized committed choice (GCC) allows multiple computations from different alternatives to occur concurrently and later commit to one of them. This provides a very powerful way for programs to *speculate* against different future events. One way of thinking about this is that it generalizes the traditional committed choice in Dijkstra's Guarded Command Language beyond *don't care* non-determinism to also handle *don't know* non-determinism. The contribution of the chapter is the semantics framework for formalizing GCC. The key semantic challenge is how to handle the notion of commit given that GCC evolves a computation from a single state into many possible co-existing states. The key implementation challenge is how to contain the possibly exponential space blow-up, and this will be addressed in chapter 7.

4.1 Introduction

Nondeterminism refers to the fact that a computation may need to choose between two or more choices [Hoa85]. *Don't care* non-determinism, or *committed choice*, is the most commonly used form of nondeterminism in concurrent programming systems, e.g. Occam [Hul87] and Concurrent Prolog [ST83]. It's also the basis of many non-deterministic programming constructs such as guarded commands [Dij76], CSP [Hoa85], and π-calculus [MPW92]. The original form of committed choice is the *guarded command set* introduced by Dijkstra [Dij76]. A guarded command set (or Dijkstra's guard) is written as

$$G_1 \rightarrow S_1 \ [] \ G_2 \rightarrow S_2 \ [] \ \cdots \ [] \ G_n \rightarrow S_n \tag{4.1}$$

where G_i is a logical expression known as the *guard*, and S_i is a list of statements. The meaning of Dijkstra's guard is that one can choose any S_i to execute so long as its guard G_i is true. Otherwise if all guards are false, then it aborts. Thus the choice gives rise to a form of *don't care* non-determinism because the system doesn't care which S_i to execute so long as its guard is satisfied. This is in contrast with the *don't know* non-determinism used in OR-parallel logic programming [GPA+01]. Here, a search space is (non-deterministically) explored to find which choices lead to a solution – possible alternatives have to be tried to avoid missing a solution.

In a concurrent programming setting, a meaningful program contains operations that manipulate its environment allowing it to interact with other running programs. A key characteristic of the Dijkstra's guard (similarly guards in Concurrent Logic/Constraint Programming [Sar93]) is that guard G_i is intended as a test to choose an alternative to commit to before performing any other operations which can modify the environment. We call this kind of choice construct, *early committed choice*, as the commit happens as a test at the start of the choice construct.

In this chapter, we propose a new choice construct which allows the commit to occur at an arbitrary point within the choice. The following syntax of the new choice construct is illustrative, where `cm` denotes a commit:

$$S_1; \texttt{cm}; S_1' \ \oplus \ S_2; \texttt{cm}; S_2' \ \oplus \ \cdots \ \oplus \ S_n; \texttt{cm}; S_n'$$

Dijkstra's guard can then be rewritten in our syntax as

$$G_1; \texttt{cm}; S_1 \ \oplus \ G_2; \texttt{cm}; S_2 \ \oplus \ \cdots \ \oplus \ G_n; \texttt{cm}; S_n$$

We call our construct, *generalized committed choice* (GCC), since it generalizes the idea of early committed choice. We assume that all processes are operating in a shared environment in which all variables are global and shared. We call the this environment a *store*. The processes don't interact with each other directly, but instead communicate through modifying the values of the variables in the store. The only operations allowed are global variable assignments. The processes do not have any local variables.

Next we will give a simple example that motivates the development of GCC, and we will present an informal description of the GCC programming model, along with some related work. The remainder of the chapter is organized as follows. Section 4.2 presents a small programming

language which we embed GCC. Section 4.3 introduces the basic runtime structure for GCC. The operational semantics of our simple GCC language is given in section 4.4. We discuss various possibilities for the meaning of commit in section 4.5. We conclude the chapter in section 4.6.

4.1.1 A motivating example

The following example motivates the kinds of non-deterministic choice which are ideal applications for GCC. Imagine two people, among others, participate in an online automated second hand product trading system. In this example, they are trading in camera equipment.

Bob is a photographer who wants to upgrade his equipment. He has two choices: either sell his old camera and buy a better one; or, keep the old camera but sell his old lens and buy a better lens. Bob requires exactly one of the above scenarios to occur, to avoid ending up with two cameras or selling all his equipment but being unable to buy the upgrade. In order to maximize buying and selling opportunities, we assume that the buying and selling of items can happen in any order.

There is another person, Jill, who is broke and wants to downgrade her equipment. Jill would like to either sell her good camera or her good lens. And then using the proceeds from one of the above sales, she can now buy an average camera.

We can now program the intentions of Bob and Jill in pseudo-code as follows. Exclusive choice which is the intent of both Bob and Jill is written as `XOR`. We assume that the market has a clearing function, so that buying an item is only effective if there is a matching sell, and vice versa. Thus both the buy and sell operations are synchronized and will block if the corresponding action is not present.

```
Bob:
  (buy(goodlens); sell(averagelens))
                  XOR
  (buy(goodcam); sell(averagecam))

Jill:
  (sell(goodlens); buy(averagecam))
                  XOR
  (sell(goodcam); buy(averagecam))
```

It is easy to see that there is a perfect match between Bob and Jill as Bob can buy Jill's good camera and then sell Jill his average camera. Thus there is a way in which both parties

can be satisfied. However, since each party is not directly aware of the other, in this setting, we want them to be able to act independently.

Bob could choose one of the following two non-speculative strategies. The first is for Bob to take a bet on one of the choices and commit to that choice. For example, choose on the first choice that gets to make some progress. That is, if a good lens is for sale, then buy the good lens and take a risk by waiting for a buyer for the average lens. This ignores the second possibility to buy a better camera. This "bet-and-risk-it" strategy has obvious pitfalls. What happens if one makes the wrong choice? In the example above, if Bob's program finds a match with Jill's choice of selling good lens first, then the lens (1^{st}) choice of Bob will be committed and the camera (2^{nd}) choice will be eliminated. But as it turns out, Jill wants to buy average camera and not average lens after she has bought a good lens. As such, Bob and Jill are stuck in a deadlock both waiting to complete their trade.

The second and more conservative strategy, is that Bob could wait in both of his choices until the condition is satisfied for both buying and selling actions to carry out together, and then does the actions atomically. While this is a safer option than the previous one, Bob will certainly miss the trading opportunity with Jill as Jill will not buy his average camera until she has sold her good camera. Both parties will be blocked even when there is a potential solution available.

Though our simple example has only two players, in a real life marketplace, much larger dependency cycles involving more parties may exist. These cyclic dependencies can be resolved through the use of late commit but not the early commit strategies discussed above.

4.1.2 The Generalized Committed Choice Model

The generalized committed choice allows the following strategy, which increases the probability of getting a solution. As Bob has two choices, to maximize his chances, he would like to be able to attempt both choices simultaneously and non-deterministically, and choose the one which succeeds. This leads to a form of speculative computation. We achieve this by having the computation from each choice operate in its own independent "world" containing an independent store. So one world does not effect the other. Now, when Jill comes to the system, her program will join the two existing worlds Bob's program has created. Since Jill also has two choices, her program will further split each world it is living in, and this creates four worlds altogether, each of which represents a possible interaction between Bob's and Jill's choices.

Notice here everybody is given all the opportunities to complete their actions: while Bob may take Jill's good lens and get stuck in one world since he cannot sell his good lens; he may able to buy Jill's good camera and sell his average camera to her in another world and then eventually complete the transaction. Thus we also need a way of removing unwanted possibilities which represent other worlds and computations. In every world, Bob's and Jill's computation operate in their own independent reality. They can buy and sell items as if there is no speculation.

A *solution* in this model is loosely defined to be the successful completion of all interacting programs, to the "satisfaction" of the agents that issue these programs. Given a number of interacting programs, there may be a number of possible solutions. The GCC model aims at finding the *first* feasible solution, rather than the "best" solution or all solutions. This is so because GCC is designed to be used in open system in real time.

The model of GCC enables a programming paradigm where a computation can have a number of distinct possibilities. For simplicity, let us say there are two choices, α and β. We assume that the computations operate on a shared store which can change over time due to external events or through actions of a program. In a choice construct, we allow both α and β to proceed concurrently but isolated from each other. At the same time, there are other computations which are also concurrently interacting with this store. After some computation, one of the choices, say α can choose to commit. This has the effect as if the other possibility β never existed. When we have more than one user, all user choices are multiplied to form a number of worlds. The speculation here is that the result of a computation can be a number of worlds.

This chapter proposes a programming model for speculation in a new don't-know nondeterminism and concurrency context. The central contribution is a new programming construct GCC which generalizes early committed choice. We formalize the complicated semantics of GCC. The main challenge of the semantics is to deal with the notion of commit in the context of multiple worlds.

4.1.3 Related work

Transaction A basic database transaction provides *atomicity* and *isolation* [RG02]. This can be used in the second strategy depicted above, but since it is unlikely all blocking conditions are satisfied all at once, this method will not work for the example above most of

the time. Finally, normal relational databases and SQL do not handle non-determinism, so there is no way for Bob or Jill to specify their choices.

Long-Lived Transaction Long-lived transactions such as "Sagas" [GMS87] do away with the isolation property of the transactions and allow partial changes to the database to be visible to other transactions. Also, one level of nested transactions are allowed in Saga. The problem of saga is that the operations can be unsafe, that is, if one process cannot go through, then the whole saga needs to be compensated. But since the processes can interleave, sometimes no compensation operations are possible, and hence the system becomes irreparably inconsistent. The saga model does not handle the above example firstly because it does not provide non-deterministic choice; secondly, even if choices are possible, multiple choices are operating in the same environment, which means once Bob has bought the good lens for example, he will not have the money to pay for good camera, even if it is available. Furthermore, the compensating transactions have to be provided by the user program rather than being resolved by the system.

CCP and Deep Guards Dijkstra's guards and concurrent constraint programming (CCP) [Sar93] techniques such as GHC [Ued86] and Oz [Smo95, RBD+03] use *early committed choice* to handle non-determinism. Early committed choice can be used to implement the *bet-and-risk-it* strategy, but as we have shown, this may lead to a blocked/deadlocked computation. Furthermore, CCP languages require monotonic stores and is thus not applicable in our context. We point out that *deep guards* in CCP are still early committed choice since although they could include non-determinism and computation, a deep guard is *not* allowed to affect the store.

Transaction Logic $\mathcal{TR}$ and especially the later Concurrent Transaction Logic ($\mathcal{CTR}$) [BK93b, BK96] are concurrent, rule-based update programming languages for transactions. It combines long-lived transactions and early committed non-determinism in the framework of logic databases such as Datalog. It does not support our camera example for the same reasons given in the discussion on saga and early committed choice.

Composable Memory Transactions The `orElse` construct in [HMJH05] provides a handle to state several *alternatives* in memory transactions. The transaction `s1 ‘orElse‘ s2` first runs `s1`; if it blocks (retries), then `s1` is abandoned with no effect, `s2` is run. If `s2` is

also blocked (retries), then the whole transaction retries. In a way, the `orElse` construct offers a don't know type of non-determinism as it attempts the choices one by one until a satisfying one is found. However, the semantics of `orElse` is different from that of GCC, because the choices are attempted sequentially in the former, while GCC attempts all choices simultaneously.

Active Databases While the *Event-Condition-Action* rules in active databases [VB98, DHL90] provide some degree of reactivity to the users, non-determinism is only achieved with early committed choice of which rule to fire first. In addition, the actions carried out in ECA rules are either simple database read/write operations or user-defined procedure calls. But in all cases, these actions are done atomically and in isolation, hence interleavings are not possible.

4.2 Programming with GCC

We illustrate the programming paradigm of GCC with the following simple setting. Let there be a number concurrent or parallel programs (processes) interacting with a common runtime system, which provides a global computation environment or a global memory we call a *store*. Without loss of generality, we assume programs do not use any local variables. Synchronization is achieved by use of the common store and the use of a blocking guarded action. For now, we simply assume the store is a piece of shared memory which contains (variable, value) pairs. The main operation on the store are variable assignments which are atomic.

In what follows, we present some simple programming constructs as for programming with GCC, and also the compiled form of the stylized language. It should be clear from this section that although we have used a simple setting, GCC can be integrated into more complex programming languages and concurrent and parallel systems.

4.2.1 The stylized language

We introduce the following minimal concurrent language in which we have added GCC. The grammar of a program r in this chapter is defined inductively as:

r ::=		
	`noop`	No operation
|	$x := v$	atomic assignment
|	`if` c `then` r_1 `else` r_2	conditional
|	`while` c `do` r_1	loop
|	$c \Rightarrow \delta$	guarded atomic action
|	$r_1;\ r_2$	sequence
|	$r_1 \ \oplus \ r_2$	GCC
|	`cm`	commit this choice
|	`cu`	commit other choice

The first four constructs are rather standard and thus require little explanation. The assignment operation assigns a value v to a global variable x in the store atomically. Boolean condition c is tested in both the conditional, loop and guard constructs. An example of c is $x + y \leq 10$.

Guarded atomic action, or guard in short, is provided to give rise to allow for reactive behavior and enable synchronization among the programs. $c \Rightarrow \delta$, blocks until condition c is true w.r.t. the store and then atomically executes δ. δ is an action such as noop, assignment, `cm` and `cu`, all w.r.t. the store.

The choice construct, $r_1 \ \oplus \ r_2$, defines two computations r_1 and r_2 which are to be executed speculatively. For simplicity, in this chapter, we only deal with binary choices and it is straightforward to extend this to an arbitrary number of alternatives. Both r_1 and r_2 execute with any updates isolated from each other. Nested choices are allowed. Unless otherwise stated, in the remainder of this chapter, we refer to generalized committed choice simply as *choice*.

Within a choice, there are two special operations, namely `cm` and `cu`, which stand for "commit me" and "commit you". These can only be used within the scope of a choice and they are referring to the innermost enclosing choice structure. `cm` expresses the intention to *commit* to this branch of a choice and to remove other branch as if it did not exist; `cu` expresses the intention to *terminate* this branch of a choice and commit to the other branch. Notice that `cm` and `cu` are *not* symmetrical since after executing `cm` in a branch, the program can continue, whereas in the case of `cu`; the program instance will be *halted*. If `cm` and `cu` are used outside the scope of a choice, they have no effect. Furthermore, only the first use of `cm` or `cu` has any effect

within a choice branch. We require that every choice branch must have a commit operation, the intent is that a choice must eventually be "committed". This can be achieved by adding `cm` to the end of every choice branch.

4.2.2 The abstract assembler language

The above given stylized language serves as the syntactic sugar for the programmers. In practice, programs written in the above syntax will have to be compiled into an abstract assembler language first before being executed by the GCC runtime system. Below is the set of instructions in the assembler language and a brief explanation of their use. Every instruction below takes one step to execute. The exact semantics of these instructions will be discussed later in section 4.4.

- `noop`

  does not do any operation but one time step elapses.

- `x := v`

  assigns the value v to variable x.

- `cm`

  denotes commit this choice branch.

- `cu`

  denotes commit the other choice branch and terminate this branch.

- `test(c, pc1, pc2)`

  tests condition c, if true go to program counter `pc1`, else go to program counter `pc2`. This is used in both the conditionals and the while loops.

- `goto(pc)`

  goes to program counter `pc`. `goto` is used in the while loops and choices.

- `guard(c, δ)`

  tests condition c, and if true execute δ in one step, else does nothing. Note that δ is an operation that is either `noop`, `x:=v`, `cm` or `cu`.

- `begin_choice (pc1, pc2)`

  denotes the begin of a choice construct and specifies the next program counter for both

branches. The left branch goes to `pc1`; and the right branch goes to `pc2`.

- `end_choice`

  denotes the end of a choice construct.

- `halt`

  halts the current program.

To illustrate the use of the above stylized language and the assembler language in the compiled form, let us translate the pseudo-code of Bob in section 4.1.1 as follows:

Stylized syntax:

$$
\begin{aligned}
&(goodlens \geq 1 \Rightarrow goodlens := goodlens - 1;\\
&averagelens := averagelens + 1); cm\\
&\oplus\\
&(goodcam \geq 1 \Rightarrow goodcam := goodcam - 1;\\
&averagecam := averagecam + 1); cm
\end{aligned}
$$

Notice that for simplicity, we have not dealt with the details of the transactions and have only expressed the requirements of the example in terms of availabilities of the items.

Compiled form in assembler code:

```
<0> begin_choice(<1>, <5>)
<1>  guard(goodlens>=1, goodlens:=goodlens-1)
<2>  averagelens:=averagelens+1
<3>  cm
<4>  goto(<8>)  //This ends left branch

<5>  guard(goodcam>=1, goodcam:=goodcam-1)
<6>  averagecam:=averagecam+1
<7>  cm
<8> end_choice
<9> halt
```

4.3 Stores and Worlds

In this section, we formalize a suitable runtime structure for GCC. We begin with a conventional definition of a single store, which is then extended to multi-stores and multi-worlds.

4.3.1 Stores

Figure 4.1: A single store Δ

Programs work with and interact with other programs using a store. We will assume that a store, denoted by Δ represents a set of variable-value pairs $(x,\ v)$, for example, $x = 5$, $y = 8$, etc. So the store functions as shared memory, albeit slightly higher level, which a number of concurrent programs can access and modify. As we will be considering more than one store, we will also annotate a store with a version number, i.e. we depict a store graphically as a triangle with a version number in Fig. 4.1.

We will also make use of intersection and union of stores which is defined as follows:

$$\Delta_1 \cap \Delta_2 = \{(x,\ v) | (x,\ v) \in \Delta_1 \ \wedge (x,\ v) \in \Delta_2\}$$

$$\Delta_1 \cup \Delta_2 = \{(x,\ v) | (x,\ v) \in \Delta_1 \ \vee (x,\ v) \in \Delta_2\}$$

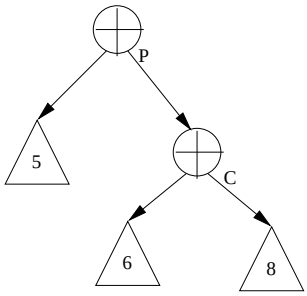


Figure 4.2: Multi-stores

The runtime structure of a program using GCC in general consists of a collection of stores organized in a tree structure. We call this structure (see Fig. 4.2) a *multi-store*. The leaves of the tree are the stores, and the intermediate nodes are choice nodes $\oplus_{id}$, where id is a unique identifier. The multiple stores are used to represent different possible values for the variables in the store which may have been modified inside alternatives of a choice.

4.3.2 Continuations and worlds

The operational semantics of the GCC and the program language is centered around the creation, evolution and deletion of a number of worlds. Before we introduce the notion of worlds and multi-worlds, we first define a program *continuation.*

Definition 4 (Continuation) *A program continuation σ is a triple* `<P, pc, cids>`*, where P stands for the program code (the abstract assembler code), pc is the program counter which initially is 0, and cids is a stack of choice ids dynamically generated for each choice construct being processed in this program. Initially cids is the empty stack.*

The runtime system with GCC is characterized as a number of program continuations of executing programs which may be updating the various stores in the multi-store.

Definition 5 (World) *A world is a pair (Δ, Σ), where Δ is a shared store, and Σ is a set of continuations of all the programs interacting with Δ.*

In other words, every store in the runtime system is associated with a set of continuations. Programs execute when the system picks one continuation from the set, and advances it by executing an instruction, known as a program step.

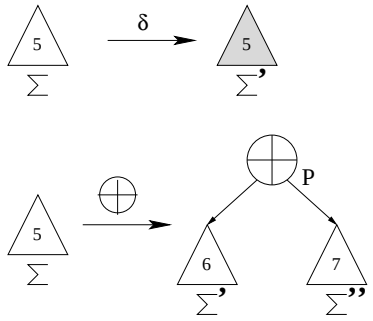


Figure 4.3: Worlds and multi-worlds

Given a world, the execution of one program step by one of the continuations either evolves the world to a new one (the store and continuation changes), or it can split the world into two new worlds using a choice with two alternatives, i.e. a choice point. This is illustrated in Fig. 4.3. The upper transition is the case when a continuation issues an update δ, i.e. x := 10, which changes store Δ_5 to its new state (denoted by the gray color). The lower transition shows the splitting of one world into two, when a continuation σ issues a choice point $\oplus_{id}$, and this creates two new stores, Δ_6 and Δ_7, each of which is a copy of the original store Δ_5, and also two

sets of continuations Σ' and Σ'' for each branch in the choice point. We call the tree runtime structure with multiple worlds as a result from executing various choices, a *multi-worlds*. Later, we will see how `cm` and `cu` help reduce the number of worlds by "chopping" sub-trees from the multi-world.

A multi-world can be formally defined as a tree:

$$\begin{array}{rcll} \mathcal{T} & ::= & \Delta_i, \Sigma & \text{(single store with continuations)} \\ & | & \mathcal{T}_1 \oplus_{id} \mathcal{T}_2 & \text{(tree of stores with continuations)} \end{array}$$

We can think of $\oplus_{id}$ as a logical XOR, and the multi-worlds is just the union of the a collection of worlds organized in a tree structure. Note that a subtree of multi-worlds is also a multi-world.

4.3.3 Global GCC runtime considerations

We envisage GCC to be used in an open environment where GCC programs can be submitted to the GCC runtime system. Thus, the composition of programs in the system can be dynamic. In this chapter, we do not go into the details of how programs can be started in practice, however the formalism and semantics is compatible with the dynamic and open mode of running programs.

Furthermore, in practice we would like to react to external events outside the control of GCC, i.e. speculate on observations from the external environment. We would also like to be able to control and observe the computations within the GCC runtime system from the outside. In this subsection, we briefly discuss these considerations.

The external environment

External variables are variables in the store which is only modified by the external environment. Thus, programs can only read external variables but not assign to them. One particularly useful external variable is the *time* variable which can be used as a global clock. This is meant to change every clock tick. Since external variables really represent an observable outside the store, when there are multiple stores, they all refer to the same external variable.

Time variables are useful to write conditions involving external time. For example, this can be used to express timeouts in the following fashion. Consider a program fragment r which is

to be executed as long as the timeout hasn't expired. This is expressed as,

$$r \quad \oplus \ (time \geq x) \Rightarrow \texttt{cm}$$

so that when time has reached x, the right branch can commit and the effect is that r is aborted.

We can also give control of choices to external variables which can allow a choice to be preempted. Suppose we have an external variable *preempt* and consider one computation r_1 and an alternative r_2 which is for preemption purposes. The following schema expresses preemption,

$$r_1 \ \oplus \ (preempt = v \Rightarrow noop; r_2) \Rightarrow \texttt{cm}$$

Setting the external variable *prempt* to v from outside the system, will kill r_1 and execute instead r_2.[1]

Querying the multi-world

The purpose of using GCC is that instead of a single world, we have multiple worlds. Thus, the multi-world tree is the result of the computation. In the case when the multi-world only contains a single world, then the information in the store is unambiguous. Otherwise, the multiple worlds give the current range of possible values of the store at the current point in the computation.

In general, we may want to extract some general information about the stores. We first define two kinds of views, conjunctive and disjunctive. A conjunctive view, $\mathcal{CV}$, takes a multi-world tree and returns the common information from all the worlds,

$$\begin{aligned} \mathcal{CV}(\Delta_i, \Sigma) &= \Delta_i \\ \mathcal{CV}(\mathcal{T}_i \oplus_{id} \mathcal{T}_j) &= \mathcal{CV}(\mathcal{T}_i) \cap \mathcal{CV}(\mathcal{T}_j) \end{aligned}$$

A disjunctive view, $\mathcal{DV}$, takes a multi-world tree and returns all the information from all the worlds,

[1] Of course, this schema can also be used with internal store variables and another program to control the preemption internally.

$$\begin{aligned}\mathcal{DV}(\Delta_i, \Sigma_i) &= \Delta_i \\ \mathcal{DV}(\mathcal{T}_i \oplus_{id} \mathcal{T}_j) &= \mathcal{DV}(\mathcal{T}_i) \cup \mathcal{DV}(\mathcal{T}_j)\end{aligned}$$

From the outside, a conjunctive view from the root of the multi-world is useful to extract the definite information which is in the current state of the multi-world computation. A disjunctive view on the other hand will show the current range of possibilities in the computation. We remark that this is one general way of observing the various speculative computations going on in a multi-world without looking at the details. Observation from the outside can also be used in conjunction with external variables to directly control the evolution of the choices and commits.

4.4 Operational Semantics

We can now describe the operation semantics of GCC and the simple programming language in terms of transitions on multi-world states. Initially, there is a single world. This can then evolve into a multi-world through the use of choice. The computation can possibly become more determined once commits reduce the alternatives. It may reduce down to a single world again. Each world in a multi-world is independent from each other and evolves separately. In the following semantics, the system will either execute a *program step* (P-step), a *commit step* (C-step), or an *environment step* (E-step),

4.4.1 P-step

A P-step advances one of the continuations in a world, i.e. a program gets to execute. In what follows, a P-step is applied to a selected world, (Δ, Σ), from the multi-world. A continuation σ is selected from one of the current executing program continuations Σ and one of the P-step rules are applied. In general, the P-step rules have the following form:

$$\Delta, \sigma \longrightarrow \Delta', \sigma' \tag{4.2}$$

where the store can possibly be changed to a new store, Δ' through an update from the program and the new continuation is σ'. We use $next(\sigma)$ to denote a continuation which is the same as

σ except that the pc refers to the next assembler instruction to follow the one in σ; $\mathcal{I}\ (\sigma)$ to denote the abstract assembler statement at continuation σ; and $\Delta[x = e]$ to denote a new store where the value of x is replaced by e.

We first look at P-steps which do not involve choices and commits. The rules for the transition $\Delta, \sigma \longrightarrow \Delta', \sigma'$ for each of the following abstract assembly instructions are as follows:

- Assignment: $\mathcal{I}\ (\sigma) = (x := e)$

  The new store $\Delta' = \Delta[x = e]$, i.e. it is updated with the assignment, and $\sigma' = next(\sigma)$. Note that external variables are read-only and cannot be updated.

- Test: $\mathcal{I}\ (\sigma) = (test(c,\ pc_1,\ pc_2))$

  The store is unchanged, $\Delta' = \Delta$. If $\Delta \models c$ then σ' is the same as σ except the program counter becomes pc_1; otherwise the program counter in σ' becomes pc_2.

- Goto: $\mathcal{I}\ (\sigma) = (goto(pc))$

  The store is unchanged, $\Delta' = \Delta$, and σ' is the same as σ except the program counter in σ' becomes pc.

- Noop: $\mathcal{I}\ (\sigma) = (noop)$

  The store is unchanged, $\Delta' = \Delta$, and $\sigma' = next(\sigma)$.

- Guard: $\mathcal{I}\ (\sigma) = (guard(c,\ x := e))$

  This executes when $\Delta \models c$ with the new store $\Delta' = \Delta[x = e]$ and $\sigma' = next(\sigma)$. So the guard only executes the $x := e$ when the condition c is true given store Δ and otherwise blocks execution of the instruction. In practice, the blocking and re-enabling of programs requires a *trigger mechanism* [JYZ05], which is not the focus of this chapter.

- Halt: $\mathcal{I}\ (\sigma) = (halt)$

  As the program halts, we simply remove its continuation from the world, therefore the transition is $\Delta, \sigma \longrightarrow \Delta$

  The store is unchanged.

The semantics for the non-choice constructs in the language as shown above is straightforward because it deals just with a single world, so it only deals with one store and the pc part of the continuation. We now turn to the choice and commit constructs which affect the multi-world. From one world, the choice construct creates two possible worlds. Each world begins

with the same store and has its own continuation for the computation for each alternative of the choice. In the compiled assembly language form, the end of the scope of a choice is denoted by an `end_choice` instruction.

- `begin_choice`: $\mathcal{I}\ (\sigma) = (\mathit{begin\_choice}(pc_1,\ pc_2))$

  The transition which creates results in two new worlds is

  $$\Delta,\ \sigma \longrightarrow (\Delta,\ \sigma') \oplus_{id} (\Delta, \sigma'')$$

  where $\sigma = \langle P, pc, cids\rangle$, id is a fresh identifier for the multi-world choice constructor $\oplus$, $\sigma' = \langle P, pc_1, \mathit{push}(id,\ cids)\rangle$, and $\sigma' = \langle P, pc_2, \mathit{push}(id,\ cids)\rangle$. Recall that $cids$ is a stack of choice identifiers in the continuation. Thus, $push$ returns a new stack with id pushed on top.

- `end_choice`: $\mathcal{I}\ (\sigma) = (\mathit{end\_choice})$

  As this is executed in a single world, the transition is

  $$\Delta,\ \sigma \longrightarrow \Delta,\ \sigma'$$

  where the store is unchanged, $\sigma = \langle P, pc, cids\rangle$, and $\sigma' = \langle P, pc', \mathit{pop}(cids)\rangle$. The new pc' refers to the next sequential instruction and pop returns a new stack with the topmost element removed.

Notice that although the `end_choice` instruction is executable in each of the alternative worlds in the choice, they are managed independently since the continuations are separate.

The intent of the commit instructions is to be able to remove some worlds. This is achieved with a combination of a P-step which has the effect of a special store update and a C-step which will delete some (possibly zero) worlds. The commit rules for the transition $\Delta,\ \sigma \longrightarrow \Delta',\ \sigma'$ are as follows and we will also use a $peek$ function which returns the topmost element of the stack:

- cm: $\mathcal{I}\ (\sigma) = (cm)$

  Let $\sigma = \langle P, pc, cids\rangle$. If the the stack $cids$ is empty, then the cm behaves like a noop, so $\Delta' = \Delta$. Otherwise, the cm is in the scope of a choice. Let $id = \mathit{peek}(cids)$. The new store $\Delta' = \Delta[cm_{id} = 1]$ records that this alternative has committed. In all cases, $\sigma' = \mathit{next}(\sigma)$. Notice that executing another cm has no effect, thus multiple cm's are idempotent.

- cu: $\mathcal{I}\ (\sigma) = (cu)$

  Let $\sigma = \langle P, pc, cids\rangle$. If the the stack $cids$ is empty or $(cm_{id},\ 1) \in \Delta$, then the cu behaves

like a noop, so $\Delta' = \Delta$ and $\sigma' = next(\sigma)$. The second case above is when a cu occurs after a cm, therefore it has no effect as the choice has already been committed. Otherwise, we want to commit the other branch. Let $id = peek(cids)$. The new store $\Delta' = \Delta[cu_{id} = 1]$ records that the other alternative is to be chosen. As cu should now halt, the transition for a cu which commits is

$$\Delta, \sigma \longrightarrow \Delta$$

since its continuation is removed.

Both rules set a flag in the store which signifies that the corresponding choice identifier has committed in a world. Note that cm and cu are not symmetric here.

4.4.2 C-step

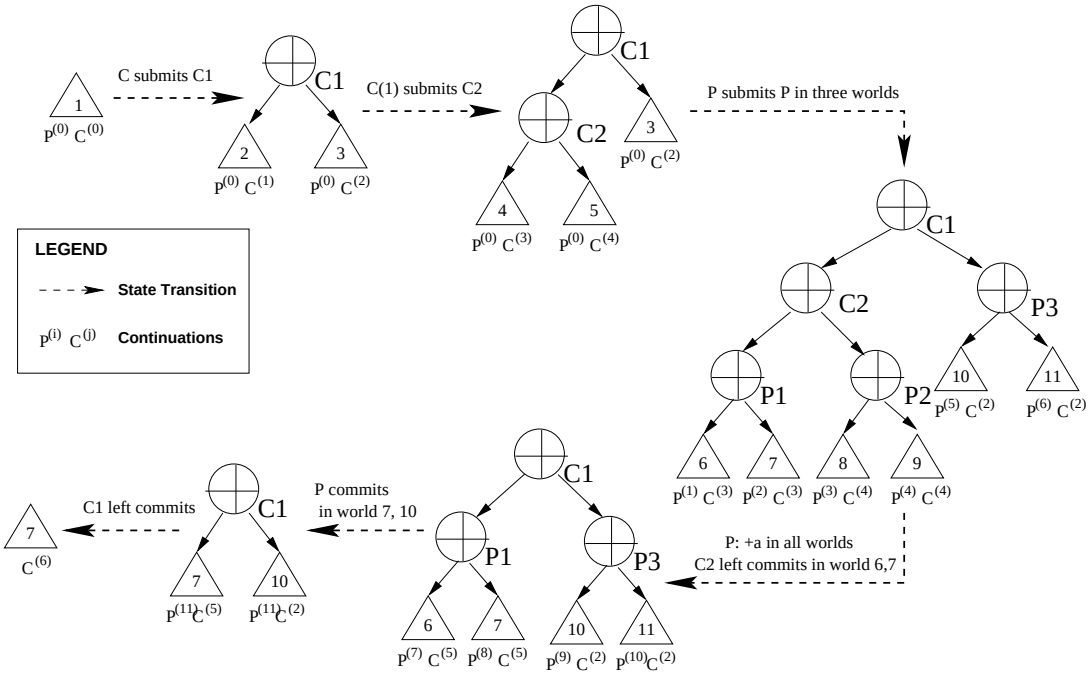
A C-step is used to remove those worlds in a multi-world which correspond to the other alternative in a choice which has executed a commit. We now give the rules for *coordinated commit*. Consider a multi-world with a subtree of the form $\mathcal{T}_1 \oplus_{id} \mathcal{T}_2$, the C-step rules are:

- $\mathcal{T}_1 \oplus_{id} \mathcal{T}_2 \longrightarrow \mathcal{T}_1$ if $\mathcal{CV}(\mathcal{T}_1) \models \mathtt{cm}_{id} = 1$. This removes the worlds in $\mathcal{T}_2$ because the coordinated commit condition means that all the worlds in alternative $\mathcal{T}_2$ have committed to the left choice.

- $\mathcal{T}_1 \oplus_{id} \mathcal{T}_2 \longrightarrow \mathcal{T}_2$ if $\mathcal{CV}(\mathcal{T}_1) \models \mathtt{cu}_{id} = 1$. This is similar to cm except that that the commit is on the opposite alternative of the choice.

Note that the C-step rules are used in conjunction with the P-step rules. In some worlds of the multi-world, commits corresponding to this choice are executed. Coordinated commit then selects some worlds to keep and some to remove from the multi-world based on which commits have been executed in the multi-world choice structure. It is called coordinated commit since only when all the worlds which arise from a choice sub-tree in the multi-world have consistently committed to the same alternative do we allow the worlds in the other alternative to be removed.

Coordinated commit is just one possible semantics for commit. In this section, we have focused on coordinated commit as it is a reasonable choice for commit. A discussion which compares coordinated commit with other alternatives is given in section 4.5.



Continuations Listing:

$P^{(0)}$: $\langle P,\ 0,\ [] \rangle$	$P^{(1)}$: $\langle P,\ 1,\ [P1] \rangle$	$P^{(2)}$: $\langle P,\ 5,\ [P1] \rangle$	$P^{(3)}$: $\langle P,\ 1,\ [P2] \rangle$
$P^{(4)}$: $\langle P,\ 5,\ [P2] \rangle$	$P^{(5)}$: $\langle P,\ 1,\ [P3] \rangle$	$P^{(6)}$: $\langle P,\ 5,\ [P3] \rangle$	$P^{(7)}$: $\langle P,\ 2,\ [P1] \rangle$
$P^{(8)}$: $\langle P,\ 6,\ [P1] \rangle$	$P^{(9)}$: $\langle P,\ 2,\ [P3] \rangle$	$P^{(10)}$: $\langle P,\ 6,\ [P3] \rangle$	$P^{(11)}$: $\langle P,\ 9,\ [] \rangle$
$C^{(0)}$: $\langle C,\ 0,\ [] \rangle$	$C^{(1)}$: $\langle C,\ 1,\ [C1] \rangle$	$C^{(2)}$: $\langle C,\ 10,\ [C1] \rangle$	$C^{(3)}$: $\langle C,\ 2,\ [C2,\ C1] \rangle$
$C^{(4)}$: $\langle C,\ 5,\ [C2,\ C1] \rangle$	$C^{(5)}$: $\langle C,\ 8,\ [C1] \rangle$	$C^{(6)}$: $\langle C,\ 12,\ [] \rangle$	

Figure 4.4: A worked example

4.4.3 E-step

An E-step occurs when an external variable, lets say x, is changed by the external environment. In an E-step, all occurrences of x in the multi-world are changed to its new value.

4.4.4 An example of multi-world transitions

Consider the following producer/consumer example. We use a simplifying notation to increase readability. We write $?x$ to denote a guard that it is possible to consume, and the condition is $(x \geq 1)$. Production is denoted by $+x$, represented as the assignment $x := x + 1$. Consumption is denoted by $-x$, which is $x := x - 1$.

$$P ::= \quad (+a;\ \ +b;\ \texttt{cm}) \ \oplus \ (+a;\ \ +c;\ \texttt{cm})$$

$$C ::= \quad ((?a \Rightarrow -a;\ \texttt{cm} \ \ \oplus_2 \ \ ?b \Rightarrow -b;\ \texttt{cm});\ \texttt{cm}) \oplus_1$$

$$((time \geq 10) \Rightarrow \texttt{cm})$$

For convenience of discussion below, we have numbered the choice constructs in program C. $Time$ is an external variable representing an external clock. The programs can be translated into assembler as follows:

```
Reactor P:
<0> begin_choice(<1>, <5>)
<1>  a:=a+1
<2>  b:=b+1
<3>  cm
<4>  goto <8>  //end left choice branch

<5>  a:=a+1
<6>  c:=c+1
<7>  cm
<8> end_choice
<9> halt


Reactor C:
<0>  begin_choice(<1>, <10>)
<1>   begin_choice(<2>, <5>)
<2>    guard(a>=1, a:=a-1)
<3>    cm
<4>    goto <7> //end left branch of choice 2

<5>    guard(b>=1, b:=b-1)
<6>    cm
<7>   end_choice
<8>   cm
<9>   goto <11> //end left branch of choice 1

<10>  guard(time>=10, cm)
<11> end_choice
<12> halt
```

We illustrate in Fig. 4.4 the dynamic evolution of the multi-world with the two concurrent programs P and C where the initial store is Δ_1, namely $\Delta_1 = \{a = 0,\ b = 0,\ c = 0,\ time = 0\}$. For the ease of discussion, we number the choice nodes of P as $P1$, $P2$ and $P3$. We denote the continuations attached to each world as $P^{(i)}$ or $C^{(i)}$ with the details of the program continuations given at the bottom of Fig. 4.4.

Suppose C gets to make its choices first, the result is three worlds with the two choice notes $C1$ and $C2$ in the multi-world. Then P starts to issue a choice which multiplies to six different worlds. As P produces a and b in one branch and a and c in another, before P commits in any of its branches, choice $C2$'s left branch can consume a and commit. The commit adds the information $\mathtt{cm}_{C2} = 1$ to both worlds, hence the C-step can be applied to prune off the worlds

of Δ_8 and Δ_9. In the $P3$ subtree, P commits in world 10 first and deletes world 11. P can also commit in the the right branch of the $P1$ subtree, using a C-step to prune off the left subtree of $P1$, namely Δ_6. At this point, there are only two worlds left, Δ_7 and Δ_{10} Since P now has committed in all worlds (7 and 10). Finally, the left branch of choice 2 in C commits which kills world 10 so there is only one remaining world. The speculation within the choices ends up with a definite result.

4.5 On the semantics of commit

The key issue of commit is when a `cm` or `cu` is executed in some world, which other worlds should be deleted and when to delete them. This amounts to which parts of the multi-world to prune and when to prune them.

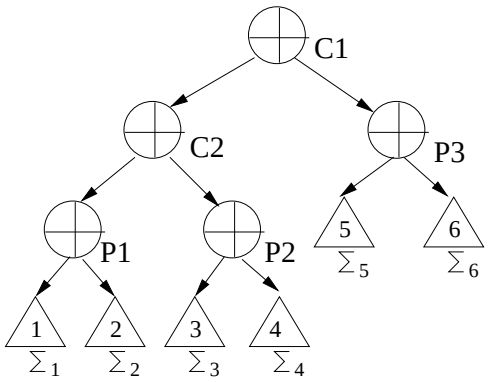


Figure 4.5: A multi-world

For the purpose of discussion, we extract out a multi-world from Fig. 4.4 and show it as Fig. 4.5. As we can see, the choice in P can be duplicated into different parts of the tree, and these choice nodes have different ids, as they are created in different worlds and hence belong to different *scopes*. Subsequently other programs that are interacting with stores Δ_6 through Δ_{11} can also issue choices and further expand the tree. The coordinated commit works as follows. A commit operation not only matches its id syntactically with the choice structure it belongs to, but also maps itself to the scope in which the choice was first launched into. For example, if a commit is executed in one of the worlds under node $P1$ in Fig. 4.5, then only some of the worlds under $P1$ should be deleted and not worlds under $P2$ or $P3$ in the tree. This is accomplished by the choice id stack, *cids*, in each program continuation. A commit always uses the id of the inner-most choice construct that surrounds this commit operation. And since the

choice ids are generated dynamically as choices are issued, the same choice construct will obtain different ids in different scopes. Thus the matching of commits with the correct choice nodes is done automatically.

We further discuss a few other alternatives for the semantics of commit, and argue why the given semantics in section 4.4 is selected. We will illustrate these semantics in Fig. 4.6, Fig. 4.7 and Fig. 4.8. For simplicity, the continuations associated with the worlds are omitted. Where commit has been executed to a store Δ_i, we write $\texttt{cm}_{id}$ under Δ_i in these figures. The stores with the commits of interest are highlighted.

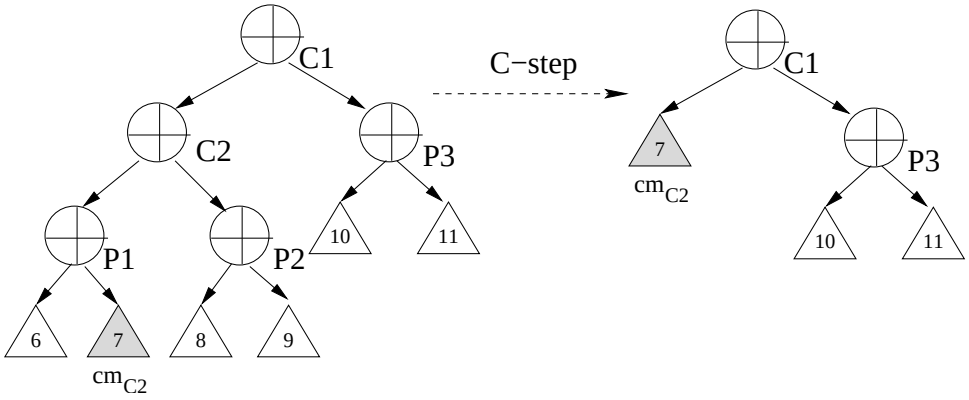


Figure 4.6: Absolutely eager commit

The first kind of commit is an *absolutely eager* commit. Here, the effect of a commits has no delay. If a `cm` whose id is i is reached in any world, this world is committed and all other worlds within the scope of this `cm`, that is, under the subtree rooted at $\oplus_i$, are killed immediately, leaving just one world under $\oplus_i$. That effectively removed all the intermediate nodes including $\oplus_i$ in that subtree (see Fig. 4.6). This for of commit does not seem very useful since this it allows for very minimal inter-play of choices, and the chances of achieving a useful speculation is small since other possibilities are eliminated immediately. In addition, with this semantics, `cu` does not make sense as `cu` may refer to a choice currently in many worlds and the system does not know which world to commit to.

A "less eager" kind of commit is *eager coordinated* commit. For a program $r ::= r_1 \oplus r_2$, if `cm` is reached in one of the worlds where r_1 is executed, then all worlds associated with r_2 are deleted immediately (see Fig. 4.7). The idea here is that syntactically r_2 is not a viable choice any more. Conversely, if `cu` is reached by r_2 in any world, then all worlds associated with r_2 are deleted immediately. However, in either cases, r returns only after r_1 has committed in *all*

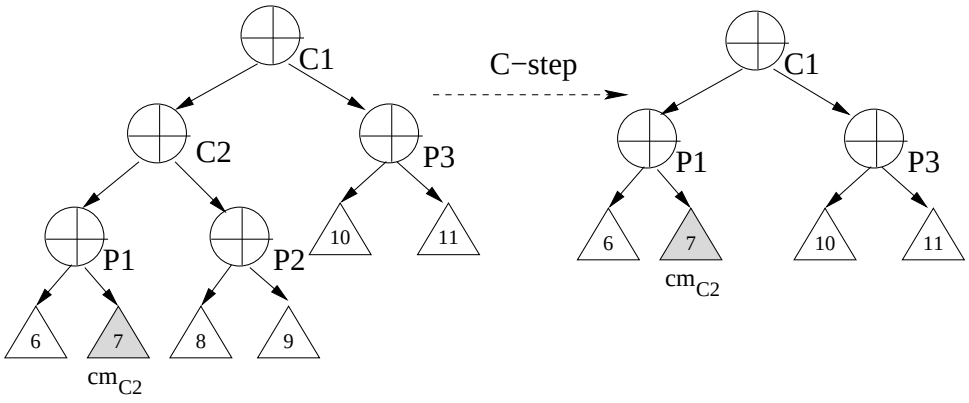


Figure 4.7: Eager coordinated commit

the remaining worlds. This type of commit is "eager" because commit in one world kills the alternative choice in all other worlds; it is "coordinate" as the program only returns after the choice can commit in all remaining worlds. The following example of two programs P and C shows a drawback of this semantics,

$$P ::= \quad (+a; \mathtt{cm}) \ \oplus \ (+b; \mathtt{cm})$$

$$C ::= \quad (?a \Rightarrow -a; \mathtt{cm}) \ \oplus \ (?b \Rightarrow -b; \mathtt{cm})$$

Four worlds are created with these two programs. We denote the world in which the left branch of P and left branch of C interact as $P_l C_l$ and likewise for other worlds. Assume nothing exists in the store initially, P produces a and b in all four worlds but hasn't committed. Now suppose C consumes a in $P_l C_l$ and commits, which kills worlds $P_l C_r$ and $P_r C_r$. However, as it turns out, P now commits in world $P_r C_l$ first and kills $P_l C_l$, which renders C in a blocked state. Had C not killed $P_l C_r$ and $P_r C_r$ early, both P and C may commit in world $P_r C_r$.

A lazy alternative commit semantics is the *late coordinated* commit. Here the system only start removing worlds when a program $r ::= r_1 \ \oplus \ r_2$ with a choice construct has committed its r_1 (or r_2) branch in *all* the worlds with which the r_1 (or r_2) branch continuations are associated. In other words, commits are not only coordinated within a scope, but also across all scopes this program was associated with. For example, in Fig. 4.8, worlds 7, 9 and 11 are only removed when a C-step `cu` in program P has been executed in all these worlds. Note that worlds 7, 9, 11 are the only worlds where the right branch of P exists. While this semantics increases the possibility of getting a solution and decreases the chances of deadlock from a system point of

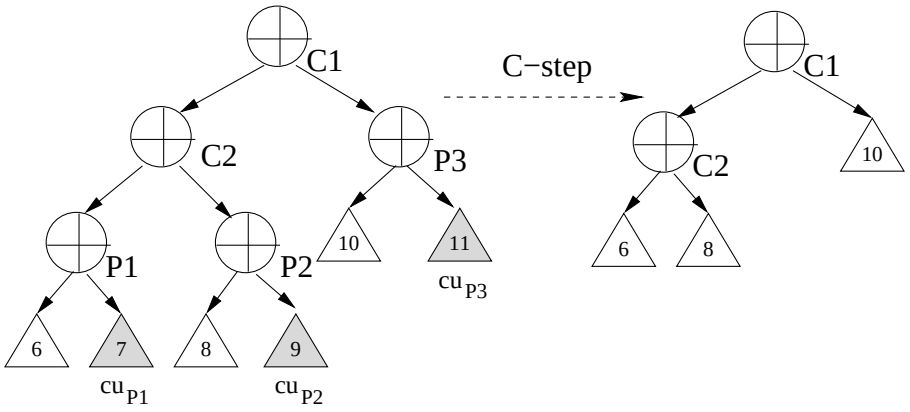


Figure 4.8: Late coordinated commit

view, it is unduly conservative in chopping the tree which may cause the multi-world to be too large. At the same time, there is a potential problem when one branch continuation of r cannot proceed to commit due to blocking, no worlds due to r can be removed from the multi-world. In a certain sense, the lazy coordinated commit is fine but may not be practical since the size of the multi-world may be too large.

The commit semantics introduced in Section 4.4 can be considered as a compromise between the eager coordinated commit and the late coordinate commit. It does not kill the alternative choice until `cm` of this choice has been reached in all worlds in the scope of the choice construct. This solves deadlock problem caused by eager coordinated commit, depicted in the example above. At the same time, it does not over-delay the removal of the worlds so that the size of the multi-world can be contained more effectively.

4.6 Conclusion

This chapter introduced an enhanced programming model for OCP known as the speculative model. It goes beyond the basic model introduced earlier by generalizing the early committed choice so that speculation is allowed. We present the use of multiple worlds to support distributed or concurrent speculative programs. We show how speculations using generalized committed choice is useful to solve a class of problems which cannot be solved by other methods. We give a semantics to formalize the dynamics of the GCC runtime structure, and in particular address the key issue of commit semantics.

Part III

Implementation

Chapter 5

Indexing with RC-tree

This chapter [1] proposes a new main memory index structure for abstract regions (objects) which may heavily overlap. These objects are "dynamic" as they have relatively short life span. The novel feature is that rather than representing an object by its minimum bounding rectangle (MBR) alone or many manually pre-processed small MBRs which is the common practice in many spatial indexes, the actual shape of the object is used to in maintaining the index. Furthermore, this technique can actually save significant space for objects with large spatial extents since pre-segmentation into many MBRs is not needed.

Because conjunctions of linear constraints can be treated as abstract geometrical regions, the techniques proposed here can be used to index linear constraints which forms a basis for triggering blocked conditions in the OCP system. This will become clear in chapter 6. The results in this chapter also have general applicability in geometrical and spatial databases. Thus, this chapter is written purely as a description of the new spatial index structure. Connection of this chapter to the OCP triggering will be made in chapter 6.

5.1 Introduction

Indexing of *spatial objects*, objects with non-zero area or volume in more than one dimension, is an important problem which occurs in databases, computational geometry and computer graphics. The underlying challenge is to provide efficient data structures and algorithms so that a spatial query only needs to check a small fraction of the objects in the database. Such queries are typically point queries, also known as stabbing queries, range queries and nearest neighbor queries.

In this chapter, we want to consider a new class of applications where the spatial objects can

[1] Content of this chapter has been published in [JYZ06]

have significant overlap. Consider a real estate service which tries to match the requirements of buyers with sellers who have properties for sale over some time span. A buyer might have the following requirement: "*a 3-bedroom apartment between 9th to 12th floor, located within 2 km radius of the subway station and along river bank*". Suppose there is no current apartment matching this need. At some later time, a property which can match this buyer's requirement is available, then the service would try to alert the potential buyer. Suppose there are many such buyers, it would be better to create an index so that any new property can be matched efficiently with only a fraction of the outstanding potential buyers.

Another example is a stock trading setting with a large number of traders and a constantly changing market. Stock brokers, like `tradestation.com`, already offer traders the ability to trade using triggers. Here, a trigger is a rule which defines an action to invoke when the rule condition is true. Conditions are trader-defined constraints on stock prices, volumes, interest rate, cash, etc. and can include historical statistics such as moving averages. Each trader has potentially a number of triggers. In a market with lots of traders, the total number of triggers can be very large. Given a market change, rather than attempting to evaluate all triggers, we can use indexing to filter out irrelevant triggers.

We can see that in both examples, one can represent the matching problem as a query to search a spatial index where the objects are *abstract regions* in a multidimensional space. Such abstract regions (objects) can have significant overlap. In fact, in the real estate example, we would expect many buyers to have common desires even though the complete requirement of each buyer is different; and similarly in the stock trading example, traders may have similar interests. By contrast, in traditional GIS applications, the spatial objects are usually tied to some physical, rigid objects which have no or little overlap. Furthermore, the objects correspond to an abstract region could have an arbitrary shape. In applications like the ones above, one could expect the regions to form some kind of geometric shape.

The second example suggests that one class of applications may be to index rules in an active database [WC96] so as to avoid the triggering of irrelevant rules. The use of a spatial index is useful when the rule condition is a complex expression not so suitable for discrimination networks [FRS93] which are based more on dataflow.

Another feature of these applications is that the objects are *dynamic* in the sense that they are mostly *transient*. We would expect that in the first example, the buyers are mostly interested in properties for a period of time. Similarly, in the stock trading example, day traders' triggers

typically do not last for longer than a few hours. Thus, the index will need to support dynamic object insertion and deletion. Although there are insertions and deletions, one would expect as with most databases that queries are more frequent, thus we favor efficient queries over insert/delete performance.

Traditionally, the common representation of spatial objects used in indexing algorithms is their minimum bounding rectangles (MBRs). Rather than working with the actual shape of the object, its MBR approximation is used in the indexing. An object could be either represented using a single MBR or further segmented into a number of MBRs if the object is large. This segmentation process is usually employed as a pre-processing step to the indexing algorithm. Consequently the original shape of the object is lost and not taken into account in the indexing algorithm. Consider an object with a considerable spatial extent, e.g. a freeway in a GIS application, this can be pre-processed into many small rectangles (see figure 5.5). This is okay as GIS applications usually involve just permanent fixtures.

However, in applications which involve dynamic abstract regions, it is not feasible to determine the resolution of the segmentation in advance. Segmentations that are too coarse introduce too much "dead space", i.e. increase the number of misses; segmentations that are too refined would increase the space and time costs unnecessarily. Most importantly, pre-segmentation is *not* a good solution for dynamic regions which have limited life spans.

In the past, indexes were mostly stored in secondary storage, and therefore much of the work has been concerned with indexing techniques that seek to reduce the number of I/O operations because disk I/O access times are several orders of magnitude slower than main memory. This equation is changed with the advent of machines with large address spaces and large main memories. For example, even commodity PCs can have multi-gigabytes of memory. Since the objects in the applications we consider are mostly transient, a memory based index makes good sense. Furthermore, as we will demonstrate, to effectively index the same number of regions, our new index requires much less space than other methods based on pre-segmentation. Thus it becomes possible, and very plausible, to store the index completely in main memory [GC05].

This shifts the problem of indexing since a significant difference exists between disk-based versus in-memory indexing. In disk-based indexing, the number of I/O operations, which is a coarse measure is used; whereas the efficiency of an in-memory index depends on the number of memory operations incurred which is much more fine-grained. As such, the disk-based strategies of increasing the size of index nodes and fan-out and decreasing the height of the tree may

actually have a detrimental effect when applied to in-memory index.

In this chapter, we introduce a novel in-memory data structure for indexing spatial object by taking advantage of the underlying shapes of the objects, as well as their MBRs. Rather than directly using an MBR approximation and processing of the MBRs, the objects are approximated and partitioned dynamically in a lazy fashion, which allows for better discrimination of the objects in the search tree. Such technique avoids some of the problems and costs associated with pre-segmentation.

Our structure, called the RC-tree (*Reducible Clip-Tree*), is a general weight-balanced binary tree that features the clipping of the actual objects and the dynamic re-computation of new MBRs of the clipped objects, a technique we call *domain reduction*. We show that under the assumptions of a problem specific parameter, the *spacing factor*, the RC-tree enjoys amortized logarithmic update time and average case logarithmic search time.

Our experiments on a set of synthetic overlapping data sets and some real life non-overlapping data indicate that RC-tree significantly outperforms the R-tree variants and PMR-quadtree under comparison in both search cost and query accuracy, by orders of magnitude for overlapping data. For non-overlapping real-life GIS data sets, RC-tree is also very competitive as it uniformly performs better than the above methods. It is possible to control the space efficiency and insertion cost of RC-tree by tuning a number of parameters. At the end of the chapter, we present an initial study of the cache effects on RC-tree, and some results on cache-obliviousness.

5.2 Basic Ideas

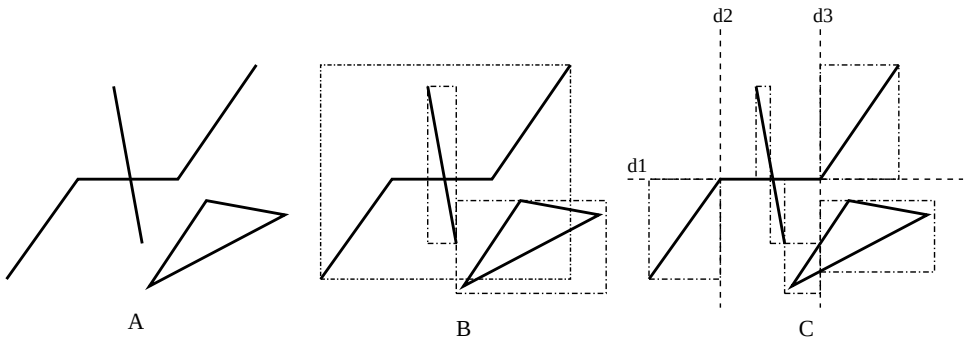


Figure 5.1: Domain Reduction and Clipping Example

The RC-tree combines three main ideas, namely: object clipping, domain reduction and rebalancing.

Like the R^+-tree [SRF87] and the quadtree [Sam90], the RC-tree adopts a space-partitioning strategy. Let us informally define a *discriminator* as a hyper-plane that partitions the space. Indexed objects which intersects with a discriminator are clipped into two parts. Due to the clipping, the MBRs of subtrees at the same level are disjoint, thus avoiding multiple traversals during point-query search. The intermediate MBRs in the RC-tree also helps filter out some objects during the search process.

The difference with the RC-tree is that it indexes actual shapes of the original objects and not their MBRs, so clipping is done on the original objects, and the MBRs of the two resulting sub-objects are used to replaced the original MBR. This way, the size of the MBRs are further reduced. This technique is called *domain reduction*. The idea of domain reduction is illustrated in part (A) – (C) in figure 5.1. Three objects drawn with thick lines are shown in part (A). Part (B) depicts a single MBR representation for each object with the MBR drawn in dot-dashed lines. This is the representation used in R-tree and its variants when there is no pre-segmentation. The MBRs here heavily overlap and there is also a lot of dead space in each MBR. This means that an arbitrary point query would have very low accuracy and multiple search paths have to be followed due to the overlapping of MBRs.

Domain reduction makes use of the discriminators created during the process of building or rebalancing the tree. These discriminators dynamically produce MBR approximations by segmenting the the indexed objects. In part (C), suppose the discriminators are d1, d2 and d3 shown with the gray dashed line. With these particular discriminators, 6 MBRs are created by domain reduction. As a result, there is no overlap between the MBRs and much less dead space in (C) than in part (B). Note that selecting different discriminators would lead to a different set of MBRs.

A good choice of discriminators makes domain reduction more effective. Possible criteria for picking discriminators are based on the shape, placement and number of objects. Because discriminators are created to separate objects of different shapes, the object clipping resulting from the discrimination is generated in a lazy and demand driven fashion. Thus, objects are only segmented when there is a need to discriminate them, which is in contrast with a static MBR segmentation strategy that may create too many (approximation is finer than needed) or too few (approximation is too coarse grained) MBRs.

Domain reduction can also lead to substantial space reduction. We compare RC-tree with another binary MBR-clipping-based search tree which uses the same clipping strategy as in R^+-

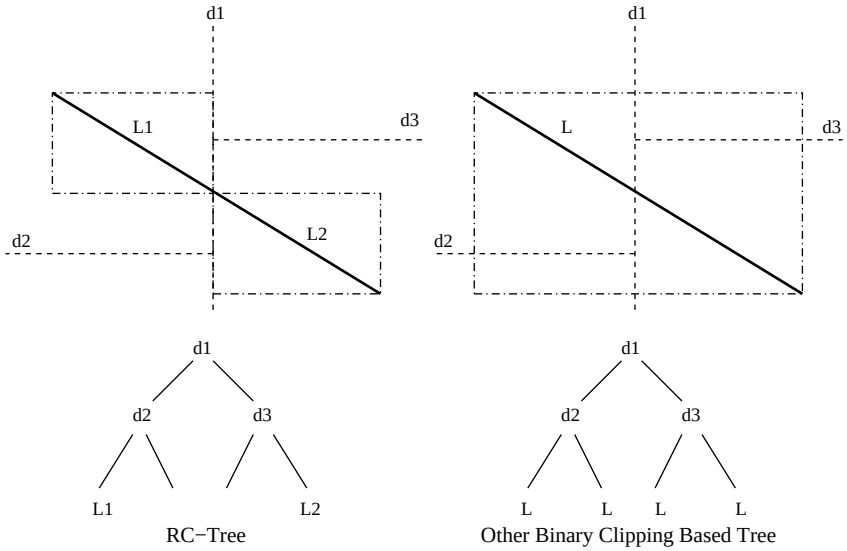


Figure 5.2: Advantage of Clipping and Domain Reduction in Insertion

tree, in figure 5.2. One can see that in this example, the reduction in MBRs in RC-tree creates a tree with only two items ("L1" and "L2") in the leaves, while the other tree has four items (denoted by "L"). Notice the same object duplicating strategy in the latter is used in R^+-tree. 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 duplication strategy inserts four times. Therefore, domain reduction in RC-tree can also reduce insertion time, compared to duplicating strategies.

We remark that domain reduction can yield benefits in general to objects with arbitrary shapes as long as they are not rectangular. There is of course a tradeoff between the choice of representation of the object and the accuracy and cost of the clipping. For example, a more approximate representation of the object could be used for faster clipping. Since RC-tree has no overlapping MBRs in its nodes, to have good search efficiency, it suffices to have a logarithmic tree height bound.

RC-tree uses a binary *general weight-balanced tree* [And99] as its backbone. Rebalancing by partial rebuilding is done whenever the number of nodes in one part of tree is deemed to be out of balance. Rebalancing also serves another important purpose which is to control object clipping and space usage. It repartitions the original objects which reside in given subtree by choosing a series of new discriminators. This would lead to a possibly different collection of MBRs and clipped objects.

5.3 Related Work

Most of the spatial indexing techniques use B-tree approaches which are designed for large nodes with high fanout. The requirement that nodes be at least half full leads to a height balanced tree. R-trees [Gut84] extend the B-tree notion to multi-dimensional search trees. R*-trees [BKSS90] improve the R-tree with a different node splitting algorithm. Subsequently, there are many other R-tree variants which perform better in one scenario or another. But almost all R-tree variants suffers the same problem when it comes to search: in order to maintain the node fill factor and height balance, R-trees allow the the MBRs in the internal nodes of the tree to overlap. This means that search may happen to multiple paths of the tree, and can be expensive, worst case linear time if all internal MBRs overlap even though the data objects do no overlap at all! For formal analytical treatment of the R-tree, readers are referred to [dBGHO03]. An exception is the most recent PR-tree [AdBHY04], which enjoys a worst case optimal bound of $O((N/B)^{1-1/d} + T/B)$, where d is the number of dimensions and T is the output size. PR-tree, however, is a hybrid between R-tree and kd-tree. The PR-tree is meant to be a static index with special bulk loading and search algorithms whereas what we want is a dynamic index.

A notably different R-tree variant, R^+-trees [SRF87] avoids the problem of having to search multiple paths in the search tree by requiring disjoint MBRs. While both R^+-tree and RC-tree do clipping, R^+-tree clips the intermediate MBRs and duplicates the object MBRs. Whereas RC-tree clips the actual objects and applies domain reduction to them. R^+-tree's maintenance is also different from the approximate weight balance approach in the RC-tree. R^+-trees also do not have the the benefit of reducing object clipping by partial rebuilding.

Because geometrical objects are rather complex, most R-tree variants approximate the objects by MBRs and index the MBRs only. Such approximation can be very inaccurate if the data objects are narrow and skewed. As a result, query misses are common in these algorithms. A handful of extensions to R-trees move away from MBR approximations. For example P-trees [Jag90] use polygonal approximations, and SR-trees [KS97] use spheres as estimation. But no B-tree related structures we are aware of index the actual objects without approximation, which is what we do in RC-tree. Furthermore, RC-tree's domain reduction technique, which approximates sub-objects dynamically and on-demand, is not seen in any other structures and algorithms.

A different area of research, largely from the computational geometry community, is data structures that can be used to index spatial objects. Examples are quadtree, kd-tree and its extensions.

Quadtree is a class of hierarchical data structures based on recursive partitioning of space, which is similar to R^+-tree and RC-tree. However, quadtree has fixed branching factor (usually 4) and equal space partitions at the same level in the tree. While the original quadtree was designed for representing point data, the PMR-quadtree [Sam90] is a variant for storing objects of arbitrary shapes, e.g. line segments. Like RC-tree, clipping of the actual objects are used. However, quadtree variants are neither height nor weight balanced, so search performance cannot be guaranteed at all.

Kd-trees [Ben75] extended binary search trees to the case of multidimensional search trees and was originally designed for points rather than spatial objects. Extensions like Skd-tree [OMSD87] and Matsuyama's kd-tree [MHN84] give a form of kd-tree for storing spatial objects of non-zero size. Both of them represent a spatial object by its centroid which is a point. And then they behave like kd-tree in partitioning the points. In Skd-tree the objects can overlap which results in an overlapping partitioning of space, so like R-tree, multiple paths need to be searched. In Matsumaya et al., objects are duplicated in the partitioned regions which intersect with its MBR and thus can lead to excessive duplication with large objects.

Given the large number of spatial indexing techniques, we refer the reader to the excellent survey by Gaede and Gunther [GG98] as well as the survey focusing on R-trees [MNPT]. The approach in this chapter revisits many of the issues in a different way by combining some features in the binary kd-tree, the MBR clipping idea in R^+-tree, and the domain reduction technique. We address the question of excessive space consumption with domain reduction and rebalancing, and also provide ways of tuning the space-time tradeoff, faster search with more space versus slower search with less space.

The way RC-tree dynamically segments objects to be indexed can be compared with the *z-ordering* with *redundancy* in [Ore89, Ore90]. The author investigated the the effect of redundancy on query accuracy and search efficiency and noted a "diminishing return" of redundancy. [Ore89, Ore90] thus suggests some optimum level of redundancy for spatial indexes. However, z-ordering with redundacy method has a predefined maximum resolution of segmentation, and such constraint does not exist in RC-tree. This makes it very different from domain reduction. It is also significantly different because it is a transformed one-dimensional index whereas the

RC-tree is a multi-dimensional index.

Recently, several attempts have been made to address main memory use of R-tree and its variants. Kim et al. [KCK01] introduced a cache-conscious version of R-tree, known as CR-tree, that exhibits up to two times speed-up with its search performance. The main idea in that work is to quantize the MBR entries which allows for a higher fanout for the same node size, given a particular cache line size. However, such technique is largely independent from the underlying data structure and can be implemented in most index structures, including RC-tree. Ross et al. [RSS01] presents a Cost-based Unbalanced R-tree for main memory. CUR-Tree takes advantage of cost functions that characterize the data distributions of the objects, and there is no minimum filling factor constraint. As a result, CUR-Tree is no longer a height balanced tree. The above examples are R-trees made *cache conscious*, which means exact size of the cache has to be known in advance. In the last part of this chapter, we experimented with a *cache-oblivious* version of RC-tree, modeled after the cache-oblivious B-tree [BDFC00]. It is shown in [BDFC00] that such cache oblivious structure matches the optimal query efficiency within a constant factor, without explicit knowledge of the cache and memory parameters.

5.4 The RC-Tree

This section describes the RC-tree and the algorithms for searching and updating the data structure.

5.4.1 Definition

A RC-tree is a general weight-balanced binary tree for efficient search and update of spatial objects in $k-$dimensional space. 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 hyper-plane, it is clipped and the two resulting clipped objects go into the respective subtrees where they belong. The root node is assigned the entire space. The hyper-plane serves to discriminates the objects in two sub-spaces, and hence it is called the *discriminator*. More specifically, every intermediate node contains the following information: discriminator hyper-plane, the number of original objects that are indexed under this node, the MBR of all the

objects (including clipped ones) under this node, and left/right pointers to the subtrees.

All original data objects, either in their entirety or divided in pieces after being clipped, are stored in the leaf nodes. Objects with identical shape and orientation are treated as one single object. A leaf node can store one or more objects. When it stores more than one objects, the leaf node is called an *overflow node*. Overflow nodes store objects as a set without any additional structure.

The objects are defined by a conjunction of constraints O, over a subset of the k variable in the geometrical space. For example, a line segment l passing through the origin could be defined as:

$$2x + 3y = 0 \wedge -5 \leq x \leq 5$$

Discriminators are also modeled as a constraint, e.g. $x \leq 8$ is a discriminator that divides the whole space by a hyper-plane $x = 8$. The constraint approach provides a convenient mechanism for describing and using arbitrary shapes and discriminators.A similar approach which uses constraints to model general discriminators is in [Stu97]. If an object defined by constraint O is contained entirely in the left half-space of a discriminator d, we say

$$O \Rightarrow d.$$

If an object O is clipped by d, the left-hand part is $O \wedge d$, and the right-hand part is $O \wedge \neg d$. All other operators from first-order logic apply with their obvious meanings. Every object has an MBR, which is the projections of its constraint O in each of the k dimensions.

A RC-tree T is called *α-balanced*, if the height of T observes

$$h(T) \leq (1 + \alpha) \log(|T|), \tag{5.1}$$

where $\alpha \geq 0$, and $|T|$ is the number of nodes (intermediate and leaf) in T.

This generalized definition of weight-balance is used as a mechanism to bound the height of the search tree to within a log factor.

5.4.2 Algorithm

Let $T.d$ be the discriminator at node T, and $|T|$ be the number of nodes under T. $T.right$ and $T.left$ are the left child and right child of T, respectively. Let $h(T)$ be the height of the tree

rooted at T.

Algorithm Pack(S, T, Levels)

Input: A set of n^* unordered objects S, $Levels$ to pack into

Output: A weight-balanced RC-tree T of height $\log(n^*)$.

P1. Split(S, $S1$, $S2$, T);

P2. if $Levels > 0$, then

Pack($S1$, $T.left$, $Levels - 1$),

Pack($S2$, $T.right$, $Levels - 1$);

else make T a leaf (or overflow) node that contains S.

$Levels$ is initialized to $\lceil \log(n^*) \rceil$. Pack bounds the tree height by $\log(n^*)$.

Algorithm Split(S, S1, S2, T)

Input: A set of objects S

Output: Two sets of objects partitioned by d, and a tree node T defined by d and the MBR of S.

SP1. select a discriminator $T.d$ using the partition procedure with an objective function f;

SP2. for each $O \in S$:

if $O \Rightarrow T.d$, then $S1 := S1 \cup \{O\}$;

else if $O \Rightarrow \neg T.d$, then $S2 := S2 \cup \{O\}$;

else $S1 := S1 \cup \{O \wedge T.d\}$, $S2 := S2 \cup \{O \wedge \neg T.d\}$.

The choice of discriminator is controlled by the partition objective function which is a heuristic to balance discrimination versus clipping.

Algorithm Insert(T, O)

Input: An RC-tree rooted at T, a new object O

output: A new RC-tree rooted at T

I1. if T is not a leaf, then

if $O \Rightarrow T.d$ then Insert($T.left$, O);

else if $O \Rightarrow \neg T.d$ then Insert($T.right$, O);

else Insert($T.left$, $O \wedge T.d$) and

Insert($T.right$, $O \wedge \neg T.d$);

if $h(T) > (1+\alpha)\log(|T|)$, Rebuild($T$);

I2. if T is a leaf node:

add O to T;

if $|T| > L$, then for set of nodes S in T:

Split(S, $S1$, $S2$, T),

attach $S1$ to $T.left$ and $S2$ to $T.right$.

The leaf capacity, L, is a factor that can be tuned to affect the insertion cost and the space usage. If a leaf node contains more than L objects ($|T| > L$), it is split. However if there is no suitable discriminator to separate the objects in a leaf, no splitting is done and the leaf node becomes an overflow node.

Algorithm Delete(T, O)

Input: An RC-tree rooted at T with n^* original objects, a new object O

Output: A new RC-tree rooted at T

D1. if $delcount == 2^{\frac{\beta}{1+\alpha}-1}n^*$ (see [And99]), then

Rebuild(T), $delcount := 0$;

else $delcount := delcount + 1$;

D2. if T is not a leaf, then

if $O \Rightarrow T.d$, then Delete($T.left$, O);

else if $O \Rightarrow \neg T.d$, then Delete($T.right$, O);

else Delete($T.left$, $O \wedge T.d$), and

Delete($T.right$, $O \wedge \neg T.d$);

if $|T.left| + |T.right| \leq L$, then

extract the objects in $T.left$ and $T.right$,

join the clipped parts and put them in a set S

make a leaf node with objects in S;

D3. if T is a leaf node, then

Remove O from T.

Here, L is use to control how much merging is done which reduces rebalancing. Every top level call to Delete initializes *delcount* is to zero. β is a constant to control the frequency of rebuilding [Ove83].

Algorithm Search(T, W)

Input: An RC-tree rooted at T, a search window W

Output: All objects that intersect W

S1. if T is not leaf, then

if $W \Rightarrow T.d$, Search($T.left$, W);

else if $W \Rightarrow \neg T.d$, Search($T.right$, W);

else return Search($T.left$, W) $\cup$

Search($T.right$, W);

S2. if T is leaf, then

check all objects in T and

return those intersecting with W.

Algorithm Rebuild (T)

Input: An imbalanced subtree rooted at T

Output: A balanced subtree rooted at T.

T1. extract the set of all original objects from T into set S;

T2. Pack (S, T).

5.4.3 Discussion

In the remainder of this section, we will discuss some specific techniques used in the RC-tree algorithm.

MBRs for the intermediate nodes

To speed up the rejection of negative queries, we also add MBRs to the intermediate nodes in the RC-tree. This is not shown in the previous algorithm section. The MBR at any node in the tree is the minimum bounding box of all the objects stored in the subtree rooted at this node. The intermediate MBRs can be updated as new objects are inserted or deleted below the corresponding nodes. Our experiments indicate that adding the MBRs makes searching 70%-80% more efficient than the RC-tree without them.

Partition

The partition procedure is used both in splitting nodes on the leaves, when the capacity of the leaf has been exceeded; and during partial rebuilding. It is used to find a discriminator for the set of objects and as such is critical to the indexing performance. Ideally we want a discriminator that balances the weights of two resulting sets of objects and minimizes the amount of clipping so that the space utilization is minimized. We also want the partitioning procedure to be efficient. Since these goals are often conflicting, we propose two kinds of partition methods: one uses an objective function which combines the effect of weight balance and minimum clipping; the other does fast partitioning and ignores weight and clipping. We call the first method *RC-SWEEP* and the second method *RC-MID*.

In RC-SWEEP, we sort the given set of objects in each dimension in the space respectively. Then we adopt the plane sweep procedure similar to the one in the R^+-tree. Here, we use the boundaries of the objects MBRs as possible candidates for the discriminator. For each candidate discriminator d, we calculate a cost value based on the following objective function:

$$f(d) = \delta(n_l + n_r - n_d) + \sigma|n_l - n_r|, \tag{5.2}$$

where n_l is the number of objects that will go to the left set if d is the partitioning discriminator, and n_r is the number of objects that will go the right set, and n_d is the number of objects that are not clipped by d, or do not intersect d. δ and σ are constant weights which determines if the cost function is biased toward balance or toward minimization of clippings. After sweeping through all the possible candidates in all dimensions, we will pick a d that gives the minimum $f(d)$ value. [2]

[2] A cheaper heuristic is to only try some dimensions.

In RC-MID, we assume that the MBR of the set of the objects is known in advance. This information can be computed incrementally as objects are inserted. Let x be a dimension in which the MBR has the largest extent, and we select a $d = (x_{lb} + x_{ub})/2$, where x_{lb} and x_{ub} are the lower and upper bound of the MBR in x direction, respectively. RC-MID gives a discriminator in constant time, but its effectiveness depends on the distribution of the objects. It is more effective when the objects are roughly similar in size and uniformly distributed in space.

We remark that it is not essential for the discriminators to be orthogonal. An arbitrary hyper-planes/constraints can also be used to partition the space. This is the approach used in Binary Space Partitioning (BSP) [FKN80]. For efficiency reasons, we use orthogonal hyper-planes as an easy constraint discriminator which works well in practice.

Partial rebuilding

To re-balance the RC-tree dynamically, we use a partial rebuilding technique, first introduced by Overmars [Ove83]. In partial rebuilding, after every Insertion(T, O) checks against the balancing criterion of $h(T') \leq (1 + \alpha)\log(|T|)$, where $h(T')$ is the height of the new subtree rooted at T', as a result of the insertion. The balancing criterion ensures that the tree height is with a constant factor of the log of the number of nodes in the tree. Note here we are not using the relative weight of left and right sub-tree to define a weight-balanced tree, as it is commonly defined. Our height-based definition gives a generalized balanced treed that allows for worst case bound on the search time, which is key to indexing. The actual weight difference of the sub-trees is not important.

When the balance of RC-tree is broken, the partial rebuilding finds the lowest node T in the tree where balancing criterion (5.1) is not satisfied, and "flush" out all objects in the subtree into a list. If two objects belong to one original object, they are merged. Thus, at the end of the flush, there are n^* distinct objects in the list. Our algorithm sorts the objects in all dimensions and then applies Split(S, $S1$, $S2$, T) recursively to obtain a new, balanced tree. The recursive split uses the same sorted list as an optimization.

In deletion, a global rebuilding of the whole tree is done once enough number of deletes are done. For details of the proof on complexity of deletion with global rebuilding, please refer to [And99].

Space control

Because clipping techniques generally use more space, a space control mechanism can be applied. Here, a space factor γ can be used to tune the space usage in the RC-tree. If total number of clipped objects in the a sub-tree $N > \gamma n^*$, further splitting is forbidden. This means the algorithm *Pack* may terminate before the counter reaches $\log(n^*)$. If there are still more than L objects in the leaf, this leaf becomes an overflow node.

5.5 Analysis

In this section, we sketch the proofs of our results on the time and space complexity of RC-tree.

To facilitate the analysis, we first define the notation: k is the number of dimensions of the indexed objects. n^* is the number of original data objects to be indexed; n is the number of nodes (intermediate and leaf) in the tree. N is the number of data objects in the leaf nodes. We further define the following parameter.

Definition 6 (Spacing factor) *Given a set of objects S and their MBRs M, let B be the set of all bounding hyper-planes of MBRs in M. Spacing factor s is the largest ratio of the largest extent in any dimension of an object to the smallest distance between any two hyper-planes in M normal to that that direction.*

The spacing factor s is essentially the maximum number of parts any object can be divided into in any direction. In other words, an object can at be "cut up" at most into s^k smaller pieces. Because discriminators are usually defined by boundaries of the objects' MBRs, if the degree of overlapping is roughly constant with respect to the number of original objects n^*, then s can be treated as constant, too.

We further assume that no two data objects have the same MBRs and all the discriminators in the RC-tree are defined by the boundaries of the MBRs of the original data objects.

Our main results are:

1. The amortized insertion cost of a balanced RC-tree of n^* original objects is $O(s^k \log(n^*))$.
2. A point query takes on average $O((1.5)^k(\log(n^*) + s^k))$, and $O(2^k n^*)$ in the worst case.
3. An RC-tree has worst case space $O(s^k n^*)$.

n^*, n and N are related in the following way:

$$2n^* - 1 \leq n \leq 2N - 1 \tag{5.3}$$

This is because without clipping (objects are not overlapped at all), $n = 2n^* - 1 = 2N - 1$. But with clipping, both n and N increase. Where there is overflow node, $N > (n+1)/2$.

Lemma 1 $2n^* - 1 \leq n \leq 2N - 1 \leq 2s^k n^* - 1$

PROOF. Since every object can be clipped into at most s^k parts, $N \leq s^k n^*$, it follows from from (5.3). □

Lemma 2 (Cheap inserts) *The time complexity of inserting an object into a tree of height bounded by* $(1+\alpha)\log(n^*)$ *without partial rebuilding is bounded by* $s^k(1+\alpha)\log(n^*)$, *or* $O(s^k \log(n^*))$.

PROOF. This can be shown because an object can at most intersect with s^k discriminators, so there will be at most s^k different insertion paths down the tree. This makes the cost of insertion $s^k(1+\alpha)\log(n^*)$. □

Lemma 3 *For* n^* *non-identical objects with non-identical MBRs, there exists a perfectly balanced RC-tree of height* $\log(n^*)$.

PROOF. Consider the worst case of n^* MBRs that share the top, left and bottom sides and are different by only the right side. Then there are at least $n^* - 1$ of these right sides which can be used as discriminators to distinguish these MBRs. One can build a perfectly balanced binary tree of height $\log(n^*)$ with these discriminators as intermediate nodes. □

We are now ready to state the important theorems on insertion and search cost of RC-tree. We sketch the proofs as follows.

Theorem 1 *In a k-dimensional balanced RC-tree for indexing MBRs, the time complexity of inserting into a RC-tree with* n^* *original data objects is* $O(s^k \log(n^*))$ *where s is the spacing factor and k is the number of dimensions.*

PROOF. We define weight difference of RC-tree at a node v, $\delta(v)$ as,

$$\delta(v) = Max(0, |v.left| - |v.right| - 1) \tag{5.4}$$

A perfectly balanced RC-tree T has weight difference $\delta(T) = 0$.

From [And99], for a binary tree, if $h(T) > (1+\alpha)\log(|T|)$, and let v be the lowest node on T's longest path such that $h(v) > (1+\alpha)\log(|v|)$, then

$$\delta(v) > (2^{1-1/(1-\alpha)} - 1)|v| - 1. \tag{5.5}$$

In other words, $\delta(v)$ has changed from 0 to $(2^{1-1/(1-\alpha)} - 1)|v| - 1$ between two partial rebuilds. Since every cheap insertion creates at most s^k more objects, $\delta(v)$ is also changed by at most s^k by each insert. Hence the number of cheap inserts that take place between two consecutive rebuilds is

$$\frac{(2^{1-1/(1-\alpha)} - 1)|v| - 1}{s^k} = O(|v|/s^k).$$

Let v be the root of T with n nodes and n^* original objects, then the number of cheap inserts between two consecutive rebuilds in T is $O(n/s^k)$ or $O(n^*)$.

Now let us bound the cost of a partial rebuild at node v. The rebuilding algorithm involves flattening the tree rooted at v, and the reconstruction of a perfectly balanced RC-tree by recursively partitioning the objects. The flattening of the tree takes $O(N)$ or $O(s^k n^*)$ time, due to Lemma 1. The recursive packing is bounded by $s^k n^* \log(n^*)$. Therefore the entire rebuilding costs

$O(n^*(s^k \log(n^*) + s^k))$.

The amortized insertion cost is thus:

$$\begin{aligned} t_{ins}(n^*) &= \frac{\text{rebuilding time} + \text{time to do cheap inserts}}{\text{num of cheap inserts}} \\ &= O\left(\frac{n^*(s^k \log(n^*) + s^k) + n^* s^k \log(n^*)}{n^*}\right) \\ &= O(s^k \log(n^*)) \end{aligned}$$

Where k is fixed (in the case of 2-d or 3-d spatial objects) and s is a constant, we get an amortized insertion cost of $O(\log(n^*))$. □

Theorem 2 *A point-query search in a k-dimensional balanced RC-tree is expected to take* $O((1.5)^k(\log(n^*) + s^k))$. *The worst case time is* $O(2^k n^*)$.

PROOF. In a point-query search, the worst case is when the the value of the point in each

dimension is on the hyper-plane of some discriminator. Thus a query point of k dimension coincides with at most k discriminators in any RC-tree. This is because no two identical discriminators exist in any path of the RC-tree. Thus there can be at most 2^k different search paths in any search. To get the expected number of search paths, $E(N_{paths})$, we sum up the total number of paths in all possibilities and divide it by the number of cases.

$$\begin{aligned} E(N_{paths}) &= \frac{\binom{k}{0}2^k + \binom{k}{1}2^{k-1} + \cdots + \binom{k}{k}2^0}{\binom{k}{0} + \binom{k}{1} + \cdots + \binom{k}{k}} \\ &= \frac{(1+2)^k}{2^k} \\ &= (1.5)^k \end{aligned}$$

Given the bounded height of $(1+\alpha)\log(n^*)$, and the bound on N and number of leafs $(n+1)/2$ in Lemma 1, the average number of objects in each leaf is s^k. Therefore the average path length for a search is $(1+\alpha)\log(n^*) + s^k$. The worst case path length is $O(n^*)$, when all objects are overlapping and of similar sizes, so there is no discriminator to distinguish them. □

The following theorem on the space complexity is easily shown by acknowledging the fact that any object can be divided at most s^k times in any RC-tree, so the total number of nodes is $(n+N) \leq 3s^k n^* - 1$.

Theorem 3 *A k-dimensional balanced RC-tree has a worst-case space usage* $O(s^k n^*)$.

With space control, the space could be effectively less than this bound.

5.6 Empirics

We now compare several RC-tree variants (different leaf capacitys and with/without space control) experimentally with several R-tree variants and PMR-quadtree on a number of datasets. The experiments were done on a Pentium 4 2.4GHz with 512M of RAM running Linux 2.4.20. Our RC-tree implementation is in the generalized constraint based style as described in section 5.4 and written in C++.

We have investigated the performance of the two partitioning methods for RC-tree introduced in section 5.4. We noted that while RC-MID performs dramatically better in insertion cost for most of the datasets, the space requirement can be much higher. Some benchmark

problems require too many clippings and the tree grows too fast if there were no space control. Therefore, we will use RC-SWEEP which is more consistent and reliable in the rest of this section.

The R-tree and R*-tree are the C++ implementation by Marios Hadjieleftheriou which supports a main memory index. The R$^+$-tree is adapted from the C implementation by Timos Sellis' group. The R-tree variants implementations can be found at the R-Tree Portal [R-T]. For the R-tree family, we have used fanout sizes ranging from 3 to 50 for the index nodes. Although the RC-tree is a binary search tree, it does not make sense to compare R-trees with a node size of 2 as the tree is no longer balanced. Indeed, search becomes very expensive with binary R-trees. R*(3) is excluded from all experiments because the implementation works only for fanout 4 and above.

The PMR-quadtree is a C implementation based on the algorithm in [Sam90] with fanout 4 and a bucket size 20 and 50. PMR-quadtree with smaller bucket sizes cannot handle some of our datasets due to memory exhaustion. We control the height and the space of PMR-quadtree by limiting the minimum quadrant size to 0.1×0.1.

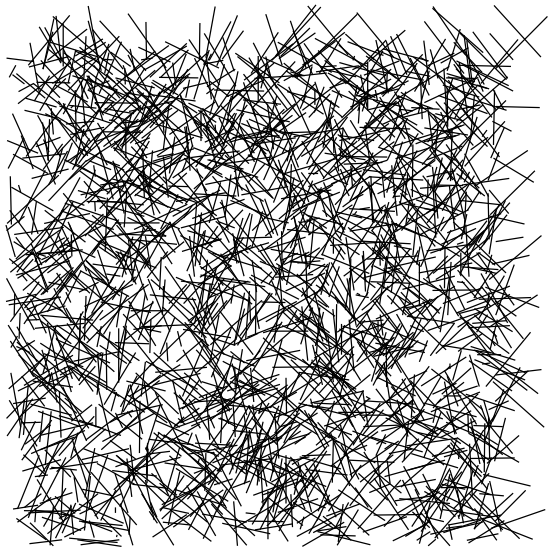


Figure 5.3: Schematic of rand (lines)

Given that we have various implementations in different languages, the primary measure of cost in searches or updates to the database is number of discriminators or MBRs compared/tested. Each comparison/testing is assume to be of unit cost. We denote this measure of search/update cost uniformly as *accesses*.

In the graphs below, $RC(x)$ denotes an RC-tree with leaf capacity x and no space control,

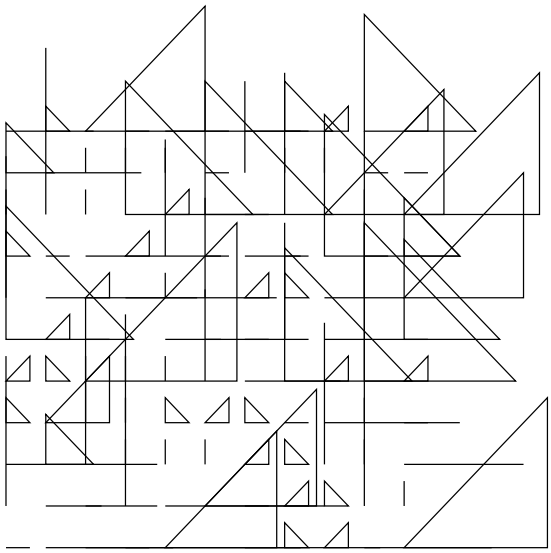


Figure 5.4: Schematic of grid (triangles)

and $RC(SC)$ denotes an RC-tree with leaf capacity 1 and space control $\gamma = 1.3$. In all RC-trees, balancing factor $\alpha = 0.9$. $R(x)$ denotes an R-tree using quadratic splitting with a node size of x. Quadratic splitting is used because the search performance is generally better than linear splitting without sacrificing much on insert cost [BKSS90]. We remark that the node size is important as it affects space usage and performance. With an in-memory structure, it does not make sense to use large nodes because the cost of processing a node is linear in the node size. There are also cache effects which are discussed in section 5.7. PMR-quadtree is denoted as $QT(x)$, where x is the bucket size. We have used *random point queries* in all these experiments.

Space usage is compared by counting the total number of nodes multiplied by the number of floating numbers needed to represent the discriminator/MBR in each node. The sizes for pointers are ignored here and thus underestimates the space usage for the R-tree family when the node size increases.

5.6.1 Synthetic overlapping objects

In the first part of the experiment, we compare the search/insertion/space performance of RC-tree with other indexes on synthetic datasets consisting of line segments or triangles. Lines and triangles are chosen as they are about the simplest geometric shapes in real-life applications. They also give rise to domain reduction when clipped. The datasets are as follows:

- rand(0/1): uniformly distributed objects of variable sizes, see figure 5.3.

Figure 5.5: German Roads (rectangles)

- clust(5/10): variable-size objects around 5 or 10 clusters
- grid(2/3): variable-size objects placed systematically in a grid pattern, see figure 5.4.

The number after rand and grid denotes the degree overlapping, e.g. there is more overlapping in rand1 than rand0. Figures 5.3 and 5.4 only illustrate the patterns of the datasets, and the actual layouts of the objects in these classes of datasets are much denser: each problem has 50,000 objects of sizes ranging from 100 to 500, populating an area of 10000×10000.

We expect the insertion cost of RC-tree to be higher than other R-tree variants, primarily due to partial rebuilding, clipping, and insertion into multiple paths in the RC-tree. We observe from figure 5.6 that, in general, increasing leaf capacity reduces the insertion cost for both RC-tree and PMR-quadtree, though the same does not hold for R-tree family. Space control also plays a part in controlling the insertion cost. In cases like rand0-tri and grid2-tri, these measures have brought down the insertion cost of RC-tree to being comparable and even better than its peers. The insertion cost is lowest for R-tree and R*-tree when the fanout is around 10, and it mostly increases monotonically for R^+-tree as fanout goes up.

In any data base, queries always happens more frequently than updates, even for dynamic abstract regions that RC-tree is designed for. Thus we believe query efficiency is more important than update efficiency. From figure 5.7, the search performance of RC-tree is uniformly much better than all competitors, in many cases by more than 10 times. The insertion cost savings

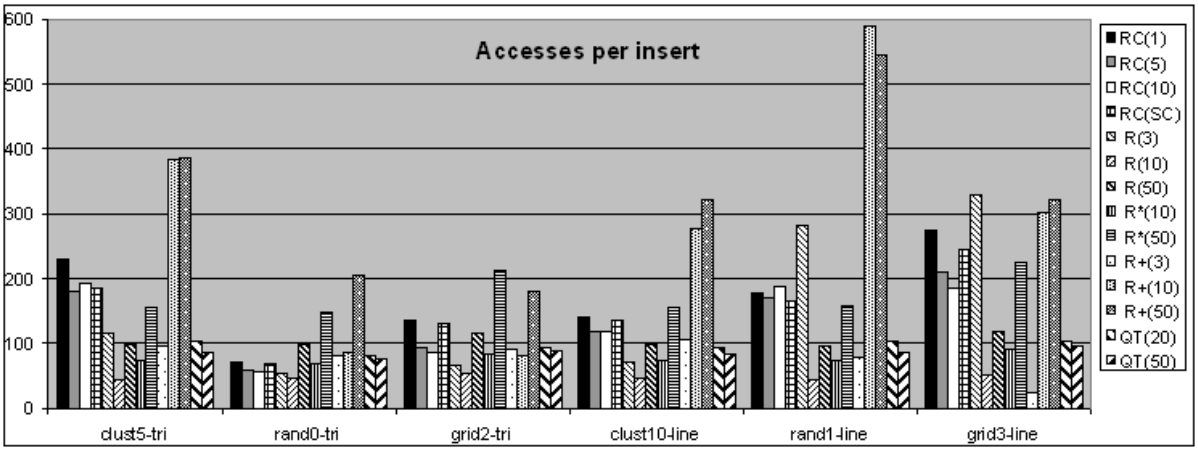


Figure 5.6: Insertion Cost for Synthetic Data

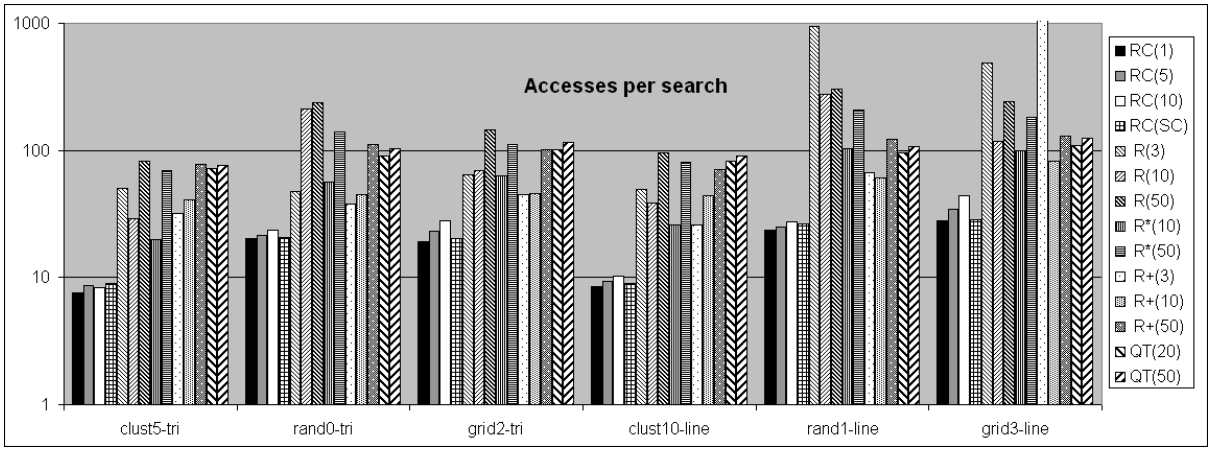


Figure 5.7: Search Cost for Synthetic Data (Log Scale)

from increased leaf capacity come with a price in terms of search cost for almost all methods, as we observe the general trend that larger nodes means more expensive search. In general, the search cost of R-trees increases significantly as fanout increases. In the case of rand1 and grid3 which have high MBR overlapping, the advantage of clipping-based methods such as RC and QT over the non-clipping methods is obvious. This suggests that space partitioning is a more suitable technique when it comes to indexing overlapping data. Quadtree has average search performance usually better than R and R*, but similar to R^+-tree. This is due to the clipping nature of both structures. However, the unbalanced nature of quadtree offsets some of that advantage of clipping.

With lines and triangles, RC-tree stores the actual object shapes rather than their MBRs. Due to the heavy overlapping, RC-tree tends to clip more in order to best discriminate the

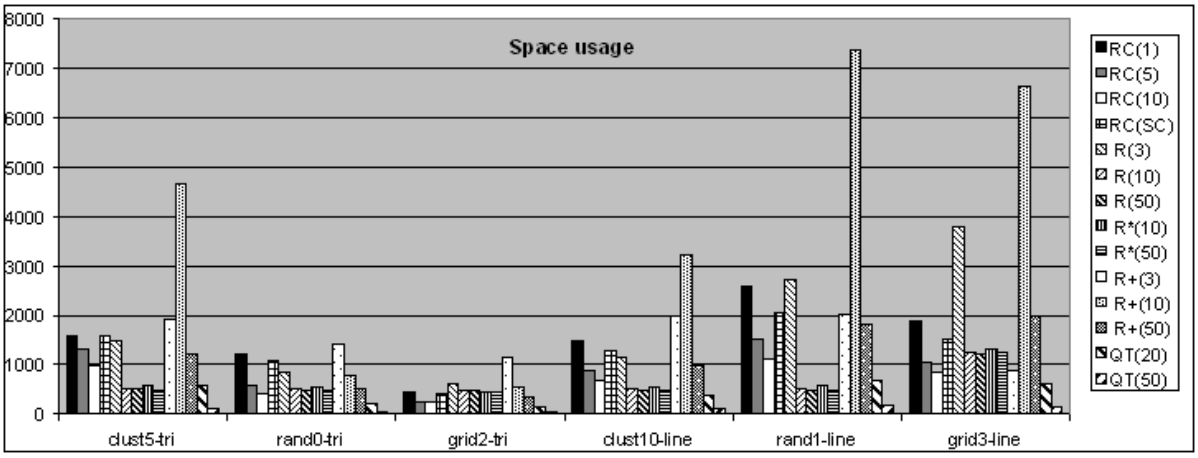


Figure 5.8: Space Usage for Synthetic Data ('000 numbers)

objects. Space efficiency is, however, generally better than that of R^+-tree (see figure 5.8). This is because RC-tree's domain reduction strategy reduces the size of the MBRs of clipped objects on the fly and hence reduce tree size (figure 5.2), whereas storing the MBRs rather than the objects themselves in R^+-tree does not help. The space control and increased leaf capacity in RC-tree help reducing the space significantly to be competitive against all other methods. One also note that increase in bucket size reduces the space of quadtree greatly too, without sacrificing the search performance too much. Large bucket size causes less clipping and hence better space efficiency. And because most of the data used here are quite uniformly distributed, the quadtrees with large bucket sizes are shorter and more balanced. The linear search within a large bucket offsets the benefit of a smaller (and more balanced) tree to give little changed search performance.

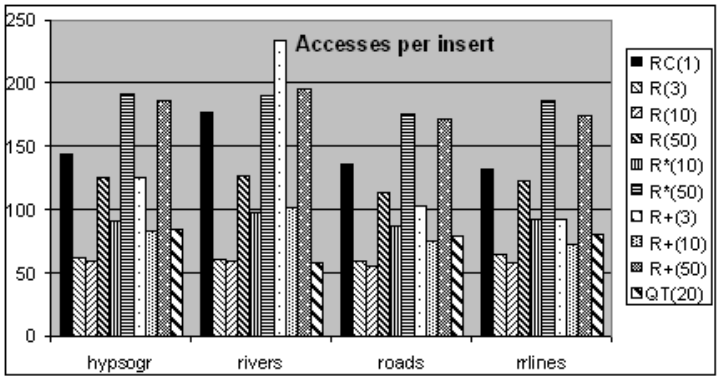


Figure 5.9: Insertion Cost on GIS Datasets

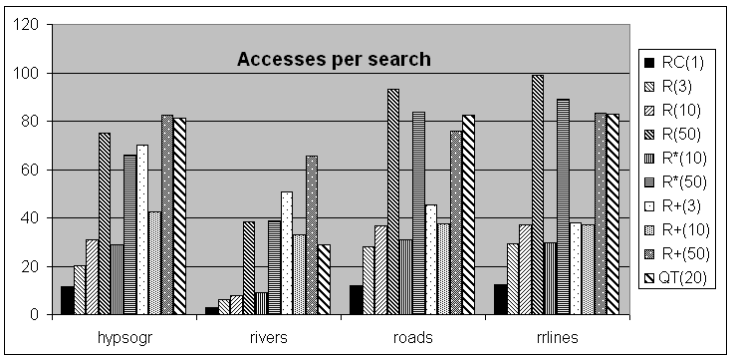


Figure 5.10: Search Cost on GIS Datasets

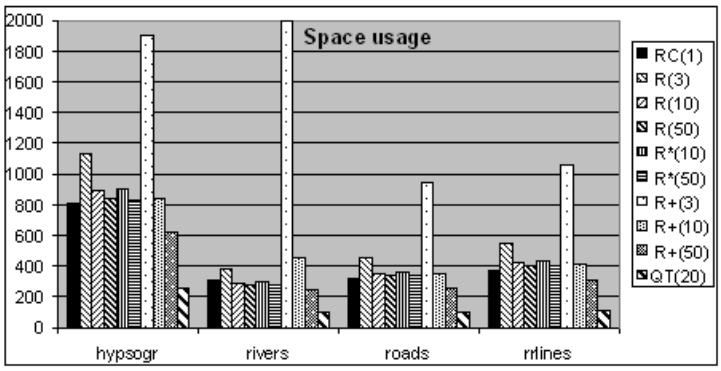


Figure 5.11: Space Usage: GIS Datasets ('000 numbers)

5.6.2 Non-overlapping rectangle datasets

In the second part of the experiment, we compare the performance of RC(1) with other indexes on the following traditional GIS datasets, which are made up of small *pre-segmented rectangles* that have little or no overlapping (the number of rectangles are indicated in brackets): roads (30674) (figure 5.5), rrlines (36334), rivers (24650) and hypsogr (76999). These are real life datasets also obtained from [R-T].

From figures 5.9, 5.10 and 5.11, we are surprised to find that, with moderately high insertion cost, comparable space usage, but much better search cost, RC(1) again puts up a strong competition, even though RC-tree is not designed for indexing rectangles. The space usage of RC-tree is small for rectangle data largely because the partial rebalancing technique has helped maintain the tree in more balanced shape than its counterparts. We can see from figure 5.11 that R^+-trees costs too much space with small node sizes, intuitively one can expect that R^+-tree causes more MBR clipping with smaller fanout nodes.

Finally, to get an idea of how RC-tree fares in terms of real time, we compare the wall clock

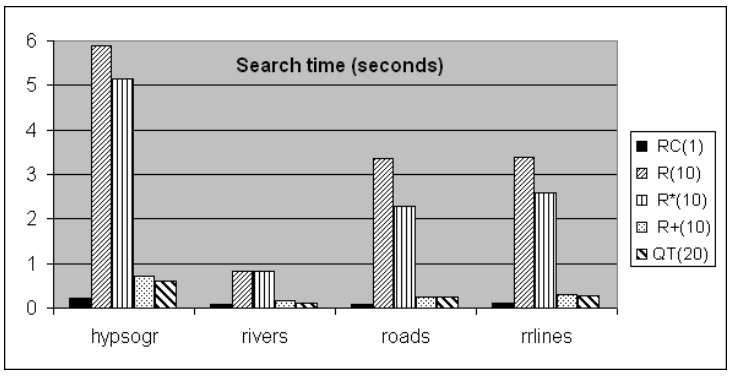


Figure 5.12: Search Cost on GIS datasets (wall clock time)

query processing time of RC(1) with the best performing variants in all other methods on the GIS datasets. The number queries for each dataset is equal to the number of objects in that dataset. Figure 5.12 shows the clear advantage of RC-tree over others.

5.6.3 Query accuracy and range queries

This section compares query accuracy among the indexing methods and then compares the difference between dynamic segmentation with static pre-segmentation.

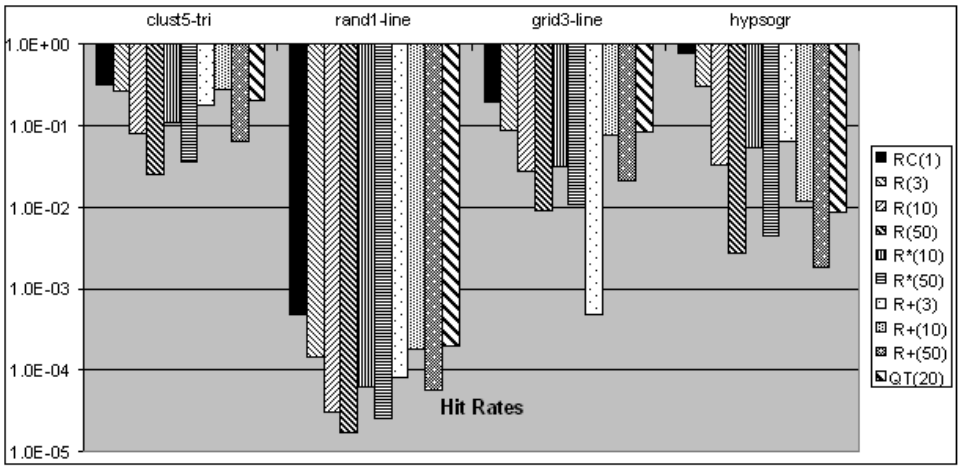


Figure 5.13: Hit Rates on Synthetic and GIS Datasets (Negative Log Scale)

Figure 5.13 compares accuracy across indexing methods where accuracy is defined as

$$\text{hit rate} = \frac{\text{number of hits}}{\text{number hits} + \text{number of misses}}$$

For the GIS datasets, RC-tree is significantly better than other indexes, in some cases by several orders of magnitude. The hypsogr dataset is chosen as representative of the results

because the performance advantage of RC-tree is quite similar across all GIS data. For the synthetic datasets, one would expect that the probability of getting a hit is increased with highly overlapping regions. Here, clust5-tri, which has large areas of overlap, gives a smaller difference in hit rate among the indexes. Nevertheless, RC-tree enjoys the best accuracy across all methods. We conjecture that keeping actual shapes and domain reduction helps reduce dead space and the dynamic decomposition with clipping gives better discrimination. Note that query accuracy is worse in the PMR-quadtree.

Even though the original motivation of RC-tree requires only point queries, our algorithm handles range queries as well. Table 5.1 records our investigation of range query performance among the various indexes. "Size" in table 5.1 refers to query window size. We tested range queries to two datasets: roads and rand0-tri, and we compare RC(1) with the best performing competitors, namely R-tree family with fanout of 10 and quadtree with leaf capacity 20. The first number in every cell of the table is the average search cost, and the second number is the hit rate. Table 5.1 clearly demonstrates the advantage of RC-tree in these two dimensions for range queries.

5.6.4 Dynamic vs. static segmentation

To demonstrate the advantage of dynamic segmentation in RC-tree as opposed to the static pre-segmentation used in MBR-based indexes such as the R-tree family, we conducted the following experiment. We index in RC-tree 5000 randomly positioned line segments of the same length (500) in a 10000×10000 area. We record the number of misses given by the RC-tree with 5000 random point queries, which is 17327. Then for every other index, we segment the given lines by cutting the line segments into smaller equal pieces and index the MBRs of the segmented pieces, such that the number misses is as close to the RC-tree as possible. In other words, we want to study the amount of segmentation needed and the associated costs of various indexes when a certain query accuracy is demanded by the application. Table 5.2 records such results. The second column under "Seg." in the table records the average number of pieces each object in the original dataset is segmented into. For example, for $R^+(5)$ to achieve misses of 17985, the original data have to be cut into 10 pieces each. The 4^{th}, 5^{th} and last columns record search, insertion and space costs. It is obvious from table 5.2 that other indexes incur more space and search cost and, in some cases insertion cost as well, to have the same query accuracy as RC-tree. For $R^+(10)$, we were not able to achieve the same number of misses without exhausting

roads						
Size	0	20	40	60	80	100
RC	12.5	15.5	19.7	25. 0	31.6	39.4
(1)	62.9%	86.5%	92.1%	94.3%	95.7%	96.6%
R	36.8	39.8	43.3	47.1	51.4	56
(10)	2.9%	11.6%	20.7%	28.3%	34.8%	40.4%
R*	31	33.6	36.7	40.2	44.1	48.5
(10)	3.6%	13.9%	24.1%	32.2%	39.2%	44.9%
R^+	37.4	45.5	54.4	63.6	73.6	84.2
(10)	1.3%	5.65%	10.1%	14.0%	17.4%	20.4%
QT	82.6	86.1	90.2	94.7	99.8	105.4
(20)	0.9%	5.2%	11.1%	17%	22.6%	27.8%
rand0-tri						
Size	0	20	40	60	80	100
RC	21.1	29.1	39.4	52.0	66.5	83.7
(1)	14.8%	39.5%	53.3%	62.3%	68.5%	73%
R	162.2	170.9	179.9	189.4	199.4	209.8
(10)	1.8%	4.6%	7.5%	10.4%	13.3%	16.1%
R*	62.0	67.2	72.9	79.0	85.6	92.6
(10)	4.9%	11.3%	17.1%	22.4%	27.2%	31.5%
R^+	44.7	51.4	59.1	68.0	77.8	88.8
(10)	6.8%	15.7%	22.8%	28.3%	32.7%	36.2%
QT	90.4	96.6	103.8	111.9	121	131
(20)	5.9%	15.5%	24.3%	31.84%	38.1%	43.4%

Table 5.1: Range Queries (Accesses/Hit Rates)

memory.

5.6.5 Insertion and search cost vs. data size

To verify the analytical results about insertion cost and search cost in section 5.5, we pick two datasets, rrlines and grid5, whose spacing factors are roughly constant against n^*, and run insertion and search algorithms of RC-tree on the first 4000, 8000, 12000 objects and so on. We record the increasing n^* and the corresponding insertion cost and search cost (in terms of the number of comparisons made). Figure 5.14 indicates that there is almost a linear correlation between the insertion cost and $\log(n^*)$, which affirms that insertion is logarithmic cost under these special cases of constant spacing factors. Figure 5.15 shows that as n^* goes large in rrlines, the search cost is no longer proportional to $\log(n^*)$, but sub-linear to it. A careful check reveals the anomaly is because of the much empty space in the dataset. This causes many queries to return negative, especially when n^* is small. Therefore when n^* is small, the search cost is under-estimated. Figure 5.14 and 5.15 also show results from RC-tree with space control.

	Seg.	Misses	Search	Insert	Space
RC	1	17327	18.32	110.0	185.9K
R(3)	4.2	18483	58.5	53.4	399.5K
R(5)	16.7	18233	85.9	46.8	1.03M
R(10)	62.5	20189	164.3	62.1	3.6M
R*(5)	16.7	16391	199.1	216.4	1.2M
R*(10)	62.5	16752	71.1	91.9	3.6M
R^+(3)	3.3	16606	35.7	100.6	733.2K
R^+(5)	10	17985	34.55	66.82	832.7K
R^+(10)	100	33310	49.82	74.25	4.58M
QT(20)	100	20860	63.5	61.9	3.28M

Table 5.2: Dynamic Segmentation vs. Static Segmentation

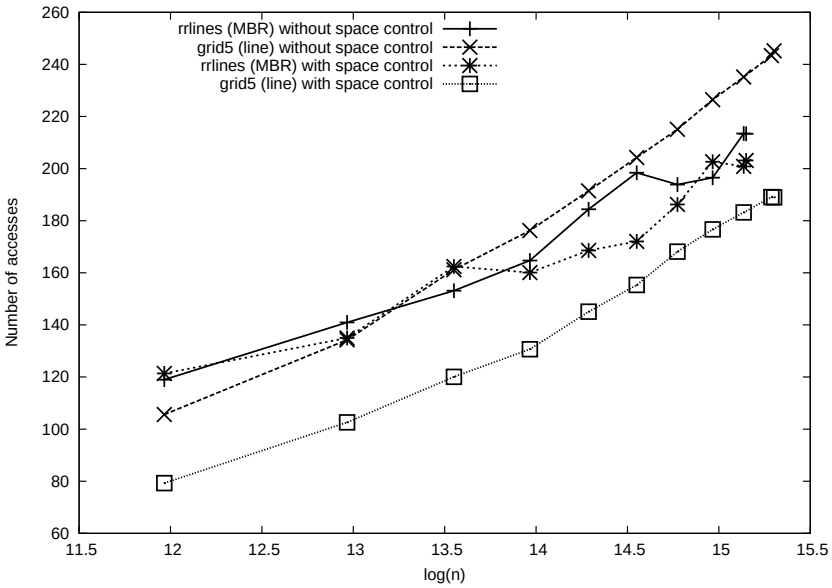


Figure 5.14: Insertion cost vs. $\log(n^*)$

Space control improves insert efficiency because less splitting and clipping is done, but the flip side is that more objects are stored in the overflow nodes, and hence they have to be searched in linear order. As a result the search time is increased.

5.7 Toward Cache-Oblivious RC-Tree

R-tree, R^+-tree and other B-tree descendants were designed to be I/O efficient due to the access speed difference between the main memory and the secondary storage, i.e. hard disk. Similar difference exists between the CPU cache and the main memory. Therefore, the issues of cache effects need to be considered for an in-memory index structure. This section investigates the

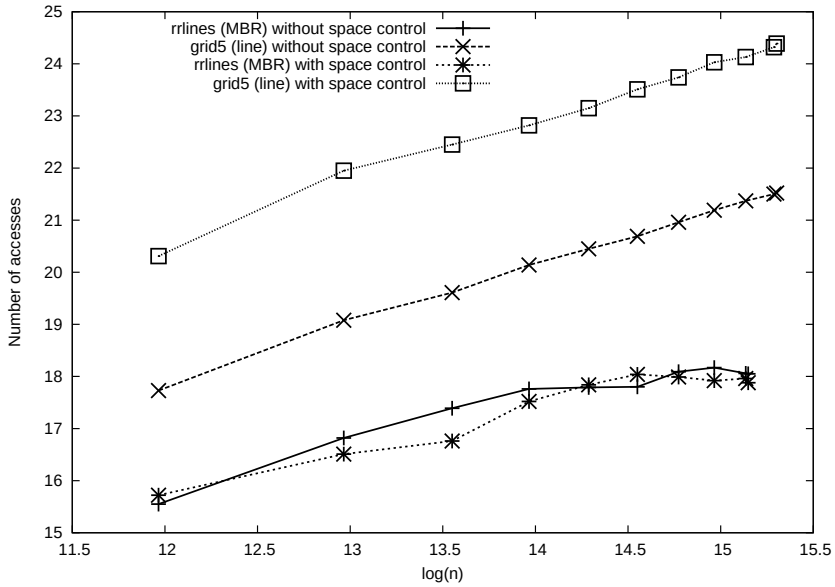


Figure 5.15: Search cost vs. $\log(n^*)$

application of cache-obliviousness to RC-tree.

Cache-oblivious algorithms are derived from the *ideal-cache model* [FLPR99], which reasons about and makes use of two-level memory hierarchy, i.e. the cache and memory, without explicit knowledge of block size B of the memory, and the cache size M. Our cache-oblivious RC-tree is based on the cache-oblivious B-tree by Bender et al.[BDFC00].

We made some minor modification to RC-tree and its algorithms to make it cache-oblivious. First, we follow the dynamic insertion algorithm of [BDFC00] such that the tree nodes in RC-tree form the *van Emde Boas layout* [BDFC00], and the smallest recursive subtree is of height two, i.e. it consists of three nodes. By adjusting balancing factor α, we could arrive at a weight-balanced tree with arbitrary compactness (at the expense of insertion time of course). To the extreme of the perfect weight-balance, the nodes are perfectly lined up in the memory with no "holes" in between at all. Then we change the implementation of the overflow nodes in the leaves from a linked list of indexable objects to a linked list of *chunk* of objects, where each chunk is allocated in contiguous memory as an array. Each leaf node of the RC-tree maintains a point to the beginning of a chunk-list.

We will focus on the search performance of cache-oblivious RC-tree. In our experiments, we compared the search time between the cache-oblivious RC-tree and the RC-tree without such arrangements. The GIS datasets are used for this purpose. Data, including rectangle descriptions and query points are first loaded into the memory before insertion and search

tests. The tree node size is 28 bytes while the L2 cache line size on the experiment PC is 128 bytes, good enough to accommodate the smallest recursive tree of 3 nodes. With $\alpha = 0.5$ and 1,000,000 queries for each data set, cache-oblivious RC-tree on average enjoys up to 20% speed up against its non-cache-oblivious counterparts. However as α decreases, the benefit of the van Emde Boas layout arrangement also diminishes. For example, at $\alpha = 0.2$, the speedup is down to 3-5%.

Thus we see that there is a cache effect but it can be small. We attribute the effect to the following two factors. First, the re-balance techniques used in the RC-tree involves splitting the tree node in a top-down, depth-first manner. As a result, after rebalancing, nodes do get some spatial locality. So in a sense, the basic algorithm is already cache friendly. RC-trees with lower α are more compact due to more re-balance operations, hence applying cache-obliviousness has a correspondingly smaller effect.

Second, the relatively small size of the cache line (which only holds 2 layers of tree nodes) limits the advantage of the van Emde Boas layout. We conjecture, a bigger cache line would probably make a bigger difference.

We conclude that a cache-oblivious version of the RC-tree does have an improvement. It would only be useful if the cache line size is large compared with the node size.

5.8 Extending RC-tree for Set-based Attributes

As we have mentioned in the beginning of this chapter, the purpose of the RC-tree is to efficiently index sustain conditions which can be viewed as multi-dimensional spatial objects. So far, we have assumed that these conditions are conjunctions of linear constraints on some numerical or *range-based* attributes such as stock prices, volumes and distances. In practice, symbolic or *set-based* attributes also exists in many applications. For example, attributes such as colors, names, categories cannot be described by linear constraints. Instead, they are usually constrained by the membership of a set or subset, such as, "*the color of the wall being yellow, green or blue*", which is a member of a subset of colors. It is possible for a sustain condition to contain such attributes, if they exists in the base predicates of the store. In this section, we give a brief discussion on the extension of RC-tree to index objects with set-based attributes.

For symbolic or set-based attributes, one rarely a constraint across different attributes unless some partial ordering exist among these attributes. And if the partial order does exist, in many

cases, the attributes can then be transformed into numerical forms and treated as numerical attributes. For example, if a relation `p(Color1, Color2)` has two attributes both being color. And if one wants to say $Color1$ being darker than $Color2$, then there is usually a way to quantify the "darkness" of colors and hence replace the symbols for colors by their corresponding numerical darkness values, so that now one can say $Color1 > Color2$. Therefore, for the purpose of this thesis, we only consider unary constraints on set-based attributes which are of set membership type, that is, for an a value X of an attribute A, $X \in S$, where S is a subset of values of A.

We solve the indexing of set-based attributes by coding the subsets in the constraints as a binary string. For example, if attribute color has at most 256 different values, we will reserve a string of binary numbers whose length is 256. Every possible color has a predefined position in this string. A subset of colors will but the string with corresponding positions set to 1 and other positions set to 0. Now we can index multi-dimensional objects with mixed attribute types, where some of their attributes are set-based.

As RC-tree is a clipping-based index, same principle applies to set attributes. Discriminators in those set-based dimensions are also sets themselves represented by binary strings of the same length. Let c be a set-based constraint of the i^{th} attribute of an indexable object o, and $c = \{x | x \in s\}$. Let d be the discriminator set of the i^{th} attribute. Then $s \cap d$ goes to the left subtree of d and $s \cap \overline{d}$ goes to the right. Where $s \not\subseteq d$ or $s \not\subseteq \overline{d}$, the object is clipped and inserted into both subtrees. This technique is also found in the RD-tree of the GiST framework [HNP95].

We use a heuristic similar to RC-MID for the selection of set-based discriminators. Because the domain of set attributes are usually not large, the heuristic is that we always try to partition the objects by the set-based dimension first. And we always cut the existing set space S into exactly halves (i.e. $|d| = |S|/2$) and in such a way the the number of objects in the left partition is as close as possible as the ones in the right. That partition, which is a set, is the new discriminator. When such discriminator cannot be found, we will attempt other range-based dimensions like in RC-SWEEP.

5.9 Conclusion

In this chapter, we have demonstrated that the RC-tree is a new clipping-based spatial index which can give very good search performance in main memory when compared with a number of common spatial indexes. The RC-tree is particularly good when the spatial objects have significant overlap. It is also competitive for traditional non-overlapping scenarios. Furthermore, the RC-tree allows control over the time and space tradeoff. One can tune for more space and less time, or vice versa. The key to the success of the RC-tree is that it combines dynamic segmentation, domain reduction and partial rebuilding with tunable heuristics.

The work in this chapter, we believe, breathes new life into indexing techniques based on space partitioning and object clipping. Due to concerns of space costs, such techniques have received little attention since R^+-tree. Our results demonstrate this space problem does exist in R^+-tree. However, with RC-tree, we show that a space partitioning index can enjoy superior query performance when indexing overlapping objects and at the same time have competitive space efficiency. The use of dynamic segmentation in fact further reduces the space requirements by keeping the number of objects indexed small.

The RC-tree will be the critical underlying index structure used for triggering reactors in the OCP system. The next chapter will introduce how this index structure is incorporated in the trigger framework.

Chapter 6

Trigger Framework

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, given large number of blocked proceses, the problem of determining just which processes are to be triggered by an often-occurring event is intractable.

As we have mentioned in section 3.4.2, in practical implementation of an OCP system, an indexing mechanism is needed to realize the trigger table depicted in the model. Since the store of OCP is a CLP program and the blocking conditions of reactors are written as CLP goals, the problem of triggering reactors in general is now narrowed down to triggering CLP conditions. In this chapter [1], we discuss the triggering problem for OCP, and present a methodology for dealing with it.

6.1 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 7 (View) *A* view *is simply a rule defining a distinguished set of non-base predicates. It has the general form:*

[1] Content of this chapter has been published in [JYZ05]

$$p(\tilde{X}_0) :\text{-} q_1(\tilde{X}_1),\ q_2(\tilde{X}_2),\ \ldots,\ q_n(\tilde{X}_n),\ \Psi(\tilde{X}_0, \cdots, \tilde{X}_n). \tag{6.1}$$

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 8 (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 9 (Induced View) *Let p be a view of the form $p(\tilde{X}) :\text{-} Body$. Let c be a blocking condition $p(\tilde{X}),\ \Psi(\tilde{X})$. The view of p* induced *by c is the rule*

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

To determine if a blocking condition c on a view p is enabled by an update δ is in general an undecidable problem (CSP). 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.

Direct execution of the induced views is, unfortunately, not very practical, because c can be 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 with the help of the RC-tree introduced in chapter 5. We then show how to reduce composite views to basic ones so that the RC-tree can be used.

6.2 Basic views

As we know, conjunctions of constraints are essentially geometrical shapes. For example, the following constraints:

$$2x + 3y - 6 \leq 0 \wedge 0 \leq x \leq 3 \wedge 0 \leq y \leq 2$$

can be represented by a right triangle anchored at the origin of the Cartesian system as shown in figure 6.1.

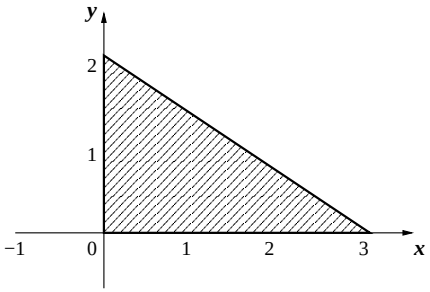


Figure 6.1: A Triangle Shape

Let $\Psi(\tilde{X})$ be a constraint and q a base predicate. Then the first type of basic condition $c ::= q(\tilde{X}), \Psi(\tilde{X})$ can treated as a geometrical shape. Accordingly, to find out if an update δ on $q(\tilde{X})$ where $\tilde{X}$ has been grounded enables c is equivalent to checking of the point $\tilde{X}$ lies within the shape representing c. As such, if we represent many basic conditions by their "shapes" and index them in a spatial index structure, then we can trigger these conditions by querying the index with *point queries* formulated by the update δ.

There is a wealth of publications on indexing multi-dimensional spatial objects (e.g. in [Gae98] and [Sam90]) which may be used for this purpose. However, none of the existing indexes have been reported to handle objects with large extent and substantial overlapping

well, and the setting of the OCP system calls for indexing of objects of this nature. Therefore, we have chosen to use RC-tree which is better suited for this purpose.

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 $c ::= p(\tilde{X}_0), \Psi(\tilde{X}_0)$, 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 section 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.

6.3 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 6.2.

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 is 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:

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

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

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

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 Example 4. 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. 6.2 shows the simplified definition of `deliverable`.

By inspecting the third rule of `deliverable`, one can see that `deliverable` is the transitive

```
deliverable(A, B, Budget):-
  0 < Cost <= Budget,
  vessel(A, B, Cost).

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

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

Figure 6.2: Simplified `deliverable`

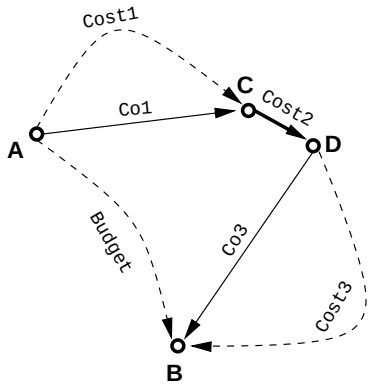


Figure 6.3: Abstraction of `deliverable`

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 closures. 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. 6.2. We illustrate this scenario in Fig. 6.3, 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 end this section with a summary of the characteristics of abstraction:

- it reduces composite views to basic views so that direct indexing can be done on the basic conditions;
- an abstraction is a *conservative* estimation of the original view, which means the set of all predicates satisfying the abstraction is a *superset* of the set of predicates that satisfy the original view, therefore, an update that does not trigger an abstracted condition definitely does not trigger the original blocking condition. In other words, exclusion of reactors by abstraction is *safe*;
- verification of the application-based abstraction against the original view is easier and often feasible;
- the technique of abstraction is often domain specific.

6.4 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 Example 4 for transporting cargos 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 [Fly].

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 6.3 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 6.1. 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 6.1: Hit Rate and Average Trigger Time

From table 6.1, 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.

6.5 Conclusion

This chapter introduces a framework for triggering complex reactor conditions which are blocked. The framework is used to implement the trigger model in section 3.4.2. The framework concerns the use of one or more indexes (RC-trees) to store conditions which are pre-processed into basic views. Complex conditions can be reduced to basic views through abstraction. We give a concrete example of how abstraction is done. The methodology used here for abstraction can be readily generalized for other scenarios.

Chapter 7

Implementation Techniques of GCC

The speculation in GCC means that the choices in one program are multiplied with choices from all other programs. This means that when choices do not or cannot commit for a long time, the multi-world can grow exponentially large. We remark that this is not a drawback but rather a consequence of allowing the power which comes from committing "late" and the ability to speculate. Furthermore, the amount of parallelism or concurrency can also become exponentially large since the programs are running in parallel at the leaves.

The main challenge to realize GCC is to enable optimizations which can reduce the space and computation requirements. In this section, we discuss some key implementation ideas which address these issues as follows:

1. re-distribution of data in the multi-worlds
2. re-distribution of program continuations in multi-worlds
3. structure-sharing of portions of the multi-world

7.1 Reducing data storage

Earlier we defined conjunctive views on the multi-world. We now define a *differential view*, $\mathcal{DFV}$, for every node of the multi-world except the root node.

Definition 10 (Differential view) *For a subtree $\mathcal{T}_1 \oplus_{id} \mathcal{T}_2$),*

$$\begin{aligned} \mathcal{DFV}(\mathcal{T}_1) &= \mathcal{CV}(\mathcal{T}_1) \setminus \mathcal{CV}(\mathcal{T}_1 \oplus_{id} \mathcal{T}_2) \\ \mathcal{DFV}(\mathcal{T}_2) &= \mathcal{CV}(\mathcal{T}_2) \setminus \mathcal{CV}(\mathcal{T}_1 \oplus_{id} \mathcal{T}_2) \end{aligned}$$

For special case of the root of the multi-world $\mathcal{T}_0$,

$$\mathcal{DFV}(\mathcal{T}_0) = \mathcal{CV}(\mathcal{T}_0)$$

In our original semantics, the store of each world is kept at the leaves of the multi-world tree. This can lead to a lot of data duplication if only a few variables are relevant to speculation. The idea of the optimization is to re-organize the multi-world so that, instead of having the data at the leaves, portions of the stores can be materialized at other nodes in the tree by using a differential view. In essence, this method stores common data as high as possible in the tree, to reduce storage redundancy. This results in the root node storing the common data of all the worlds; its left (right) child stores the common data of all the world in the left (right) sub-tree minus the common data of the root, and so on.

One can use a strategy that periodically materializes the $\mathcal{DFV}$ at respective nodes and thus save storage space. This can make evaluation of multi-world queries more efficient as common data are stored higher and the top-down queries are more likely to be answered early in the tree traversal.

For efficient evaluation of guard conditions, we can make use of materialized disjunctive view. Guard conditions are substantially more expensive than tests when the guards block. A blocked guard is only enabled by another update either by another program or by an external variable change. The reason why the disjunctive view is useful at a particular node in the multi-world is that a blocked guard may occur in many worlds in the sub-tree from that node. So a disjunctive view can be used as an indexing condition to approximate whether a change in the view might wake up a blocked guard.

However, disjunctive views are large and can be too expensive to materialize and is counterproductive if one is trying to reduce storage redundancy. The idea then is to store an approximation $\mathcal{CP}$ of that view $\mathcal{DV}$, such that $\mathcal{DV} \models \mathcal{CP}$. We call such approximation $\mathcal{CP}$, a *common property* of a multi-world $\mathcal{T}$, written as $\mathcal{CP}(\mathcal{T})$.[1] There are many choices for a $\mathcal{CP}$, and it is probably best left to the implementor of the runtime system. The simplest $\mathcal{CP}$ of a variable with a set of ordered values, is to use an interval. For example, if a multi-world contains the following data about variable x: $x = \{3, \ -2, \ 5, \ 6, \ 9\}$, a possible common property of that tree for x is $-2 \leq x \leq 9$. So if under this $\mathcal{CP}$ a blocked guard was not satisfied previously, then

[1]The common property is similar to Minimum Bounding Rectangle (MBR) which is frequently used in approximating irregular shapes in spatial search trees and indexes.

it would still not be woken up by any updates to the stores for which this approximation is still valid. We remark that CPis used not only for guards but also for other purposes which we will explain in the next section.

7.2 Reducing the number of continuations

There are two sources of concurrency and parallelism in a multi-world. Firstly, the internal concurrency between possibly interacting programs in one world. Secondly, we may have many worlds which run independently of each other. The idea here is to that we can reduce the amount of computation needed by reducing the number of program continuations in the multi-world. To achieve this, we can collect continuations which are identical copies (due to other programs splitting the worlds) from the leaves, treat them as one copy, which we call the *synchronous continuation* and execute it at a higher node in the multi-world tree. In other words, instead of running many copies of the same program on the leaves, we run just one copy in a higher internal node, and thus save computation.

Recall from the last section that we can materialize $\mathcal{DFV}$'s and $\mathcal{CP}$'s in the internal nodes, and the current strategy of "gathering" continuations up works well because it is likely that a synchronous continuation can be moved up to an internal node where the data it operates on are materialized.

Of course, it is not always safe to "synchronize" the continuations from the leaves but there are many important cases when this is possible. For example, suppose there is a program

$$P ::= \texttt{while}(w > 0)\ \texttt{do}\ w := w - 1$$

and variable w is currently not speculated (i.e. takes on a definite value) and will not be updated by any other continuations in the system in future, then executing P at the a node higher in the tree has the same effect as executing multiple instances of P at the leaves of the sub-tree.

If a synchronous continuation σ attached to node $\mathcal{T}$ references variables that are currently materialized at $\mathcal{T}$, then operations can be done at node $\mathcal{T}$. For example, if the operation is an assignment of a variable x, and $(x, v) \in \mathcal{DFV}(\mathcal{T})$, then $\mathcal{DFV}(\mathcal{T})$ is updated directly.

$$\mathcal{DFV}(\mathcal{T}), \sigma \xrightarrow{x:=v'} \mathcal{DFV}(\mathcal{T})[x = v'], \sigma'.$$

If the variable x referred to by σ, is not in $\mathcal{DFV}(\mathcal{T})$, then there are two cases. Let us briefly discuss the actions to be taken in these two cases.

First, if $(x,\ ?)$ [2] does not imply $\mathcal{CP}(\mathcal{T})$, then the variable is materialized higher. One can locate x in the ancestor of $\mathcal{T}$, $\mathcal{T}'$, and redistribute x in the subtree so that node $\mathcal{T}$ now contains $(x,\ v)$, and x is also copied to the highest possible descendent nodes of $\mathcal{T}'$ such that these nodes, together with $\mathcal{T}$, form a *frontier* around $\mathcal{T}'$. Then transition proceeds as usual.

Second, if $(x,\ ?)$ implies $\mathcal{CP}(\mathcal{T})$, it could either be in the ancestors of $\mathcal{T}$ or in the subtree of $\mathcal{T}$, as $\mathcal{CP}$ is only an approximation. We can locate it as in the first case if it is above $\mathcal{T}$. Otherwise, we have to clone the continuation σ because the x is under speculation and is taking on different values in the descendent nodes of $\mathcal{T}$. We will now attach one copy of σ at every node under $\mathcal{T}$ where x can be found in the $\mathcal{DFV}$ of that node, and remove σ from $\mathcal{T}$.

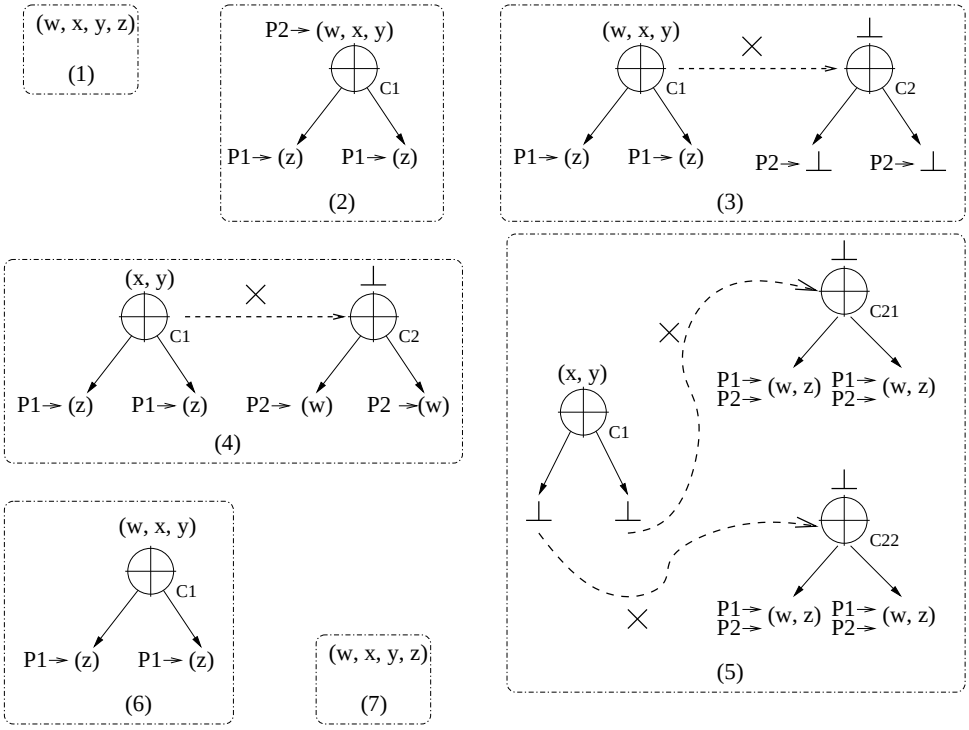


Figure 7.1: Example of Structure Sharing

7.3 Reducing the size of multi-world

In this section, we introduce a strategy called *structure sharing* to contain the problem of compounding choices in the multi-world. The main idea of structure sharing is to keep the

[2] "?" denotes any value.

tree in linear form as much as possible and to maintain only pointers between the structures to indicate that a "cross product" relationship exists between them. We can linearize two subtrees, so long as the Σ's in one tree are data independent from the other. Data can to be moved from subtree to subtree to maintain the data independence.

We now demonstrate the strategy in figure 7.1 with the following example. Let us assume $\mathcal{DV}(\mathcal{T})$ contains four variables, $\{w, x, y, z\}$. Consider the following two programs.

$P1 ::= ((0 < z \leq 10) \Rightarrow z := z + 1) \quad \oplus \quad ((10 < z \leq 20) \Rightarrow z := z - 1)$

$P2 ::= (w > 0 \Rightarrow w := 1; z > 0 \Rightarrow z := 1) \quad \oplus \quad (w < 0 \Rightarrow w := 0)$

We represent the $\mathcal{DFV}$ by the names of the set of variables it contains in parentheses, and $\perp$ if the $\mathcal{DFV}$ is empty. P1$\rightarrow$ and P2$\rightarrow$ denote continuations from programs P1 and P2, and where they are attached to in the tree currently. Choice nodes are marked by unique ids. Diagrams (1) through (7) animate the steps of the multi-world transforming under structure sharing.

In the beginning of Fig. 7.1-(1) there is one store with four variables (w, x, y, z) and continuations from programs P1 and P2. In Fig. 7.1-(2), P1 starts first and makes a choice on z. P1 has to split to two copies as the continuation will be different in general. Since the first operation of P1 in both branches only requires z, only z is duplicated in two leaves and other variables are materialized in the $\mathcal{DFV}$ in the root. P2 executes next.

In Fig. 7.1-(3), P2 is issuing a new choice construct and its continuation is split in two. But instead of putting P2's choices on every leaf of node C_1, we construct a new subtree C_2 and put it side-by-side to C_1. Meanwhile, we create a pointer marked by "$\times$" from C_1 to C_2, to indicate that subtree C_2 is being "structure shared" by the leaf nodes of C_1. The semantics remains that C_2 is a *subordinate tree* on all leaves of C_1. And C_1 is said to be the *superior tree* of C_2. We write $\mathcal{T} \rightsquigarrow \mathcal{T}'$ to mean that $\mathcal{T}'$ is a subordinate of $\mathcal{T}$. If node C_1 already has a subordinate tree, we follow through the pointers to the terminal node, and create new pointer to C_2 from the terminal node. Thus any node in the tree has at most one subordinate tree.

In Fig. 7.1-(4), one branch of P2 requires variable w which is not with C_2. We trace back the pointer and look for w in the superior tree. In this case, w lives in node C_1 itself. We transfer w to C_2 and then the strategy in section 7.2 can be applied. In general, if w has n different values and lives in n nodes in C_1 (it lives in exactly n nodes because there is no redundancy in our tree), we will make a copy of C_2 for each node $\mathcal{T}_i$, such that $\mathcal{T}_i \rightsquigarrow C_2$, and transfer the values of w from $\mathcal{T}_i$ to the new C_2 trees.

Some time later (Fig. 7.1-(5)) in tree C_2, a guard is issued that requires z in the left branch.

Because z is not in C_2, like in Fig. 7.1-(4), we search through C_1 till we realize that z is being speculated and has two values in two $\mathcal{DFV}$'s on the leaves. As we discussed above, since z has two values, we duplicate C_2 to two trees, and move z to the leaves of these tree for P2 to operate on. In this particular case, since C_{21} and C_{22} are the subordinate trees of the leaves of C_1, there is effective no more structure sharing, and the structure in Fig. 7.1-(5) is equivalent to the normal multi-world without optimization.

The next operation of P1 requires z but z is missing from the leaves. Since C_1 has no superior tree, we cannot bring z "down" to P1. Instead, we move it down to where the data is. Hence $P1$ is duplicated and moved to the leaves of C_{21} and C_{22}.

In Fig. 7.1-(6) and Fig. 7.1-(7), program P2 has committed in some leaves, deleted other branches and exit from the system. Eventually P1 has also committed in one of the branches and exit. The state of the system is reduced to just one definite store (w, x, y, z). Notice the sequence of (1) through (7) in Fig. 7.1 can happen any where in a multi-world, and not just the root.

To generalize, we delay the compounding of choice nodes by creating a single subordinate tree for the new choice, and structure share this subordinate tree. Logically the subordinate tree exists in the all the leaves of its superior tree. Data can move up this logical hierarchy by materializing differential views, and move down on demand by programs "living" underneath. Reactors are only duplicated and moved down the hierarchy if the data they require are not above them in the tree. The only exception is when all program continuations from the same program reaches and same program point, the data required are definite at the moment, and they are ready to execute the next step at the same time. Then these programs can be merged up and attached at their common root. However, since this is extremely coincidental, we will not consider such cases.

The following is a non-trivial but useful case when a multi-world can be linearized.

Property 1 *Given a set of programs R whose data requirements are* disjoint *from each other, the structure-shared GCC runtime structure is a linear ordering of $|R|$ subtrees, connected by "$\times$" pointers, where each subtree represents the runtime structure of a respective program.*

7.4 Simulation

To get a sense of how the implementation ideas fare in applications with speculation, we experimented with simulations based on the producer and consumer problem, and obtained some preliminary results. For simplicity, there are n_t types of resources being produced and consumed. We assume the producers' programs do not involve a choice, but just produce a number of resources of different types. The consumers' programs, on the other hand, are made up of just one GCC construct. In each branch, a series of consumptions of resources is done. Each consumption action tries to consume two items of the same type provided they are available, or else the consumer blocks until the items become available later. Each production action produces either 1, 2 or 3 items of the same type at a time.

In this simulation, computations are executed by clock ticks. At each clock tick, either a producer program is submitted to the system, or a consumer program is submitted, or nothing is submitted. At the same tick, every world in the system advances one step by picking one program (either producer or consumer) attached in this world and execute one of its next instruction.

We identify two dimensions of simulation characteristics:

- the level of overlapping interest by the producers and the consumers: high overlapping (HO) and low overlapping (LO)
- the relative rate of production against the consumption high production (HP), low(LP) and balanced production (BP)

We thus create six datasets (HO/HP, HO/LP, HO/BP, LO/HP, LO/LP, and LO/BP) each consisting of 50 producers programs and 50 consumer programs are submitted to the system in random fashion. There is a random number (from 0 to 9) of clock ticks between two consecutive submissions. For HO, $n_t = 5$, while for LO, $n_t = 100$. Table 7.1 records the maximum number of tree nodes, maximum number of program continuations and the maximum size of the data storage (in terms of total number of (variable, value) pairs stored) in each experiment.

We see that although the interaction of 100 programs has the possibility of creating a very large multi-world, the optimization techniques introduced in this chapter are successful in controlling the multi-worlds and the storage requirements. The variation in the numbers in Table 7.1 have an easy and intuitive explanation due to the different nature of the datasets. Balanced

	Max Nodes	Max Progs	Max Data Storage
HO/HP	24	17	6
HO/LP	66	45	6
HO/BP	15	10	7
LO/HP	57	39	51
LO/LP	111	75	63
LO/BP	105	70	22

Table 7.1: Simulation results

production/consumption gives the smallest size tree, and smallest number of programs, while both excessive production and excessive consumption result in larger tree and more computation. The numbers for low-overlapping interest are larger than high-overlapping interest because the consumption and production are more likely to be mismatched given larger number of types of resources. These preliminary results demonstrate the "pay only when you use" principle behind our implementation techniques.

7.5 Conclusion

This chapter presents a number of implementation techniques to address the basic problem of exponentiation in space and time in the speculative model of OCP. It is our belief that these techniques make the implementation of a speculative OCP system entirely plausible. We have carried out some preliminary simulations that indicate the above techniques are promising and more experiments are in the pipeline.

Part IV

System, Applications and Extension

Chapter 8

Prototype System

This chapter presents the architecture and implementation of a prototype system based on the basic model.

At the heart of this prototype system is a constraint logic program representing the store. Current implementation integrates the CLP($\mathcal{R}$) system [JMSY92] as the underlying CLP interpreter/system, and a server registry whose main purpose is to manage a collection of reactors and to communicate with external agents. The reactor language presented in section 3.3 is a Turing-complete language. Because the reactor is going to be embedded in an agent program which is written in the host language, some of the common features such as sequences, interleavings and loops can be omitted from the reactor language, and be programmed in the host language. The resulting simplified set of reactor constructs does not compromise the usability of the reactors. Every reactor in the prototype system is an atomic sustain (guard) with loop options. In this implementation, the host language is chosen to be Python [vRFLD03]. Python is a relatively rich-featured, extendible scripting language with a clean syntax. We provide a simple Python reactor library that handles interpretation of reactor constructs and communication to the server. All agents are meant to be coded in Python language in this system.

The prototype server was implemented in GNU C++, and the Python library was implemented in Python. The system has been installed and tested on various Linux platforms and is also expected to work on Sun Sparc OS and other Unix systems. In what follows, we will present programming of agents with the Python library, and the OCP server architecture and implementation.

8.1 Programming Agents

The basic model of OCP requires that reactor programs be evaluated and managed completely in the runtime system (or the central server). But since the reactor language is a very powerful language which can implement a Turing machine, the interpreter for this language is very complicated. The effort of building such an interpreter will unnecessarily digresses our attention to other less important system issues from the central issue of reactivity in the system. Therefore, in the prototype system implementation, we take a more realistic approach by reusing some of the Python features in the agent language such as interleaving, sequencing and loops. The general idea is to write reactors in both Python and the reactor language, and integrate the reactors in the agent program, and possibly mixed with local Python statements. Because reactors are now written in a mixture of two languages, we adopt the *two-phase interpretation.* In phase one, or the "Python phase", the Python interpreter constructs minimum, atomic reactor objects (which we will further discuss below) from the agent python program, and sends these objects (possibly concurrently if the reactor is within a choice construct) to the server. In phase two, or the "server phase", the reactor objects are evaluated and managed in the server. When the reactor objects finish execution, they are returned to the agent program and the agent program continues.

In the following subsections, we will present how to write reactors with the Python OCP library with feature-by-feature demonstrations, some longer examples, and the internals of the Python OCP library.

8.1.1 Programming style

In order to embed reactors in a Python program, one first need to import the Python OCP library:

```
import ocp
```

An agent begins with a declaration of an agent object such as this:

```
myagent = ocp.Main()
```

And subsequently, all reactor primitives are applied to the agent object `myagent`. Because Python is an object oriented language with classes, the use of objects and methods are similar to other OO languages such as C++ and Java. Therefore, `myagent` object can be extended

with new method by declaring a new agent class which inherits the default `ocp` class, and then declaring my agent as an instance of the new class. For example:

```
class MyOCPClass(ocp.main):
    def __init__(self):
        ocp.Main.__init__(slef)
        self.addproc(self.myfunc)
    def myfunc(self):
        #blah blah goes here

myagent = MyOCPCLass()
myagent.myfunc()
myagent.done()
```

When a new agent object is declared, an Internet domain socket connection is established between the agent program and the default server. At the end of the agent program code, connection should be closed by

```
myagent.done()
```

8.1.2 Atomic update δ and transaction $\langle r \rangle$

Let us first look at the atomic update δ. In OCP system, we define three kinds of atomic updates: *read a base predicate*, *write a base predicate*, *delete a base predicate* and *update a base predicate*. A base predicate is a PROLOG type predicate with a name and a number of ground arguments, e.g. `coord(10, 35)` or `friends_of(ken, [ajay, mike, lily])`. Prolog style wildcards "_" can be used to denote any match for a variable.

- *read.* Reading the values of a number of variables in a predicate is done by the Python primitive `ocp_rb(L, Goal)`, where L is a list of variable names (in character string form), and *Goal* is a CLP *goal* which contains the variables in L, also as character string. A CLP($\mathcal{R}$) goal can be a single predicate or a conjunction of predicates. `ocp_rb(L, Predicate)` returns a status of the read (0 for failed and 1 for successful), and the values of the variables in list. For example, one can write:

  ```
  status, values = myagent.ocp_rb(['X', 'Y'], "coord(X, Y), X>0, X<100")
  ```

  `coord(X, Y), X>0, X<100` is in CLP($\mathcal{R}$) syntax and hence quoted as string. The variables X and Y are CLP($\mathcal{R}$) variables and are not directly accessible in the Python program. If `coord(X, Y)` is defined in the store, one possible pair of coordinates with X

between 0 and 100 will be returned and stored in *values*. If there are multiple solution to a CLP($\mathcal{R}$) goal, the first solution is used. Notice `ocp_rb` is the only atomic update primitive that allows unbound variables and constraints in the predicate.

- *write*. The primitive `ocp_wb(Predicate)` writes a *ground* base predicate to the store, *without* overriding any other predicates. For example:

```
myagent.ocp_wb("friends_of(jack, [don, john])")
```

- *delete*. The primitive `ocp_db(Predicate)` deletes a ground predicate from the store, and fails if no such predicate exists:

```
myagent.ocp_wb("friends_of(jack, _)")
```

  The above statement means delete any predicate that matches `friends_of(jack, ?)`.

- *update*. `ocp_ub(Predicate1, Predicate2)` replaces *Predicate*1 in the store with *Predicate*2. e.g.

```
myagent.ocp_ub("friends_of(ken, _)",
               "friends_of(ken, [ajay, mike, lily, pippin])")
```

  when *ken* has got a new friend *pippin*.

The purpose of atomic transaction $\langle r \rangle$ is to group a number of updates together and execute them in an atomic fashion. In an OCP system, such transaction has to be defined as a goal in the CLP($\mathcal{R}$) program which is also the store. The transaction is effected by running the CLP($\mathcal{R}$) goal. As we will show in the next section, since our CLP($\mathcal{R}$) interpreter is sequential and it runs one goal at a time, atomicity is guaranteed. To invoke an atomic transaction in the agent program, one simple writes:

```
status, ret = myagent.ocp_run(somegoal(x, y, z))
```

where *status* indicates whether the goal has been successfully run, and *ret* carries the return value in a string. All arguments to the goal must be ground.

8.1.3 Sequence r_1; r_2

Python is an imperative, sequential programming language. Statements are interpreted sequentially. Since the semantics of OCP sequence is that, r_1 executes and terminates before r_2 starts, embedding r_1 and r_2 in any Python program, with r_1 preceding r_2 will guarantee the same effect. Therefore, no special construct needs to be provided for reactor sequence.

8.1.4 Interleaving r_1 & r_2

The meaning of r_1 & r_2 is that r_1 and r_2 can execute in interleaving fashion. In the prototype system, we treat the Python interpreter and the store collectively as the interpreter system for reactors, it is possible execute interleaving reactors in different processes in Python and each process may interact with the store independently to achieve the concurrency. Python **os** module has a comprehensive coverage of process management which include common system calls such as `fork`, `wait`, `waitpid`, `system`, `execl`, `kill`. For example, a simple Python skeleton code below could implement r_1 & r_2:

```
children = []
pid1 = os.fork()
if pid1 == 0:                 #the child process
   #r1's code ...
else children.append(pid1)  #the parent process

pid1 = os.fork()
if pid2 == 0:                 #the child process
   #r2's code ...
else children.append(pid2)  #the parent process

for pid in children:
   os.waitpid(pid, 0)
```

8.1.5 Loops

The current version of the prototype system does not support interruptible loop [1], because most applications we have in mind can be developed without this feature. We can, however, emulate the uninterruptible loop using the various normal Python loop constructs such as `while` and `for`. If we want to say:

```
repeat
```

[1] Interruptible loop can be implemented by creating a special primitive `watching(cond)` in the Python OCP library, and watch condition can be indexed and triggered like sustain conditions.

```
    stock(ibm, Price);
    buy(ken, ibm, 100);
until (Price>50)
```

We could simulate it in the following Python code:

```
loop = True
while loop:
    myagent.ocp_rb(['Price'], "stock(ibm, Price)")
    myagent.ocp_run("buy(ken, ibm, 100)")
    loop = (Price <= 50)
```

While this simulation is not completely equivalent to the above reactor because the last statement in the loop and the checking of until condition should be atomic, it serves the purpose of most applications.

On top of the Python loops, we implement a special case of the interruptible loop, i.e. `repeat` δ `watching` c, where δ is an *atomic* update or transaction. To implement this, we simply add an condition argument to the atomic update and transaction primitives: e.g. we extend `ocp_rb(L, Predicate)` to `ocp_rb(L, Predicate, WatchCond)`, and we extend `ocp_run(Goal)` to `ocp_run(Goal, WatchCond)`. The `WatchCond` will be processed, indexed and triggered like a sustain condition, which we will detail in the following section.

8.1.6 Sustain $c \Rightarrow r$

Sustain is the most important feature in OCP which gives rise to reactivity in the system. In the prototype system, we allow r to contain both OCP primitives and normal Python statements. The syntax is as follows:

```
myagent.sustain(Condition)
#Python code
myagent.desustain()
```

`Condition` is in the CLP($\mathcal{R}$) syntax and quoted as a string. This means, any python embodied between `sustain()` and `desustain()` is treated as a reactor r that is being executed under a sustain condition. Since the sustain condition is checked before every *update* to the store, an embedded mixed OCP-Python code, e.g. $(\delta_1;\ \sigma_1;\ \sigma_2;\ \delta_2;\ \sigma_3)$, where δ_i denotes updates to the store, or *global updates*; and σ_i denotes updates to local Python variables, or *local updates*, has the following equivalence:

$$c \Rightarrow (\delta_1;\ \sigma_1;\ \sigma_2;\ \delta_2;\ \sigma_3) \quad \equiv \quad (c \Rightarrow \delta_1;\ \sigma_1;\ \sigma_2;\ c \Rightarrow \delta_2;\ \sigma_3)$$

In other words, the sustain condition is only imposed on global updates, and the original program has turned into an interleaving of atomic sustains and local updates. The atomic sustains, as we will see shortly, will be converted into reactor objects and send to the server for processing.

Let us now focus on the sustain condition `Condition` in the above syntax. Recall that in Definition 8 of subsection 6.1, a sustain condition can be expressed as

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

where $p(\tilde{X})$ is a (reducible) CLP views, in the form of a CLP($\mathcal{R}$) predicate. $\tilde{X}$ are essentially view arguments which are instantiated or constrained by $\Psi(\tilde{X})$, in the reactor. In the prototype system, we further restrict $\Psi(\tilde{X})$ to be only variable instantiation such as $X = 10,\ Y = white$, and not constraints. Notice this restriction does not reduce the expressiveness of the conditions at all since all constraints can be specified in $p(\tilde{X})$. $\Psi(\tilde{X})$ simply instanciate the various parameters specified by $\tilde{X}$. Variables that are not instantiated are *output variables*. To make things easier, in Python reactor, we write just $p(\tilde{X_0})$, where $\tilde{X_0}$ consists of either reactor specific ground values or output variables. Ground values include any numerical values or lists of ground values. One can think of the CLP views as templates. Different reactors can use the same view with different argument instantiations. The Python programmer can only use views available to her which are written in the CLP($\mathcal{R}$) program and nothing else. This provides a level of safety to the store.

For example, one can write

```
myagent.sustain("deliverable(singapore, beijing, 2500, ID)")
```

Here all variables except `ID` are ground. `ID` is an output variable. A final note is that variables (in capital letters) in the sustain condition are bound to the same variables in the action. But these variables are CLP($\mathcal{R}$) variables and are different from the Python variables in the agent program.

8.1.7 Choice $r_1 \mid\mid r_2$

Choices are implemented as Python objects in the prototype system. To use a choice construct, one must first declare a Choice object; and then add predefined procedures to the choice object, before running the choice. For example, the following Python code creates a choice object instance, adds four choices to it, and runs it. Notice that arguments to the choice procedures can be passed in at run time.

```
mychoices = ocp.Choice(None)
mychoices.addproc(self, self.move)
mychoices.addproc(self, self.move)
mychoices.addproc(self, self.move)
mychoices.addproc(self, self.move)
mychoices.run("east", "south", "west", "north")
```

Every branch of the choice structure is implemented as a Python process running in parallel with each other. Every choice object, when created, comes with a unique choice id. All branches of the same choice construct share the same choice id. In case of nested choices, a choice branch is always identified by the id of its closest encapsulating choice object.

8.1.8 Domain definitions

We have mentioned the indexing for set-based attributes in reactor conditions in section 5.8. Applications almost always define a *system domain* for each attributes, for example, if it is the color of cars, there can be maximum 10 colors because these are all the colors the car maker is producing; If it's the price of a stock, the range could be [0, 1000], because stock price more than $1000 would be exorbitant! Reactor programmers, when programming the reactors, may specify a "tighter" domain for each attribute. For example, a car buyer may be only interested in white and silver cars, and the set of all cars interesting to her is just `{white, silver}`. In CLP($\mathcal{R}$) , a set is represented by a list, such as `[white, silver]`. The system domain is required so that the user domain can be translated into binary representation as discussed in section 5.8.

Definition 11 (Domains) *System domain or $\mathcal{SD}$ is the set of all possible values of a given attribute. User domain or $\mathcal{UD}$ is the subset of values of an attribute that are of interest to the user. $\mathcal{UD} \subseteq \mathcal{SD}$.*

When set-based attributes are used in the sustain condition, for each set-based attribute, the system domain need to be provided to the Python OCP library by calling

```
myagent.load_domain_file(DOMAIN_FILE)
```

where `DOMAIN_FILE` is a string that contains a file name to the domains of all set-based attributes. An example of such a domain file is as follows:

```
p:1:a,b,c,d,e,f,g
q:1:white,black,blue,green,orange,yellow,purple,red
q:2:a,b,c,d,e,f,g
```

Every line of this file contains the system domain of one set-based attribute. The format of the line is

```
PredicateName:ArgumentNumber:{Set of comma seperated values}
```

The argument number starts from 0. The purpose of this domain file, which is also used by the server, is to convert sets given in the sustain condition in to a binary number, which is the way all sets are represented in the system. For example, given the domain file above, a sustain condition

```
myagent.sustain("p(5, [b, c, e])")
```

will have the second argument of `p` translated into a binary number `0000 0000 0001 0110b`, or `0x16` in hexadecimal. This binary number can be 32-bit, 64-bit or even longer depending on the system implementation. The binary number will be indexed in the RC-tree as a set for the second attribute of predicate `p`.

8.1.9 Some examples

In the following examples, we will first present a reactor in the reactor language and then we will translate it into Python agent program. Note that this prototype implementation is mainly for the purpose of explaining and testing the concepts in OCP, therefore it has been designed without too much concern for elegance and ease of use. More succinct and expressive agent syntax can be developed in more sophisticated implementations.

Example 10

$$(x \geq 10) \Rightarrow (y = y + x;\ z = z - x)$$

$$\&$$

$$(20 \leq y \leq 40) \Rightarrow (x = x - y;\ z = z + y)$$

Because we are using CLP($\mathcal{R}$) as the store, variables are defined only with respect to predicates. For this example, we assume x and y are defined in the CLP($\mathcal{R}$) as `value(x, X)` and `value(y, Y)`. And there exists a view `range(X, Low, High)`, that defines the range between a variable X. Thus, the complete Python program that represents the above is as follows.

```
import ocp
import os

a=ocp.Main()
a.load_domain_file("domains.dat")

children = []
pid1 = os.fork()
if pid1 == 0:                #the child process
   a.sustain("range(x, 10, inf)")
   a.ocp_ub("value(y, Y)", "value(y, Y+X)")
   a.ocp_ub("value(z, Z)", "value(z, Z-X)")
   a.desustain()
else:
   children.append(pid1)  #the parent process

pid2 = os.fork()
if pid2 == 0:                #the child process
   a.sustain("range(y, 20, 40)")
   a.ocp_ub("value(x, X)", "value(x, X-Y)")
   a.ocp_ub("value(z, Z)", "value(z, Z+Y)")
   a.desustain()
else:
   children.append(pid2)  #the parent process

for pid in children:
   os.waitpid(pid, 0)

a.done()
```

□

Example 11

repeat

$(cell(x+1,y) == 0) \Rightarrow \langle cell(x,y) = 0;\ \ cell(x+1,y) = id \rangle$

||

$(cell(x,y+1) == 0) \Rightarrow \langle cell(x,y) = 0;\ \ cell(x,y+1) = id \rangle$

||

$(cell(x-1,y) == 0) \Rightarrow \langle cell(x,y) = 0;\ \ cell(x-1,y) = id \rangle$

||

$(cell(x,y-1) == 0) \Rightarrow \langle cell(x,y) = 0;\ \ cell(x,y-1) = id \rangle$

until $(cell(x+1,y) > 0 \wedge cell(x,y+1) > 0 \wedge$

$cell(x-1,y) > 0 \wedge cell(x,y-1) > 0)$

This example demonstrates an agent doing "random walk". If a cell at position $(x,\ y)$ is occupied by an agent, then $cell(x,\ y) = id$, where id is the id of the agent. If a cell is not occupied by any agent, then $cell(x,\ y) = 0$. The purpose of the reactor is to repeatedly move to one of its four adjacent cells to its east, north, west, or south, if the cell is not occupied by other agents, that is, $cell(x',\ y') == 0$. The Python code follows.

```
import ocp

myid = 35
class ChoiceAgent(ocp.Choice):
    def __init__(self):
        ocp.Choice.__init__(self, None)
    def move(self, dir):
       status, [x,y], =self.ocp_rb(['X', 'Y'], "cell(%d, X, Y)" % myid)
       if dir == "east":
          self.sustain("cell(%d, %d, 0)" %  (x+1, y))
          self.ocp_run("ub(cell(X, Y, _), cell(X, Y, 0)), \
              ub(cell(X+1, Y, _), cell(X+1, Y, %d))" % myid)
          self.desustain()
       if dir == "north":
          self.sustain("cell(%d, %d, 0)" %  (x, y+1))
          self.ocp_run("ub(cell(X, Y, _), cell(X, Y, 0)), \
              ub(cell(X, Y+1, _), cell(X, Y+1, %d))" % myid)
          self.desustain()
       if dir == "west":
          self.sustain("cell(%d, %d, 0)" %  (x-1, y))
          self.ocp_run("ub(cell(X, Y, _), cell(X, Y, 0)), \
              ub(cell(X-1, Y, _), cell(X-1, Y, %d))" % myid)
          self.desustain()
       if dir == "south":
          self.sustain("cell(%d, %d, 0)" %  (x, y-1))
          self.ocp_run("ub(cell(X, Y, _), cell(X, Y, 0)), \
```

Class	Primitive	Purpose
Main	`load_domain_file(FILENAME)`	Load domain file
Main	`setserver(SERVER_ADDR)`	Set server host address
Main	`setport(PORT)`	Set server listen port
Main	`addproc(PROCEDURE)`	Add a procedure to the current object
Main	`run(ARGS)`	Run the added procedure with list of arguments
Main	`sustain(COND)`	Begins a sustain block with formatted conditions
Main	`desustain()`	Ends a sustain block
Main	`ocp_run(GOAL)`	Runs an atomic goal
Main	`ocp_rb(L, GOAL)`	Reads a list of variables from a goal
Main	`ocp_wb(PRED)`	Writes a ground predicate to the store
Main	`ocp_db(PRED)`	Deletes a ground predicate from the store
Main	`ocp_ub(PRED1, PRED2)`	Updates PRED1 with PRED2
Main	`ocp_consult(CLPRFILE)`	Makes the server consult a CLP($\mathcal{R}$) file
Main	`ocp_noop()`	No-op
Main	`ocp_commit()`	Commits a branch
Main	`ocp_done()`	Ends reactor and closes socket connect
Choice	`__init__(PARENT)`	Set a parent choice in nested choices

Table 8.1: Python API Reference

```
                ub(cell(X, Y-1, _), cell(X, Y-1, %d))" % myid)
            self.desustain()

myagent= ChoiceAgent()
myagent.load_domain_file("domains.dat")
myagent.addproc(myagent, myagent.move)
myagent.addproc(myagent, myagent.move)
myagent.addproc(myagent, myagent.move)
myagent.addproc(myagent, myagent.move)
loop = 1
while loop:
    myagent.run("east", "north", "west", "south")
    myagent.sustain("cell(X, Y, %d)" % myid)
    ret, loop = myagent.ocp_run("cell(X+1, Y, ID1), ID1>0, \
                                 cell(X-1, Y, ID2), ID2>0, \
                                 cell(X, Y+1, ID3), ID3>0, \
                                 cell(X, Y-1, ID4), ID4>0,")
    myagent.desustain()
```

□

We end this section with a summary of the Python API in table 8.1.

8.1.10 Internals of the OCP library

The Python OCP library in defined in a single file called `ocp.py`, which is imported as a module by any Python program which wants to interact with the server. The `ocp.py` module contains two classes: Main and Choice, and Choice is a derived class from Main. Main class is defines a normal reactive agent that can interact with the shared store with primitive such as `ocp_rb()` and `ocp_run()`, and, possible under sustain condition. Sustains can be nested, and this is implemented by simply putting the conditions on a stack. The Choice class is a special ocp.Main class. Among its additional attributes are choice id, and a list of procedures each of which represents a branch to this choice. Every procedure, when started with the `run()` primitive, starts a new process, and runs independently with each other.

As we have shown in section 8.1.6, this prototype system decomposes every complex reactor into a sequence of atomic (sustainable) reactors. Each of the reactors is essentially a (*condition*, *action*) pair where condition is compiled from the condition stack and action is one of those listed in section 8.1.2. Beside condition and action, the reactor object also contains the the choice id if this reactor is in one branch of a choice construct, the Internet address and port of the agent host machine, and a generated password.

The communication model of the system is a connection-oriented socket connection. When instantiated, every ocp.Main object and every process in the choice object starts a connection-based communication channel with the pre-defined server at a pre-defined port. Server and port can be set with `setserver(SERVER)` and `setport(PORT)`. The agent behaves both in client mode and server mode in this system. When it submits the reactor to the server, it is in the client mode. If the server acknowledges the receipt of the reactor and starts to process it, the agent turns into server mode and listens to the socket for messages from the server. This is why the reactor object contains the agent's host address and listen port. To avoid getting wrong message from other entities than the server, a password is generated and attached to the reactor as well. Only replies with the correct password is accepted by the agent. The communication protocol between the agent and the server is depicted in figure 8.1.

The first two messages are for handshake. If OCP server is overloaded with reactors, it may send a `BUSY:rid` instead of `OK:rid`, where `rid` is a unique reactor id assigned by the OCP server. In that case, the ocp module will automatically retry sending the reactor after a period of time.

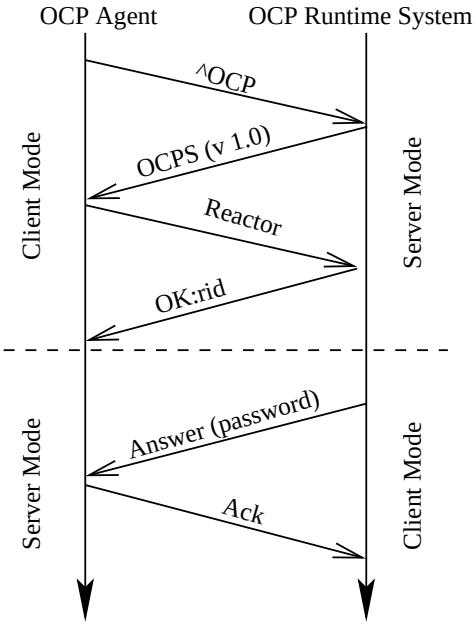


Figure 8.1: Communication Protocol between the Agent and the server

Choice committals are settled in the server and will be discussed in the next section. When one branch of a choice commits, other running and blocked choices will be nullified, and the control will return to the calling processes, which will exit by themselves. When all processes of a choice exit, the parent process calls a special internal primitive `release_choice()` that sends a special reactor to the server to release the choice id of this choice in the server. This notifies the server that this choice has been successfully settled.

8.2 The Server

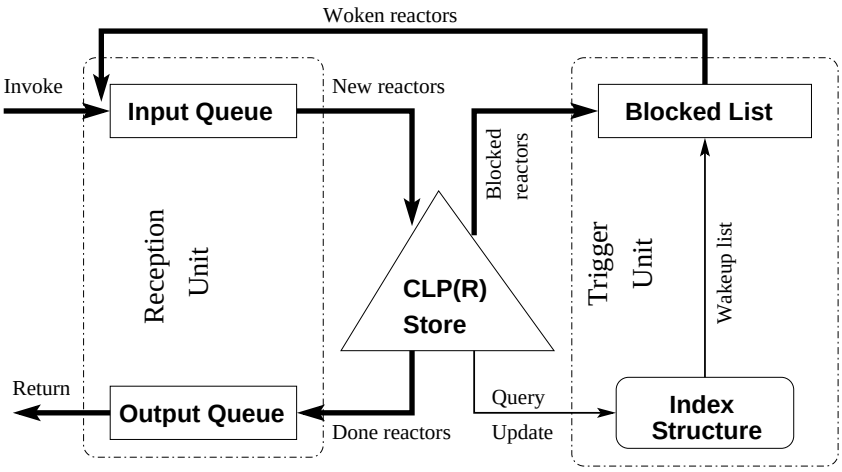


Figure 8.2: Architecture of the server

The basic model is characterized by a number of agents programming against a centralized shared store. Because reactors are invoked asynchronously and concurrently, and there is only one store, the reactors need to be serialized. Most importantly, when some reactors are blocked and blocked due to unsatisfiability of their sustain conditions, a triggering mechanism is required to "wake up" some of them, if the store has been changed and some of the conditions become true again. In the prototype system, the central server registry is responsible for the communication, serialization, execution and the triggering of the reactors.

The server is a multi-threaded run-time system. Structurally it consists of five modules: an input queue, an output queue, a CLP($\mathcal{R}$) store, a blocked list and an index structure (see figure 8.2). And these can be roughly divided into three work units: the reception unit, the store and the trigger unit. The dark arrows in the figure represents the flow of reactors. Except for the CLP($\mathcal{R}$) store which is implemented in C, the server is implemented in C++ with POSIX thread library. In the following subsections, we will discuss these units individually and how they work together to form a working runtime system.

8.2.1 Reception unit

The reception unit contains two queues: an input queue and an output queue, and each of them is manipulated by a separate thread. The input thread waits for incoming reactors from agents, serializes them, and pushes them to the input queue; the output thread watches an output queue, and output the completed reactors to their originating agents whenever they appear on the queue.

As we have mentioned in section 8.1.10, the agents communicate with the server by the Internet socket connection. When the server is started, the input thread opens the socket at a known port (e.g. 9999) and listens for connection. When an agent connects, the input thread will spawn a *conversation* thread to handle this request and continue listening itself. After successful handshake between the agent and the conversation thread, the agent will send over the reactor definition within a long string delimited with "#". The conversation thread parses through the string to pick up all the fields such as sustain condition in CLP($\mathcal{R}$) format, action goal, choice id, agent address and port, password and so on. Then the conversation thread uses this information to construct an reactor object. The constructed reactor objects are then pushed to the input queue, ready for evaluation. All queues are implemented as doubly-linked lists. The prototype of the reactor object class is as follows.

```
class ocp_reactor
{
 int id;              //reactor id
 char* hostname;      //agent hostname
 short hostport;      //agent host port
 char* passwd;        //agent's password
 int choiceid;
 char* vars;          //variables used in this reactor
 char* cond;          //sustain condition in CLP(R) format
 char* action;        //action goal
 char* answer;        //answer string
 bool hot;            //hot is 1 is the reactor is currently
                      //waiting to be/being evaluated
                      //hot is 0 if it's currently in the blocked mode
 bool blocked;        //1 if reactor has been blocked before, otherwise 0
 ocp_data_str* last; //pointer to last reactor in the queue
 ocp_data_str* next; //pointer to next reactor in the queue
};
```

The `hot` flag is provided to take into account of the case of one reactor being woken up a number of times by a single CLP($\mathcal{R}$) goal in an action. This is also to provide for the case of a query point (see later) falls on the boundaries of RC-tree partitions and wakes up a reactor more than once. With `hot`, we will not wake up a reactor again if it is already awaken. The `blocked` flag helps to determine if the reactor needs to be indexed if the sustain condition fails. If `blocked` is false, then this reactor is a new one and will be indexed; otherwise nothing is done.

The output thread behaves like a client. Whenever a reactor appears in the output queue, which means, it has been executed and completed, the output thread checks the agent address, port and password in the reactor object, and sends the answer string back to the agent. If the agent is not contactable, the output thread puts the reactor to the back of the queue and try later. A reactor that cannot be returned after a period of time will be discarded by the server.

8.2.2 CLP($\mathcal{R}$) Store unit

The CLP($\mathcal{R}$) store unit actually is comprised of two independent components: one is the modified CLP($\mathcal{R}$) system; the other is a C++ wrapper thread which manages the start, terminate and restart of the CLP($\mathcal{R}$) system, and acts as a communication interface between the CLP($\mathcal{R}$) system and other units and components in the OCP server. We call the wrapper *run thread*. The run thread communicates with the CLP($\mathcal{R}$) system via message queues [2] and with other thread

[2] The CLP($\mathcal{R}$) 1.2 system has been modified to direct all standard output to the message queue that connects the run thread.

through shared variables with synchronization.

The CLP($\mathcal{R}$) language is an instance of the CLP scheme defined by Jaffar and Lassez [JL87]. Its operational model is similar to that of PROLOG. A major difference is that unification is replaced by a more general mechanism: solving constraints in the domain of uninterpreted functors over real arithmetic *terms*. For detailed programming techniques and system function library, please refer to [HJM+92]. Further technical information on CLP($\mathcal{R}$) is available on language design and implementation [JMSY92], meta-programming [HMSY89] and delay mechanisms [JMY91].

A CLP($\mathcal{R}$) program is a collection of *facts* and *rules* as in PROLOG. The definition of a rule is similar to that of a PROLOG clause, but it differs in two important ways: rules can contain constraints as well as atoms in the body, and the definition of terms is more general. A *goal* is a rule without a head, as usual.

The body of a rule may contain any number of arithmetic constraints, separated by commas in the usual way. Constraints are equations or inequalities, built up from real constants, variables, `+`, `-`, `*`, `/`, and `=`, `>=`, `<=`, `>`, `<` where all of these symbols have the usual meanings and parentheses may be used in the obvious way to resolve ambiguity. Unary arithmetic negation is also available, as are some special interpreted function symbols.

The CLP($\mathcal{R}$) system v1.2 is a virtual machine based CLP compiler which supports dynamic code and bahaves like an interpreter. The interpreter is loaded with one CLP($\mathcal{R}$) program which represents the store of our operational model, and it is used to process one user goal at a time. To simplify the implementation in the system, we disallow updating of rules, and only allow reading, adding, deleting and changing of facts (base predicates) in the program. And these are done by the added CLP($\mathcal{R}$) system predicates `rb/2`, `wb/1`, `db/1` and `ub/2`, respectively [3]

Due to the code space constraint of the CLP($\mathcal{R}$) system, the prototype OCP systems needs to restart CLP($\mathcal{R}$) periodically. It saves all dynamic facts in a "dynamic file" and reloads it when it restarts the CLP($\mathcal{R}$) system. For example, a dynamic file could look like the following.

```
program('/home/kzhu/ocp/ocp/shipping.clpr').
dyn([[p, 2], [q, 2]]).
:-dynamic(map, 3).
map(seoul, shanghai, 4668).
map(singapore, quebec, 14633).

:-dynamic(vessel, 3).
```

[3]The numbers after the slashes are arities of the predicates.

```
vessel(hongkong, shanghai, 20000).
vessel(singapore, beijing, 10000).
```

In this case, `shipping.clpr` is the original CLP($\mathcal{R}$) program (from figure 6.2) which contains all the rules and is stored separately from the dynamic file:

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

%recursive case: A...C-D...B
deliverable(A, B, Budget):-
  Cost1+Cost2+Cost3 <= Budget,
  deliverable(A, C, Cost1),
  vessel(C, D, Cost2),
  deliverable(D, B, Cost3).

index(deliverable(_, _, Budget)):-
  wrap("index_deliverable1(C): 0<C<=%\n", [Budget]), fail.
index(deliverable(_, _, Budget)):-
  wrap("index_deliverable2(C1, C2): 0<C1+C2<=%\n", [Budget]), fail.
index(deliverable(_, _, Budget)):-
  wrap("index_deliverable3(C1, C2, C3): 0<C1+C2+C3<=%\n", [Budget]).

trigger(vessel(_, _, C), [index_deliverable1(C),
        index_deliverable2(C, _), index_deliverable2(_, C),
        index_deliverable3(_, C, _)]).
trigger(map(_, _, C), [index_deliverable3(C, _, _),
                       index_deliverable3(_, _, C)]).
```

The predefined predicate `index(V)` wraps an string that can be easily parsed into an indexable object that corresponds to the view `V`, and sends it to the trigger unit, using the special system predicate `wrap`. Every view in the CLP($\mathcal{R}$) program needs to be provided with a corresponding `index` rule so that when a condition based on the view fails, an indexable object can be created. In the above example, `deliverable` predicate has three rules, the first rule has just one variable `Cost` when `A`, `B` and `Budget` are ground by a blocking condition. Hence the indexable object should be 1-dimensional on variable `Cost`. Similarly the second rule gives an object of 2 dimensions on `Cost1` and `Cost2`. The third rule is a recursive one. According to the abstraction, this can also be reduced to a 3-variable constraints on `Cost1`, `Cost2`, and `Cost3`. Thus we have three index rules, each of which creates an object on different dimensions.

The names of the index trees that the objects will be inserted to are `index_deliverable1`, `index_deliverable2`, and so on.

Another predefined predicate `trigger/2` hashes a new update of a base predicate to a list of triggers that will be used as queries to different index trees. The predicates of the triggers are the identifiers of the index trees. The use of `index` and `trigger` will be discussed further in section 8.2.3.

After the CLP($\mathcal{R}$) system is started, the run thread goes into an infinite loop. In every iteration, it blocks and waits for a reactor object to appear in the input queue, and if it does, the run thread pops the reactor from the queue and starts evaluation.

During the evaluation phase, the sustain condition is run as a CLP($\mathcal{R}$) goal by the interpreter. If the goal succeeds which means the store entails the condition, then the action portion of the reactor is run as another goal immediately afterwards with all the variable bindings in the condition. The status of the run, i.e. success or failure, as well as possible answer to the goal are packaged into the reactor object which is then appended to the output queue. Successful updates to dynamic facts of the CLP($\mathcal{R}$) program will produce a number of triggers (by using the `trigger/2` predicate) which are picked up by the run thread and sent to the trigger unit.

If the sustain condition evaluates to fail, an additional goal involving `index/1` is run to obtain the definition of the "shape" that corresponds to the condition (or abstraction of the condition) in a string, and the string as well as the reactor object itself are sent to the trigger unit for further processing.

We added another system predicate `inset/2` to allow for set-based constraints. `inset/2` is implemented as:

```
inset(X, [X|_]).
inset(X, []):-fail.
inset(X, [Y|L]):- inset(X, L), !.
```

If a reactor which carries a choice id makes a successful update including actions that involve `wb`, `db`, `ub` and `cm` to the CLP($\mathcal{R}$) program, this means the branch in which this reactor is issued from has committed. At this point, the run thread searches through the blocked list and the input queue for blocked and pending reactors with the same choice id, and purges them from the system by setting `CHOICE_KILLED` as the answer to these reactors and pushing them to the output queue. However, due to the asynchronism of the system, some reactors from the same choice structure may keep arriving at the server. To purge these "late-coming" reactors

as well, the system stores the committed choice id in a list, and for any reactors that comes with a committed choice id, it is immediately returned to the output queue without further processing. When the agent program has finished its choice, it sends a *release choice* reactor to the server. Upon receiving this reactor, the server removes the choice id from the committed choice id list, so that the same id can be used again by other agents.

8.2.3 Trigger unit

The trigger unit is made up of a linked list of blocked reactors, and a separate index structure which comprises of a number of RC-trees. The blocking conditions of the blocked reactors are indexed in these RC-trees in the form of indexable objects. The blocked reactors are stored in a linked list in order to achieve constant update time. No search will be involved in this blocked list, as the addresses of all the reactors in the list are also indexed in the index structure. We will now focus on the implementation of triggering in the prototype system, and the arrangement of the index structure, in particular.

Indexable objects

The various techniques introduced in chapter 6 reduce a composite blocking condition to a basic one which can be formulated as a k-dimensional object and be inserted into a k-dimensional RC-tree. This object is called an *indexable object*, or *indexable* in short. When a sustain condition blocks, the run thread sends the reactor and the string representation of the indexable object to the trigger unit. Every indexable string comes with a header which is the identifier *id* of the index tree to be inserted to. The reactor is stored in the "Blocked List", and the string is parsed into an `indexable` data structure of the following format:

```
class indexible_object
{
 void* addr;                 //address of the reactor to which this obj belongs
 int dimension;              //dimension of the object
 struct attribute* attribs; //attribute domains
 int numcons;                //number of linear inequalities
 inequality* cons;           //the linear inequalities
 object* next;               //pointer to next object, used in RC-tree
};
```

The constraints in an indexable is a conjunction of a number of linear inequalities. The indexable also records the address of the original reactor in the blocked list, so that when this

object is retrieved in a search of the RC-tree, the reactor can be retrieved from the blocked list in constant time. When an indexable is constructed, it is inserted into a RC-tree *id*. The dimensionality of the object and the tree should match.

Triggers

In the CLP($\mathcal{R}$) system, updates to the program are done through system predicates `assert/1` and `retract/1`, which inserts and delete a predicate, respectively. The prototype system produces triggers by "trapping" every atomic update, with the following "wrappers" to `assert` and `retract`.

```
wb(T):- assert(T), trigger(T, L), make_triggers(L).
db(T) :- retract(T).
ub(T1, T2):- db(T1), wb(T2), !.
```

`wb/1` writes a base predicate. `db/1` deletes a base predicate. `ub/2` replaces one base predicate with another. `make_triggers` system predicate creates a string of triggers, and sends them to the trigger unit. In general, only insertion of facts may re-enable a reactor because we do not allow negation in the sustain conditions. Thus only `wb` updates are captured and transformed into a list of queries for the index trees.

EXAMPLE 12

Consider the `shipping.clpr` program in section 8.2.2. Suppose an update `wb(vessel(shanghai, singapore, 7800))` happens to the store, it will produce four triggers:

```
index_deliverable1(7800)
index_deliverable2(7800, _)
index_deliverable2(_, 7800)
index_deliverable3(_, 7800, _)
```

The first trigger's variable is ground, thus can be transformed into a 1-dimensional *point query* `(7800)`. The rest of the triggers have wildcards, and hence are transformed into a two or three-dimensional *range queries* such as `(7800, *)` and `(*, 7800, *)`. These queries will be used to search in the index-trees `index_deliverable1`, `index_deliverable2` and `index_deliverable3`, respectively. □

We can see from this example that, because views are constructed from a number of base predicates, and every update to the store is only changing *one* base predicate, the search queries

constructed for views are "partial" queries, or queries with just some of the dimensions fixed.

Wake up

Here we discuss what should be done when we wake up reactors. When an update occurs, typically a number of `wb` triggers are produced. These triggers are used to query the various index trees in the index structure, which may return a list of indexables with pointers to their original reactors in the blocked list. Reactor in this returned list is said to be *successfully triggered*. We call this list the *wake up list*. Reactors on this list should be woken up and join the input queue again for re-evaluation. To ensure fairness of the system and guarantee the eventual evaluation of the other pending reactors in the input queue, we append the woken up reactors at the end of the queue instead of preempting them at the beginning.

However, because almost no indexing techniques are 100% accurate, especially when abstraction is used, the returned set of reactors is usually a superset of the set of reactors whose conditions are exactly enabled. In the other words, we may have some *false hits*. Furthermore, even if all the returned reactors are indeed enabled now, some of them may not run through the test. This is because the CLP($\mathcal{R}$) system is sequential, and given a list of woken up reactors $c_1 \Rightarrow r_1$, $c_2 \Rightarrow r_2$, etc., and, executed in that order, execution of r_1 may change the store from Δ to Δ' and $\Delta' \not\models c_2$, causing $c_2 \Rightarrow r_2$ to be blocked again. Consequently, we adopt a "lazy" deletion approach and delay the removal of query-hit indexables from the index structure, until the woken reactor is successfully executed. If the woken reactor fails the condition test again, its sustain condition doesn't have to be compiled into indexables and re-inserted into the index trees again. If condition succeeds, then all indexables compiled from this condition will be deleted from their respective RC-trees.

8.2.4 Performance evaluation

This subsection documents some experimental experiences of using the prototype system for three simple reactors on a Linux cluster. The Linux cluster contains 14 access nodes and 42 computation nodes. The access nodes are each equipped with a Pentium 4 2.8 GHz dual CPU running Linux kernel 2.6.10, and are connected via fast Ethernet. For this experiment, we only use up to 13 access nodes. One of them houses the server and the agent programs are launched on the other 12 nodes.

Below is the agent program used in the performance testing. Basically this program has three

reactors: *noop*, *read a base predicate repeatedly*, and *update a base predicate repeatedly*. These reactors do not involve any blocking because once blocked a reactor can wait for arbitrarily long. Instead they may involve some complex sustain conditions or actions that are guaranteed to succeed. We can comment out two of the reactor invocations at the bottom of the code and just run one of the three at a time. The purpose of this experiment is to measure of the average throughput of the system in terms of the number of reactors processed per sec with one, two and up to twelve agents connecting to it simultaneously.

```
import ocp
import time
import sys
import string

class Player(ocp.Main):
 def __init__(self):
 ocp.Main.__init__(self)

 def noop(self):
 for i in range(10000):
 status, ans = self.ocp_noop()
 def read(self):
 for i in range(10000):
 status, ans = self.ocp_rb("X", "p(X, c)")

 def update(self):
 for i in range(10000):
 status, ans = self.ocp_ub("p(X, d)", "p(X+1, d)")

r=Player()
r.load_domain_file("domains.dat")
r.noop()
#r.read()
#r.update()
r.done()
```

Before we test the throughput of the server, we first measure the throughput of the CLP($\mathcal{R}$) store unit. With the message queue interface between CLP($\mathcal{R}$) system and the run thread, the store takes about 20,000 goals per sec. Thus this will be the upper bound of the overall system throughput.

Table 8.2 records the throughput numbers for all three reactors with the increasing number of parallel agents interacting with the server. We further plot these numbers of figure 8.3. The throughput is measured by counting the time it takes to output 10,000 reactors to external agents. From the table and the graph, it is clear that due to the increasing complexity of these

No. of Agents →	1	2	4	6	8	10	12
Noop	448	870	1697	1740	1869	1775	1717
Read	315	589	1050	1125	1181	1130	1175
Update	298	615	1071	1174	1147	1118	1115

Table 8.2: Throughput on Linux cluster (# of reactors per sec)

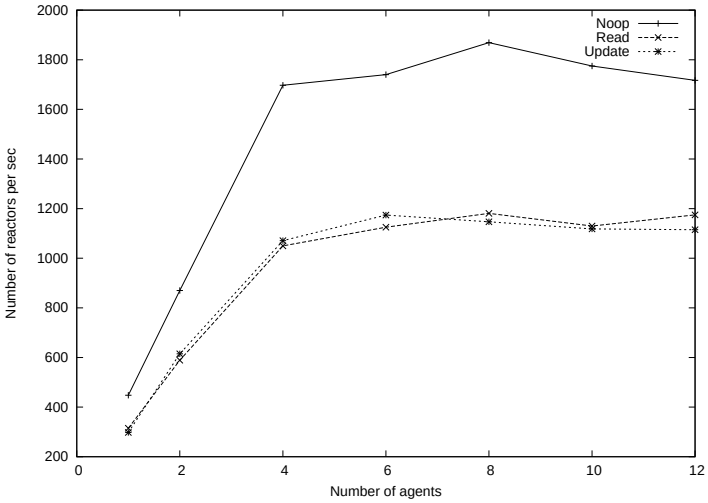


Figure 8.3: Throughput Performance of the Server

three types of reactors, there exists the following general relationship:

$$throughput(noop) > thoughtput(read) > throughput(update).$$

As for the relationship between the number of parallel connections and the throughput, we observe that with small number of parallel connections, the OCP server is under-utilized, and the utility increases as the number of parallel connection increases. However, the utility saturates and in fact reduces a bit when there are too many connections (i.e. ≥ 10), which indicates that communication overhead has taken effect. While the performance of this prototype system (with throughput in the thousands) may satisfy some small scale applications, there is certain a lot of room for improvement. We will discuss ideas that will contribute to the distribution of the OCP system to make it truly scalable in chapter 10.

Chapter 9

Applications

Open Constraint Programming has found many applications where distributed multi-agent computation is required. OCP comes particularly handy when the agent communication can be naturally done using a central medium, which, in our case, is the store or the knowledge base written in CLP. The application we are considering are in the areas of e-commerce, problem solving, distributed simulation, real-time logistics and workflow management. In the following sections, we will introduce a few specific application prototypes that we have designed or built in each of these areas. The purpose of this chapter is not to demonstrate deployed applications but rather to give examples of where and how OCP can be useful. Reactors in this chapter are written in the generic reactor language which may be readily implemented in the prototype system.

9.1 Stock Trading System

9.1.1 Introduction

Online stock trading has been the industrial norm since the beginning of the Internet era. With the performance and security of the Internet enhanced over the last decade, online trading has also become increasingly sophisticated. Watchdogs, triggers, future executions are available to many day traders around the world. For example, TradeStation.com [Tra], which represents the state of the art in online trading, now offers rule-based trading that allows the traders to specify trading strategies using *EasyLanguage*. The broker offers a dictionary (library) of trading conditions and commands with parameters to be set by the users. The industry, however, calls for more flexibility and complexity in strategy building, especially for advanced traders. In fact, a stock trading system is the original motivation when we designed OCP. Previous research

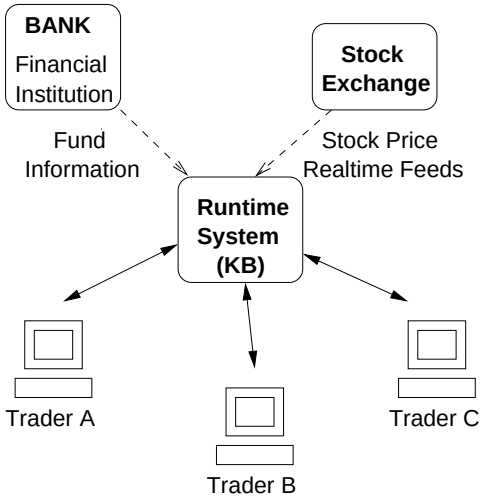


Figure 9.1: Architecture of the Stock Trading System

on agent-based portfolio management such as [SZD98] focuses more on portfolio selection or portfolio monitoring, the focus here, however, is to provide a multi-agent trading platform for traders to interact and trade in real time with the ability to specify complex rules and strategies.

We model an automated stock trading system as consisting of a centralized knowledge base, a number of autonomous trading agents deployed by human traders, stock exchange agent which supplies the real time stock information feeds and financial institution agent that provides fund informations of the traders. We assume that one human trader may deploy multiple program agents to do automatic trading, therefore bank account sharing is common in this system. The knowledge base contains dynamic facts such as stock prices, volumes, user accounts and portfolios, and both system and user defined constraint logic rules to be used as conditions or actions. We illustrate such a stock trading system in figure 9.1. In what follows, we will discuss some minimum requirements of the knowledge base for stock trading system, and the development of various agents in the system.

9.1.2 Trading knowledge base

As we have mentioned before, the knowledge base in OCP typically contains some dynamic facts or the views of various facts, and some constraint logic rules to be used as either sustain conditions, or as operations. We will now present only the basic components in such stock trading knowledge. In a real life application, the knowledge base will be much more complicated.

Dynamic facts

```
stock(ibm, 15.6, 500000).
stock(hp, 22.4, 800000).
own(ken, ibm, 500, 10.95).
own(john, hp, 800, 20.10).
cash(ken, 5000).
cash(john, 15000).
```

The above CLP($\mathcal{R}$) program shows a few types of facts used in this application. The first two facts are examples of stock name, last price, and current volume. The third and fourth facts indicate the stock ownership of two traders, i.e. the name of the stock they own, the volume of the stock and the done price. And the last two facts records the current cash position of the two traders. Note that the first type of facts is the most volatile and dynamic, and is supplied by the exchange as real time feeds. `own/3` is defined and stored on the stock trading system itself as a system predicate. The `cash/2` is a predicate updated both by the trading system (when the trader trades) and by the banks (in case the trader deposits/withdraw cash to his account).

Views as bundles, conditions and actions

It is often useful to define views on the base facts to facilitate easy access to a group of facts with some common properties. For example, the trader `ken` could have the following definitions in the knowledge base about `cheapstock`, `dearstock`, `highrisk`, `lowrisk`, etc.

```
bundle(cheapstock, S):- stock(S, P, _), P<=1.
bundle(dearstock, S):- stock(S, P, _), P>=10.
bundle(highvolume, S):-stock(S, _, V), V>=100000.
bundle(lowvolume, S):-stock(S, _, V), V<=10000.
bundle(lowrisk, S):- dearstock(S), highvolume(S).
bundle(highrisk, S):-cheapstock(S), lowvolume(S).
```

With views like this, the traders are able to implicitly *bundle* stocks together according to his or her requirements. So when `ken` talks about `dearstock`, he is actually talking about a bundle of stocks that satisfy the conditions. They can also be used to test a condition such as whether a given stock is a dear stock by `ken`'s definition. The trader can also define *values* of his or her bundles by using the following rule (where `findall(Template, Predicate, Enumerator)` is a system predicate that finds all satisfying instances of *Template* and put them into a list *Enumerator*):

```
value(User, B, V):-
    findall(Stock, (bundle(B, Stock), own(User, Stock, _)), L),
    sum_value(User, L, V, 0).
sum_value(_, [], V, V).
sum_value(User, [S|L], V, V0):-
    stock(S, P, _), own(User, S, Amt),
    sum_value(User, L, V, V0+P*Amt).
```

One can also define more complex bundle such as lowrisk/highrisk as joins of other bundles.

With these views, the trader can also specify complex conditions such as these:

```
can_afford_lowrisk_stock(X, V):-
    bundle(lowrisk, X), stock(X, PX, _),
    cash(ken, C), PX*V<=C.

imbalanced(User, B1, B2):-
    value(User, B1, V1), value(User, B2, V2),
    V1>=V2*1.1.
imbalanced(User, B1, B2):-
    value(User, B1, V1), value(User, B2, V2),
    V2>=V1*1.1.
```

Here trader `ken` specifies a condition of being to able to afford V shares of lowrisk stock X. This condition can then be used as a blocking condition before a buying action. Another condition is the values of two bundles not being balanced which can trigger rebalancing actions.

For simplicity, we assume the bid and ask price of a stock to be the same as the last price. Then buying and selling of a stock can also be coded as rules:

```
buy(User, Stock, Amount):-
    stock(Stock, P, _),
    ub(cash(User, C), cash(User, C-P*Amount)),
    (own(User, Stock, _) ->
      ub(own(User, Stock, OldAmt), own(User, Stock, OldAmt+Amount));
      wb(own(User, Stock, Amount))).

sell(User, Stock, Amount):-
    stock(Stock, P, _),
    ub(cash(User, C), cash(User, C+P*Amount)),
    ub(own(User, Stock, OldAmt), own(User, Stock, OldAmt-Amount)).
```

Of course, the users can write more complicated actions such as buy/sell of a percentage of a bundle, balancing the high risk and low risk stocks in their portfolios, etc.

9.1.3 Agents

In its simplest form, the stock trading system we are considering has three types of agents: Stock exchange, banks, and traders.

Stock exchange and the banks simply updates the `stock/3` and `cash/2` predicates periodically. Here we focus on the trader agents. With the kind of knowledge base in section 9.1.2 and the OCP reactor language that we have introduced in section 3.3, it is possible for the traders to specify trading strategies like the following:

Strategy 1: if price of IBM ≥ 50, then sell 100 shares of IBM

Strategy 2: if my cash level is above 5000, buy 1000 shares of a cheap stock or buy 100 shares of dear stock

Strategy 3: repeatedly buy 100 shares of any low risk stock, as long as my cash is more than 30% of my total value.

Strategy 4: if the value of my cheap stocks bundle is different from my dear stocks bundle by more than 10%, then re-balance my portfolio to equalize them.

For example, strategy 2, 3 and 4 can be translated into OCP reactors as in figure 9.2.

Reactor 2:
$(\texttt{cash}(ken,\ C),\ C \geq 5000),\ \texttt{bundle}(cheapstock,\ S)) \Rightarrow \texttt{buy}(ken, S, 1000)$
$||$
$(\texttt{cash}(ken,\ C),\ C \geq 5000),\ \texttt{bundle}(dearstock,\ S)) \Rightarrow \texttt{buy}(ken, S, 100)$

Reactor 3:
repeat
  $\texttt{can\_afford\_lowrisk\_stock}(S, 100) \Rightarrow \texttt{buy}(ken, S, 100)$
until $(\texttt{cash}(ken, C),\ \texttt{value}(ken,\ mystocks,\ V),\ C < 0.33 * V)$

Reactor 4:
$\texttt{imbalanced}(ken,\ cheapstock,\ dearstock) \Rightarrow \langle \texttt{rebalance}(cheapstock, dearstock) \rangle$

Figure 9.2: Sample Stock Trading Reactors

Just like TradeStation.com, our stock trading system can provide a nice graphical user interface such as in figure 9.3, to alleviate the programming effort of the traders, and to help the traders keep track of their portfolio, bundles, and defined strategies. To balance one's portfolio, for example, the trader can use an interface like in figure 9.4 to specify the bundles she wants to balance and the tolerance, etc.

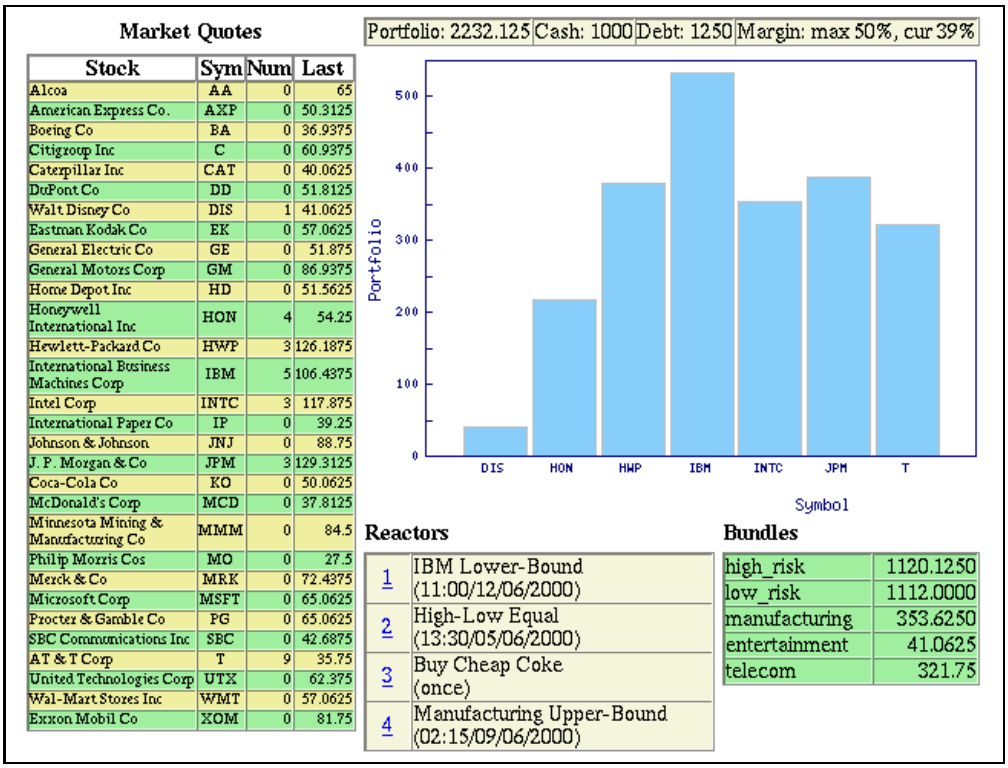


Figure 9.3: The GUI of the Stock Trading System

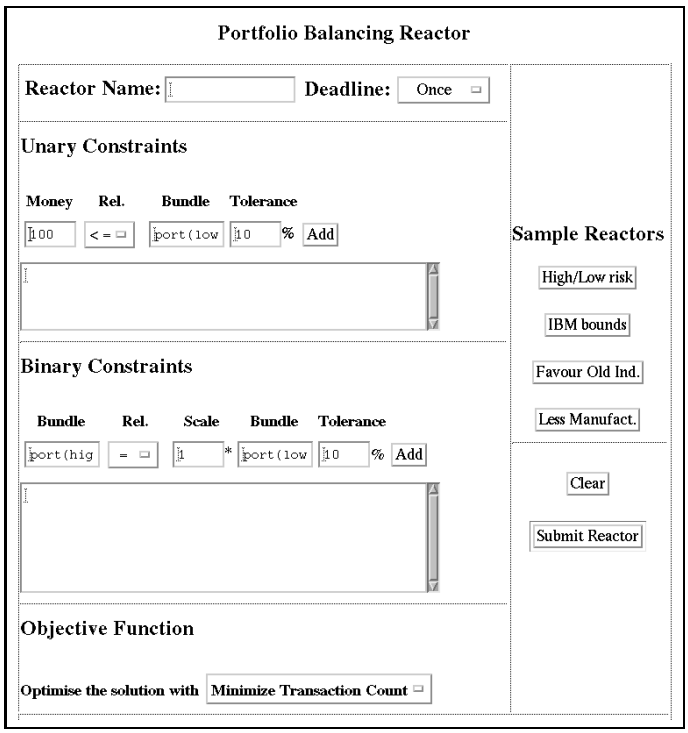


Figure 9.4: Balancing Strategy User Interface

9.2 Coordinated Constraint Search

In this section, we give an example of coordinated parallel constraint search, implemented in OCP. We consider a classic application of sequential constraint programming, backtracking search. The problem at hand involves a store of constraints over $\{0, 1\}$ (or binary constraints), and to find a binary assignment to the decision variables $X_1, \ldots, X_n$ so that some base conjunction of constraints $\mathcal{C}_0(X_1, \ldots, X_n)$ is satisfied, or report failure in case no such assignment exists. The constraints $\mathcal{C}_0$ need not be specified here, but we assume that the store contains just one conjunction of atomic constraints, represented as a set of its conjuncts. A search for a one solution for decision variables $X_1, \ldots, X_n$ can be realized by the C-like program below, which uses the updates `softtell` and `retract`. The former was described above in the constraint database example in section 3.2, while `retract(`$\mathcal{C}$`)` has the effect of removing the constraint $\mathcal{C}$ from the store. In what follows, we assume the decision variables are organized into an array $X[i],\ 1 \leq i \leq n$. The procedure below is called with `solve(1)`.

```
solve(i) {
    if (i > n) return TRUE;
    if (softtell(X[i] = 0)) {
        if (solve(X[i + 1])) return TRUE
        retract(X[i] = 0)
    }
    if (softtell(X[i] = 1)) {
        if (solve(X[i + 1])) return TRUE
        retract(X[i] = 1);
    }
    return FALSE
}
```

Thus the constraint system here acts with the above agent in functionally the same way a constraint programming library (eg. ILOG Solver [ILO, ILO01]) does to regular sequential programs.

Parallel and distributed algorithms for search problems have been previously reported in [FM87, KZ88, Sos94, BL94, Sch00, JSYZ04]. Our goal is to distribute the `solve(i)` to multiple concurrent agents so that they can work to solve the constraint search problem in parallel in the OCP system. To this end, the store needs to contain a constraint database for coordination purposes. The current setting involves a master agent which serves to divide the search space amongst any number of worker agents that volunteer at any time. Consider the search tree T for this problem. This is a binary tree of $m + 1$ levels, whose left subtree represents the choice

$x[1] = 0$ and the the right subtree $x[1] = 1$, and similarly for the remaining nodes w.r.t. the other decision variables. We identify the individual nodes of T using a string of bits so that the empty string represents T , 0 represents the left subtree of T whose right subtree is 0.1, etc.

At any time, there are a number of worker agents, each assigned a distinct node m in T ; we say that m is "marked" with a. The idea is that that agent a with node m is performing the search problem associated with m. We will maintain the property that the set of marked nodes, plus a selection of nodes which are "closed", i.e.: search below them is complete, forms a *frontier* for T . This ensures that the remaining search space is covered by the available worker agents.

An agent a, using its unique identifier id, starts by volunteering its service to the master agent. It then awaits the assignment of a node identifier m so that it can use the following program segment in figure 9.5 to start search on m. In what follows, we can extract from m its associated decision variable $x[i]$; the length of m is denoted $|m|$. The reactor `fail` returns `TRUE` when some solution (published by any agent) is found, and the reactor `succeed(n)` publishes a solution. The accessing of marks below is to be thought of as being updates to the store which contains a database of the current marks. We assume without further ado that this agent injects into the store a failure notification when it has finished its search unsuccessfully. A final but important assumption: the update $\texttt{softtell}_{\mathcal{C}_0}\texttt{(id, x[i] = 0)}$ below is a satisfiability test of the formula $\mathcal{C}_0 \wedge C_m \wedge X_i = 0$ where $\mathcal{C}_0$ is the basic constraint, C_m are all the equations of the form $X_j = 0$ or $X_j = 1$, $1 \leq j < i$ corresponding to the node m. Similarly for the other update $\texttt{softtell}_{C_0}$ `(id, x[i] = 1)`.

```
solve(id, m) {
   if (fail) exit();
   if (|m| > n) {succeed(m); exit();}
   if (softtell_C0(id, x[i] = 0))
      if (!solve(id, m.0)) retract(id, x[i] = 0);
   if (mark(id) == m) move mark(id) to m.1;
   if (mark(id) is not above m) exit();
   if (softtell_C0(id, x[i] = 1))
      if (!solve(id, m.1)) retract(id, x[i] = 1)
   return FALSE
}
```

Figure 9.5: Worker Agent Solve Code

The master agent can be simply described as a multi-threaded process which accepts vol-

unteers in one thread, and dispatches a job by assigning a marked node to a volunteer agent in the other thread. In a separate thread, it book-keeps which amongst the workers have not yet failed, and the last thread simply listens for a solution found.

In the master agent program fragment in figure 9.6, we assume that `assign_work(id)` eventually causes the agent code in figure 9.5 to be called, with a suitable node m. Call this newly volunteered agent a'. In order to determine such m, the master maintains a list of current assignments (a, m_a) indicating that the currently active worker a is responsible for the subtree rooted at m_a. If the list is empty, then we are just starting and so the incoming agent gets the whole tree. Otherwise, randomly choose one current assignment (a, m_a) and (a) re-assign (a, $m_a.0$), and (b) assign (a'; $m_a.1$). That is, "split" the assignment of a and give half of it to a'.

Note below that the actions of volunteering and then being assigned a node is performed atomically. The reactor agent `failed(id)` simply records that agent *id* has failed. The reactor checks all failed makes the flag done available when all agents have failed.

```
repeat
   volunteer(id) ⇒ assign_work(id)
      &
   agent_failed(id) ⇒ check_all_failed
      &
   succeed(m) ⇒ done
until done
```

Figure 9.6: Master Agent Code

There are some subtle properties required on marks and the current search position of agents in order for this system to work. First, note that when an agent is just about move from the left subtree of its marked node to the right, that it moves the mark along with it. The logic here is that the left subtree is now closed, and so the mark can be moved deeper down the tree. More importantly, it means that when the master agent splits a node, it will be the case that the current agent working under that node will in fact be in the left subtree of the node.

Finally for this example, we observe that a practical challenge is to contain the bottleneck caused by the $\texttt{softtell}_{\mathcal{C}_0}$ updates needed by the agents as they search. In practice, if each agent could access a local constraint system which initially holds the base constraints, and which subsequently handles the constraint solving associated with the $\texttt{softtell}_{\mathcal{C}_0}$ updates, then the only communication between a worker agent and the outside world would be the (far less frequent) access of marks. We will discuss more of technique like this in chapter 10.

9.3 Multi-Agent Simulation

9.3.1 Introduction

"A simulation is a system that represents or emulate the behavior of another system over. In a computer simulation, the system that does the emulating is a computer program." [Fuj00]. Contrary to conventional mathematical simulation which is modeling based on a set of equations, agent-based simulation is models the system of individual entities by a set of agents. This type of simulations is particularly useful in the domains where the models are dominated by discrete decision makings and characterized by high degree of distribution, such as computational economics [Tes02, SBD02], the study of social behavior [Rey87], biology [NAH98]. While simulations in these areas can be modeled by multiple (intelligent or reactive) agents, concurrency and real-time elements are often not critical. In this section, we are considering the multi-agent simulation of war games, which is highly reactive and dynamic. War game simulations are a popular way to exercise military strategy and troops management without resorting to life firing. Such simulations are practiced frequently in defense institutes and arm forces around the world.

9.3.2 Battlefield simulation

In a war game simulations, various types of combat units are deployed on a possibly complex terrain to carry out certain mission. The combat units can be troops on foot, tanks, gun ships, with different capabilities and firing power. There is one characteristic they share in common: their vision radius and firing range are always limited, and most of the time, much smaller than the dimension of the entire battlefield. For example, if one bomb is dropped at location $(x,\ y)$, not everyone on the field will be killed. Instead only soldiers within certain radius of the explosion are likely to be killed. The other observation is this example is inherently concurrent and asynchronous. All units can move on their own but every move can potentially affect some one else. Apparently a sequential algorithm will not work here simply due to the prohibitive amount of computation. Furthermore, combat units can be modeled as intelligent agents whose actions are modeled as logic rules. This is where the store in OCP comes in handy. In summary, above mentioned features of a war game simulation make it a sound candidate for an OCP application. For more information on computerized war games, please refer to [Air].

9.3.3 The Pick-A-Mushroom game

In this section, we introduce a simple "Pick-a-mushroom" game as a simplified version of the war game simulation. It possesses most of the characteristics we introduced in the last section.

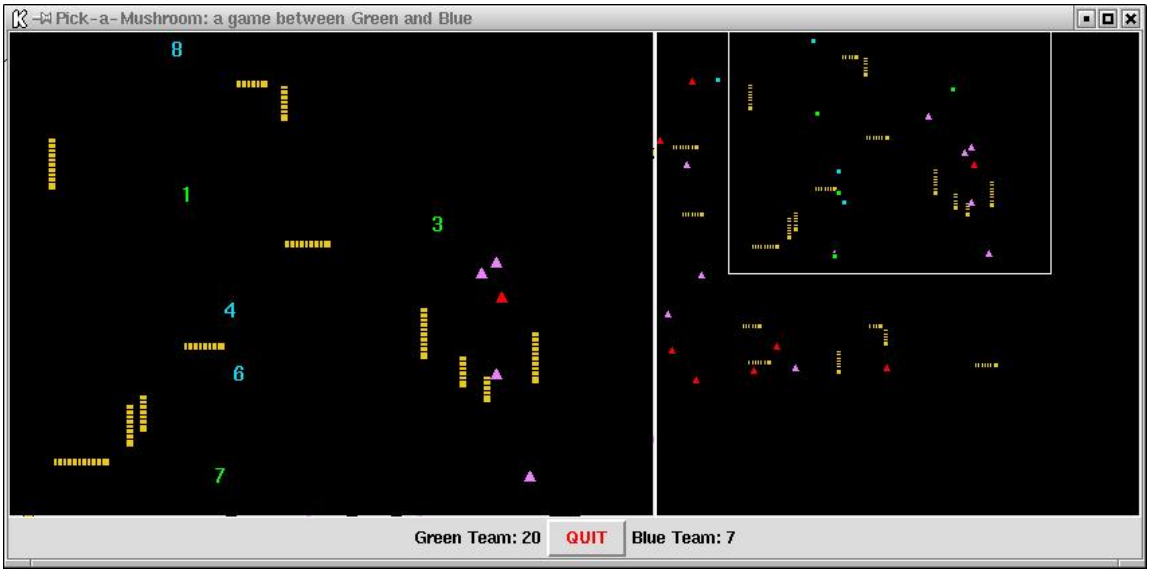


Figure 9.7: The Pick-a-Mushroom Game Console

Figure 9.7 shows the game console of "Pick-A-Mushroom". The objective of the game is for two teams of players to compete against each other by picking the mushrooms on the ground. Every mushroom comes with a score, and picking a mushroom earns the score. The team that finishes with the higher score wins.

The "playground" is an $N \times N$ blackboard depicted in the right panel of figure 9.7. The blow-up area within the zoom window is shown in the left panel. A cell on the playground is a triplet (x, y, id), where x and y are the coordinates of the cell and id is the id of the object that is currently occupying the cell. $id = 0$ if the cell is unoccupied. At any time, one cell only be occupied by at most one object (a mushroom, a blockade or a player).

A mushroom is indicated by a small red or a violet triangle. The mushrooms comes with two different sizes. The red mushrooms are bigger than the violet mushrooms. The mushrooms randomly appear or "grow" on the playground, and will remain there until they are picked by some players.

The players come in two teams, marked by distinctly by green or blue numbers, are initially randomly deployed on the playground. The number is the player's unique id. All players are limited by the uniform radius of vision $R_v \leq N$. In other words, a player cannot "see" anything

(including mushrooms, blockades and other players) beyond the distance of R_v. If a mushroom m is within the vision of a player p, m is said to be in the locality of p. A player can only move to east, south, west or north by one cell at a time. If a player moves onto and occupies a cell which is currently occupied by a mushroom, the player is considered to have "picked" the mushroom, and the score attached to the mushroom is given to the team to which the player belongs. For every player on the ground, if there is at least one mushroom in its locality, it should move toward that mushroom and try to pick it.

The blockades are indicated as yellow rectangles on the playground. The movement of the players are constrained by the blockades and also other players. If a blockade exists on the path between a player and a mushroom within locality, the player should move around the blocks and access the target. And the direction of movement of the player is calculated dynamically at every step.

We now implement two different strategies for the two teams of players. Let blue team always try to pick the *nearest* mushroom within their localities and the path to which is not blocked; and green team always pick the *biggest* mushroom within their localities and the path to which is not blocked. Figure 9.8 shows the basic reactors used by the two teams of players. Note the path is not a straight line from the player's position to the target as the player can only move in a "zigzag" fashion. If the nearest (or the biggest) mushroom is blocked, the next nearest (or the biggest) mushroom is the locality is attempted.

Blue Player:

(`canmove`(id, $east$) $\Rightarrow$ `move_to_nearest`(id, -1, 0)) ||
(`canmove`(id, $west$) $\Rightarrow$ `move_to_nearest`(id, 1, 0)) ||
(`canmove`(id, $north$) $\Rightarrow$ `move_to_nearest`(id, 0, 1)) ||
(`canmove`(id, $south$) $\Rightarrow$ `move_to_nearest`(id, 0, -1))

Green Player:

(`canmove`(id, $east$) $\Rightarrow$ `move_to_biggest`(id, -1, 0)) ||
(`canmove`(id, $west$) $\Rightarrow$ `move_to_biggest`(id, 1, 0)) ||
(`canmove`(id, $north$) $\Rightarrow$ `move_to_biggest`(id, 0, 1)) ||
(`canmove`(id, $south$) $\Rightarrow$ `move_to_biggest`(id, 0, -1))

Figure 9.8: Players Reactor

And inside the central knowledge base, we have the following definition for `canmove`, `move_to_nearest` and `move_to_biggest`. We assume every cell on the playground is represented by `matrix(x, y, id)`.

```
%Rules for sustain condition of whether the next step can be made
freespot (X, Y):- ID>0, matrix(X, Y, ID), fail.
freespot(_X, _Y).
canmove(ID, ea):-matrix(X, Y, ID), freespot(X+1, Y).
canmove(ID, we):-matrix(X, Y, ID), freespot(X-1, Y).
canmove(ID, no):-matrix(X, Y, ID), freespot(X, Y+1).
canmove(ID, so):-matrix(X, Y, ID), freespot(X, Y-1).

%Rules to check if a target is within the locality and path not blocked
through(TX, TY, TX, TY).
through(X, Y, TX, TY):-
        checkdir(X, Y, TX, TY, SX, SY),
        freespot(X+SX, Y+SY),
        through(X+SX, Y+SY, TX, TY).

target_in_locality(PX,PY,X,Y):-
        X==PX, Y==PY, fail.
target_in_locality(PX,PY, X,Y):-
        visionradius(R),
       (X-PX)*(X-PX)+(Y-PY)*(Y-PY)<=R*R.
        through(PX, PY, X, Y).

%Rules to find the nearest target location (TX, TY)
nearest_target(X, Y, TX, TY):-
        maxnumtargets(M), boardsize(BS),
        rec_targets(-M, 2*BS, -1, -1, X, Y, TX, TY), !.
rec_targets(TID, Min_Dist, TempX, TempY, X, Y, TX, TY):-
        TID<0, matrix(TXX, TYY, TID),
        abs(TXX-X, XX), abs(TYY-Y, YY), Dist=XX+YY,
        (Dist<Min_Dist, target_in_locality(X, Y, TXX, TYY)->
                rec_targets(TID+1,Dist, TXX, TYY, X, Y, TX, TY);
                rec_targets(TID+1, Min_Dist, TempX, TempY, X, Y, TX, TY)
        ).
rec_targets(TID, Min_Dist, TempX, TempY, X, Y, TX, TY):-
        TID<0, rec_targets(TID+1, Min_Dist, TempX, TempY, X, Y, TX, TY).
rec_targets(0, _, TempX, TempY, _, _, TempX, TempY):-
        TempX>=0.

%Rules to find the biggest target location (TX, TY)
biggest_target(X, Y, TX, TY):-
        maxnumtargets(M), boardsize(BS),
        big_helper(-M, 2*BS, 0, X, Y, -1, -1, TX, TY), !.
big_helper(0, _, _, _, _, TempX, TempY, TempX, TempY):- TempX>=0.
big_helper(TID, Min_Dist, MaxWt, X, Y, TempX, TempY, TX, TY):-
        TID<0,
        weight(TID, W),
        W>MaxWt, matrix(TXX, TYY, TID),
        (target_in_locality(X, Y, TXX, TYY) ->
                abs(TXX-X, XX), abs(TYY-Y, YY), Dist=XX+YY,
                big_helper(TID+1, Dist, W, X, Y, TXX, TYY, TX, TY);
                big_helper(TID+1, Min_Dist, MaxWt, X, Y,
```

```
                TempX, TempY, TX, TY)).

big_helper(TID, Min_Dist, MaxWt, X, Y, TempX, TempY, TX, TY):-
        TID<0,
        weight(TID, W),
        W==MaxWt, matrix(TXX, TYY, TID),
        abs(TXX-X, XX), abs(TYY-Y, YY), Dist=XX+YY,
        (Dist<Min_Dist, target_in_locality(X, Y, TXX, TYY) ->
                big_helper(TID+1, Dist, W, X, Y, TXX, TYY, TX, TY);
                big_helper(TID+1, Min_Dist, MaxWt, X, Y, TempX, TempY, TX, TY)).
big_helper(TID, Min_Dist, MaxWt, X, Y, TempX, TempY, TX, TY):-
        TID<0, big_helper(TID+1, Min_Dist, MaxWt, X, Y, TempX, TempY, TX, TY).

%Action rules to move to the nearest mushroom
move_to_nearest(ID):- ID>0, matrix(X, Y, ID),
        nearest_target(X, Y, TX, TY),
        moveto(ID, X, Y, TX, TY).

%Action rules to move to the biggest mushroom
move_to_biggest(ID):- ID>0, matrix(X, Y, ID),
        biggest_target(X, Y, TX, TY),
        moveto(ID, X, Y, TX, TY).
```

9.4 Real-time Dynamic Vehicle Routing

This section presents the work on applying OCP system to real time dynamic vehicle routing. The content of this section is largely adopted from [ZO00].

9.4.1 Introduction

Vehicle Routing Problem with Time Windows (VRPTW) [TLZO01] is a well-known NP-hard problem. The objective of the problem is to find routes for the vehicles to service all the customers at a minimal cost (in terms of travel distance, etc.) without violating the capacity and travel time constraints of the vehicles and the time window constraints set by the customers. Almost all VRPTW methods proposed are devoted to a static problem as the problem parameters and constraints have to be known in advance. Any change in these data requires a complete re-calculation. Changes after the vehicles are deployed are forbidden. Until now, little research has been focused on the dynamic VRPTW, where problem size and parameters change after the vehicles are already commissioned. Some of the earliest work on dynamic vehicle routing problem was from Bertsimas and Ryzin [BR91] which is essentially a generic mathematical model with the waiting time as an objective function. Some further work can be found in Powell,

Jaillet and Odoni [PJO95], Psaraftis [Psa95] and Gendreau [GGPT99]. The emphases were placed on the problem modeling, solving approaches and algorithms, and not on the real time concurrency and reactivity issues. This section proposes deal with these issues within the OCP framework. We also differ from the previous work in the problem modeling as well as objective function construction: we adopt the traditional VRPTW and maintain travel distance as the objective function.

In our research, we describe a *Real Time Dynamic Vehicle Routing Problem (RTDVRP)*, an extension to traditional VRPTW, whose problem parameters change in real time when vehicles are already commissioned. The problem is reactive in nature, and its solution benefits from the concurrent, reactive framework of the OCP. Here we view vehicles as distributed agents that communicate with each other through a common knowledge base with VPRTW solving capabilities. Vehicles communicate with the knowledge base through OCP reactors with timeout. The relationship between vehicles can be represented by constraints as one vehicle's breakdown or slowdown can affect other vehicles' designated routes. The VRPTW solver uses an *incremental local optimization (ILO)* algorithm, with states stored persistently in the knowledge base. It solves the VRPTW on a dynamic basis without solving the entire problem, hence it is quick and effective.

In this paper, we first introduce the concept of RTDVRP. Then we present *Reactive Vehicle Routing System (RVRS)*, an implementation of RVR infrastructure and a simulation based on OCP.

9.4.2 Description of RTDVRP

The traditional VRPTW is given by a set of identical vehicles, a special node called the depot, a set of customers to be visited, a directed network connecting the depot and the customers [TLZO01]. A route is defined as starting from the depot, going through a number of customers and ending at the depot. A cost d_{ij} and a travel time t_{ij} are associated with each arc of the network. Every customer in the network must be visited only once by one of the vehicles. The objective is to serve all customers with minimum vehicles and travel distance without violating time window and capacity constraints. In reality, traffic conditions change dynamically, causing initial best assignment to be invalid. For example, one vehicle may break down and another vehicle has to change its route to substitute for the breakdown vehicle. In addition, the time window of a customer can be changed on the fly, so original routes may not apply any more.

RTDVRP is an extension to VRPTW with two key differences: (1) In RTDVRP, traveling time between nodes changes dynamically due to the change of traffic condition; (2) In VRPTW, all vehicles depart at time zero, whereas in RTDVRP, there are some dynamically appointed vehicles that have differing departure times.

The model of RTDVRP

$$
\begin{aligned}
M &= \text{number of customers} \\
K &= \text{number of vehicles} \\
\text{Triptime} &= \text{travel times allocated to each vehicle} \\
C_k &= \text{maximum load of vehicle } k = C \\
d_{ij} &= \text{distance from node } i \text{ to node } j \\
t_{ij} &= \text{travel time from node } i \text{ to node } j \\
m_i &= \text{demand of customer } i \\
\text{TReady}_i &= \text{earliest serve time of customer } i \\
\text{TDue}_i &= \text{latest serve time of customer } i \\
\text{Service}_i &= \text{service time at customer } i \\
\text{Next}_i &= \text{successor of node } i \\
\text{TServe}_i &= \text{time start to serve customer } i \\
\text{Load}_i &= \text{load of vehicle when it departs from } i \\
\text{Customers} &= \{x : \mathbb{N} \mid 1 \leq x \leq M\} \\
\text{Depot} &= \{x : \mathbb{N} \mid M+1 \leq x \leq M+K\} \\
t_{ij} &= \{i, j \in \text{Customers} \cup \text{Depot} \mid \forall i, j \bullet \exists \tau \in \mathbb{N}_1 \bullet t_{ij} = \tau \times d_{ij}\} \\
\text{Time} &= \{x : \mathbb{N} \mid 0 \leq x \leq \text{Triptime}\} \\
\text{Next} &= \{(i, x) : i \in \text{Customers};\ x \in \text{Customers} \cup \text{Depot} \mid \forall i, j \in \text{Customers} \\
&\quad \bullet (i \neq j) \wedge (\text{Next}(i) \neq \text{Next}(j)) \wedge (\text{Next}(i) = j \Rightarrow \text{Next}(j) \neq i) \wedge (\text{Next}(i) \neq i)\} \\
\text{Next} &= \{(i, x) : i \in \text{Depot};\ x \in \text{Customers} \mid \forall i, j \in \text{Depot} \bullet (i \neq j) \wedge \\
&\quad (\text{Next(i)} \neq \text{Next(j)})\} \\
\text{TServe} &= \{(j, x) : j \in \text{Customers};\ x \in \text{Time} \mid (\text{TReady}_j \leq x \leq \text{TDue}_j) \wedge \\
&\quad (x \leq \text{Triptime} - t_{j(\text{Depot})} - \text{Service}_j) \wedge (\forall i, j \in \text{Customers} \bullet \text{Next}(i) = j \Rightarrow \\
&\quad \text{TServe}(j) \geq \text{TServe}(i) + t_{ij} + \text{Service}_i\} \\
\text{Load} &= \{(j, x) : j \in \text{Customers};\ x \in \mathbb{N} \mid (0 \leq x \leq C) \wedge (\forall i \in \text{Customers} \cup \text{Depot}; \\
&\quad \forall j \in \text{Customers} \bullet \text{Next}(i) = j \Rightarrow \text{Load}(j) = \text{Load}(i) + m_j)\}
\end{aligned}
$$

9.4.3 Vehicle agents and reactors

The vehicle agents are independent of each other once dispatched and they are self-controlled in the sense that they only follow the original routing plan given by the central server and next

destinations given by their own reactor programs in case of traffic situation changes. Every vehicle is reactive because they issue a number of reactors ranging from simple updates of vehicle location and traffic cost to more sophisticated residual query of next moves as a result of some cost update from any vehicle agent. In the RVR system, we will have the following reactors:

Solve a new problem:
`true` $\Rightarrow$ *solve_vrptw*

Add new vehicles:
repeat `more_vehicles_needed`(NV) $\Rightarrow$ `dispatch`(NV)
until `all_customers_visited`

Next move
repeat `new_solution`(id) $\Rightarrow$ `next_move`(id, C)
until `all_customers_visited`

Update location:
`true` $\Rightarrow$ `update_current_customer`(id, c)

Update cost:
`true` $\Rightarrow$ `update_cost`($c1$, $c2$)

Figure 9.9: RVRS Reactors

The first two reactors are system reactors invoked by the RVR system to solve a brand new problem and to add vehicles if an updated solution requires more vehicles. The next three are issued by vehicle agents. The most important one is *next move*, as it updates the route of a vehicle (with id id) for a vehicle dynamically. *Update cost* is the only vehicle reactor that may change the solution in the database. It represents the situation of a traffic jam or vehicle breakdown. Together these five reactors forms the foundation for solving RTDVRP problems in a reactive setting.

9.4.4 VRPTW solver

The VRPTW solver is a special knowledge base used in the RVRS, which processes requests sequentially. A request can be solving for a particular VRP problem, updating the location of the vehicles, or reporting traffic condition, etc. The solver is also a data store that keeps the most up-to-date solution, current locations of the vehicles, the cost matrix, a *priority list* of customers to be visited, and various constraints. The solver adopts an incremental local

optimization routing scheme similar to that of Caseau's [CL99] to achieve an overall solution. The solution is formed incrementally, while local optimization is done along the processes of constructing full solutions.

In the beginning, all customers are arranged into a stack. They are then served in order, to form the routes incrementally. Each time the heuristic looks for the best location for a customer, with fewer vehicles and less travel distance. After allocation of one customer, a series of optimization is performed involving operations on routes aiming at reducing the number of vehicles and travel distances. The solver ensures that all changes during optimization do not violate the constraints. Figure 9.10 summarizes the ILO algorithm.

```
stack customers;
while customers not empty:
   begin
      cust = customers.pop_back()
      if(!insertion(cust, existing_routes)) //no solution
        newroute(cust)
      else
        optimize(existing_routes)
   end
```

Figure 9.10: Incremental Local Optimization

`optimize` involves 3-opt-like operations to be performed within a route and between routes in the routes that have been formed.

Whenever the travel cost on a route is modified (when traffic condition changes and is observed by some of the vehicle agents), the optimization with respect to the changed route is performed. As a result of swapping nodes between two routes, some other routes maybe affected by this optimization, and they are added to a queue, waiting to be further optimized. The optimization finishes when there is no more route in the queue. This heuristic is greedy, as only downhill moves are allowed, which potentially causes the search to be trapped in a local minimum. The heuristic handles this by inserting the customers according to their priorities. Higher priority customers are inserted first Two factors are taken into account when determining the priority:

1. The distance of the node from the previous node in the priority queue (the nearer the higher priority: tendency to form a shorter route), and

2. The node's due time (the earlier the higher priority: tendency to be able to be served in

time).

The two factors are combined with a weighting factor w, $w \in [0, 1]$. Thus the priority p of a customer j is defined by:

$$p(customer_j) = w \times d_{ij} + (1 - w) \times TimeDue_j,$$

where i is the customer that has just been inserted into a route. Note that the lower the value of p, the higher priority is the customer. This weighting factor need to be tuned to suit different problems, because it has sizeable affect in the quality of the solution. The heuristic is simple yet produces good results. Compared to one of the best heuristics for VRPTW by Rochat and Taillard [RT95] on the Solomon benchmark [Sol87], ILO's total distances are only 3% more in its worst case. The number of vehicles are the same for problem set C1, C2, R2 and RC2 and just one more for problem set R1 and RC1.

Re-routing is required when a vehicle encounters a traffic jam and is unable to complete the route without violating constraints. The re-routing only applied to those unserved customers. Furthermore, in order to avoid disturbances to other vehicles, the immediate destination of the vehicles are not reallocated.

9.5 ANTS Meeting Scheduler

In this section, we present an implementation of a meeting scheduling system called ANTS by OCP. The ANTS system has a distributed calendar interface which can be used in many organizations. The content in this section is adopted from [ZS02].

9.5.1 The meeting scheduling problem

A static event scheduling problem can be defined as: *given a set of variable constraints on participants, resources and meeting time window, produce a schedule of events that achieve collective satisfaction among all the participants before pre-defined deadlines.* Mathematical model is as follows.

Let

$$\begin{aligned}
N &= \text{Number of open events} \\
e_i &= \text{An event, where } 0 \leq i \leq N-1. \\
clock &= \text{Current wallclock time.} \\
t_i &= \text{Scheduled time for event } i. \\
ws_i &= \text{Start of schedule time window for event } i. \\
we_i &= \text{End of schedule time window for event } i. \\
P_i &= \text{Set of proposed participants of event } i. \\
R_i &= \text{Set of required resources of event } i. \\
n_{r,i} &= \text{Required number of resource } r \text{ for event } i. \\
q_{r,t} &= \text{Quantity of resource } r \text{ at time } t. \\
u &= \text{User.} \\
wpref_{t,u} &= \text{(Weighted) preference of user } u \text{ for time slot } t.
\end{aligned}$$

$$\textbf{Maximize} \qquad \sum_{i=0}^{N} f(i) \tag{9.1}$$

where the function f is given by:

$$f(i) = \begin{cases} 0 & \text{if } \exists\ u \in P_i,\ wpref_{t_i,u} = 0 \\ avg(i) & \text{otherwise} \end{cases}$$

Further,

$$avg(i) = (\sum_{u \in P_i} wpref_{t_i,u})\ /\ |P_i|.$$

s.t.

$$\begin{aligned}
&\forall i \in [0, N-1],\ clock < we_i \\
&\forall i \in [0, N-1], ws_i \leq t_i \leq we_i \\
&\forall (i, r, n) \in R_i,\ q_{r,t_i} \geq n_{r,i}
\end{aligned}$$

Users may specify special constraints, such as the event should not have an unfavorable participants or that the facilities must suit their preferences. Hence we may add

$$\begin{aligned}
&\forall (i, u \in P_i)\ \neg(\exists r \in R_i,\ r \in RD_u) \\
&\forall (i, u \in P_i)\ \neg(\exists v \in P_i, v \neq u,\ v \in PD_u)
\end{aligned}$$

to the list of constraints, where

$$\begin{aligned}
PD_u &= \text{Set of participants unfavored by user } u. \\
RD_u &= \text{Set of resources incompatible with } u.
\end{aligned}$$

Such user-defined constraints can be added or removed dynamically.

Our problem is a dynamic meeting scheduling problem because user-defined constraints and their preferences can be added, removed or changed dynamically. In a dynamic system, some or all of the above variables may change before the end of the time window. Each such change will trigger a constraint solving procedure in the store.

9.5.2 Knowledge base

The ANTS store is implemented by a CLP($\mathcal{R}$) program, therefore the system is completely described by facts, rules and constraints. Conceptually, the CLP store has the following elements or base predicates (facts):

- **Users.** The members of the work group who share the calendar system. Users may be ranked in a hierarchical organization so that higher ranking users have their preferences prioritized. Here the users are categorized in three priorities. Users can be added or removed from the group.

- **User preferences.** Matrices of users' preference for events. All the preferences are real numbers between 0 and 1, where 0 represent "blackout" for that slot and 1 shows a free slot. The size of this part of the knowledge base is $U \times E \times S$, where U is number of users, E is number of events currently being scheduled and S is number of time slots. The number of slots for each event is 10×14, or ten hours a day from 8 AM to 6 PM, 7 days a week for 2 weeks, in the implementation of ANTS . For example, the list of preferences of users "Kenny" and "Weeyeh" is stored in the server as:

```
pref(kenny, pingpongmatch,
    [1.0, 1.0, 0.5, 0.2, ..., 0.3, 1.0]).
pref(weeyeh, pingpongmatch,
    [0.8, 0.8, 1.0, 0.7, ..., 1.0, 0.5]).
```

- **Public resources.** Resources required for the events like space, equipment, human (people not in the work group), etc. The resources are limited and countable. Each public resource has two-week availability time table that aligns with the proposed event's time window. For example, if an event "Ping Pong Tournament" has a scheduling time window from 8 AM Monday, June 4 to 6 PM Sunday, June 17, and the resource needed

are court (1), ping pong table (4), ping pong balls (24), then the availability list of these resources for that period of time (140 slots) will be sent to the store by the *admin*, in the following manner.

```
resource(court, pingpong,
    [0, 0, 1, ..., 1, 1, 1]).
resource(pptables, pingpong,
    [2, 2, 2, ... 6, 6, 6]).
resource(ppballs, pingpong,
    [50, 50, 50, ... 20, 20, 20]).
```

Public resources associated with an event will be removed from store if the event has been successfully scheduled or canceled.

- **Public events.** These map to the time slots. And they are the union of all users' public events. If set PE_u is the set of public events of user u, the global public events in the store is a set

$$GE = \bigcup_{u \in P_i} PE_u, \tag{9.2}$$

where P_i is the set of participants of event i. Each public event is identified by an *event identifier*. If the event is successfully scheduled by the deadline, it will be marked as a confirmed event, otherwise this predicate is eliminated from the store. Event e_i is defined as having: *title of event*, *proposed participants*, *required resources*, *deadline of submission*, and *schedule time window*. For example, some meeting `pingpongmatch` can be represented as the following in the store:

```
event(pingpongmatch,        % Event identifier
    '25/02/2001/08',           % Starting time
    '10/03/2001/17',           % Deadline
    '05/03/2001/10',           % Suggested time
    [kenny, weeyeh, roland],   % Participants
    [res(OHP, 1),
    res(large_room, 1)]).      % Resources
```

- **Bundles.** Bundles are collections of time slots with practical meaning to a user or a group of users, e.g. "all mornings", "every Tuesday afternoon", "lunch time", etc. Having bundles is an easy way to help users specify their special preferences, for example, the user may want to specify:

`not(morning(`*time*`))`

as the constraint he or she submitted in order to state that he or she does not want to attend an event if it is held in the morning.

- **Date and time.** These are important factors to watch as the system has to finish scheduling an event before a due deadline. The current date and time is constantly observed by the store.

- **Optimization rule and constraints.** All users' preferences are weighted according to their importance of either 1, 2, or 3, from the lowest to the highest. If a user u's importance is σ_u, then the weighted preference for time slot t is

$$wpref_{t,u} = \frac{3 - \sigma_u}{3} + \frac{\sigma_u \times pref_{t,u}}{3}. \tag{9.3}$$

  For large problems, the time for finding the absolute maximum value in (9.1) is prohibitive, therefore we use a probabilistic approach that solves for a few solution and pick a best one out of them.

9.5.3 User reactors

There are three *roles* in ANTS , namely *normal users*, *hosts* (or *participants*), and *administrator*. Note that a person can have multiple roles at the same time, e.g., both an event's host as well as participant.

Participants and negotiation

User's calendar consists of time slots. Each time slot refers to a specific one-hour span. Users attach a description of his or her activity, and the corresponding *priority* to each time slot. Priority states how much the user prefers to attend that event in the time slot. If a time slot is empty, it carries a default priority value until the user changes it explicitly. A sample user's calendar is shown in figure 9.11.

Priority $p_{t,u}$ of a time slot t of a user u is an integer from 0 to 10. The reverse of this priority which we call *preference* is defined as

$$pref_{t,u} = 1 - \frac{p_{t,u}}{10}. \tag{9.4}$$

Intuitively,

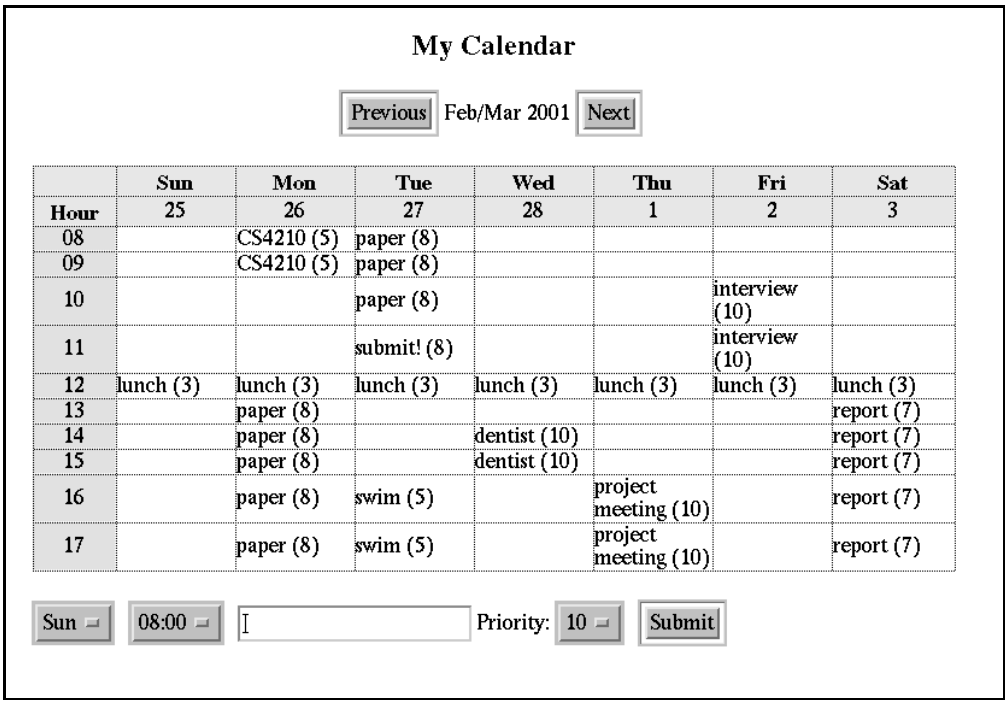


Figure 9.11: User's Calendar

$$\forall t, u, \quad pref_{t,u} \in \{0.0, 0.1, \ldots, 1.0\}.$$

A preference for a particular time slot specifies how much the user prefers a public event to be held in the time slot.

The process of negotiating the schedule of an event starts as the host proposes an event and invites some users to join. The participants send replies to the host containing information on busy/free time on its calendar and the priority of the events occupying the time slots:

repeat
  (`invited`$(event, participants), user \in participants) \Rightarrow$
  (`update_preferences`$(user, preferences)$;
  `update_constraints`$(user, constraints)$)
until `false`

Users may also change their preferences and constraints after the initial invitation if they have changed their mind about a previous submission or a new private event has been added to the calendar, or they don't like the tentative schedule produced by the system:

`true` $\Rightarrow$
(`set_preference`$(user, preferences)$;
`set_constraints`$(user, constraints)$)

Some reactors are used to mark the time slot on user's calendar when an event has been scheduled. The schedules are always tentative and subject to change if any other participants changes their mind:

repeat
  `tentative_schedule`$(event, time) \Rightarrow$
  `mark_calendar`$(event, time)$
until (`deadline`$(event, d), clock > d$)

In this reactor, the user's calendar will be marked with tentative schedule. And if a viable solution is obtained by the deadline, the event is marked permanently on the user's calendar.

Our approach reduces the iteration of proposing and replying between host and participants since the host will likely to be very informed about the participants' schedule. Although some researchers suggest meeting scheduling to be based on communication of equal participant agents [SD98, BPH+90], we supports the use of centralized repository which is an agent with different role. This reduces communication and in return also simplifies negotiation protocol.

Event's host

The host is a special user that is proposing a public event, who may or may not be one of the participants. He or she has to declare the event's time window, suggested participants, required public resources, and a suggested time slot. The host submits the following reactor to the server in order to propose an event:

`true` $\Rightarrow$ `propose`$(event,\ timewindow,\ participants,\ resources)$

He has the right to favor certain participants by giving their preferences higher priority, so that their weighted preferences are higher. Note that the host does not need to specify the deadline since it will be assumed that the deadline is two weeks from the starting date.

After all participants have submitted their preferences and constraints, the solver will present to the host the average preferences of all the time slot within the time window, along with a best solution, if any, in the form of a preferences table, as shown in figure 9.12. White denotes maximum preference, and the color becomes darker as the preference decreases. The highlighted spot is the best solution for that event.

The host reserves the right to cancel the public event if it is no longer needed, or if the

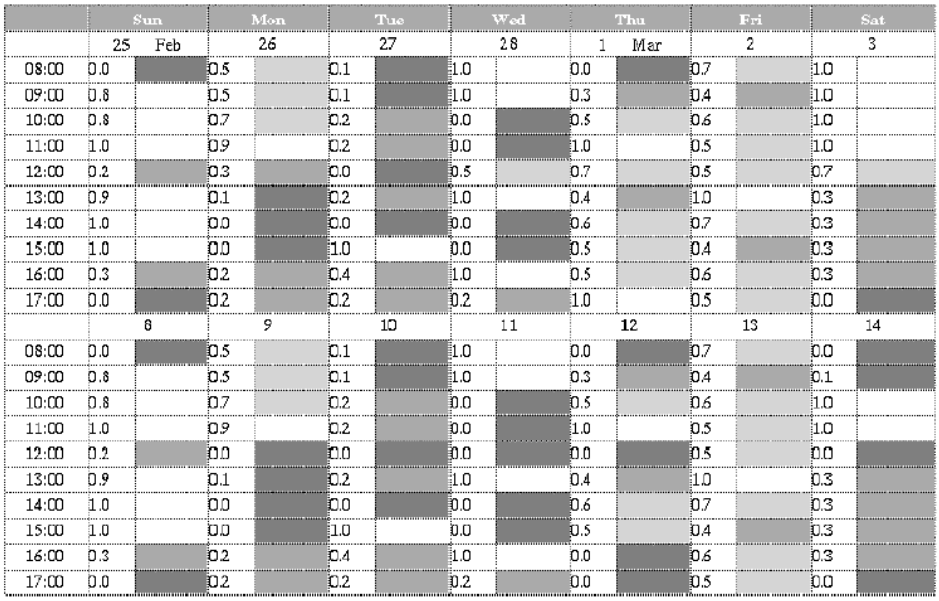


Figure 9.12: Preferences Table

schedule produced by the system is not satisfactory. He, however, cannot force the participants to attend the event at a time he or she likes. If the resulting schedule of the event is not desirable to him, he or she could black out this time slot as a participant to the event, and trigger a re-scheduling. He or she can also submit constraints to bias the solution from ANTS . The system is very flexible to cope with changes in such requirements as well. By entering different constraints using the user interface, the behavior of the system can be altered easily.

The following reactor is used by the host to notify the participants of a new schedule solution.

repeat
  `new_schedule`$(events, times)$, $event \in events$, $time \in times \Rightarrow$
  `tentative_schedule`$(event, time)$
until `false`

Administrator

The administrator is in charge of scheduling for all "live" events whenever any user preferences, constraints or required resources are updated. The following reactor sustains on such changes:

repeat
((`change_resource`$(r),\ r \in resources \Rightarrow$
`re_schedule`$(open\_events,\ times)$
||
`change_preferences`$(user,\ preferences),\ user \in participants) \Rightarrow$
`re_schedule`$(open\_events,\ times)$
||
`change_constraints`$(user,\ constraints), user \in participants) \Rightarrow$
`re_schedule`$(open\_events,\ times)$)
until (`deadline`$(event, d),\ event \in open\_events,\ clock > d$)

The administrator also controls the availability of public resources by:

`true` $\Rightarrow$ `update_resource`$(resource,\ newschedule)$

Chapter 10

Towards a Distributed OCP System

This chapter discusses some preliminary ideas of a distributed OCP system. We propose to distribute the OCP shared store to many locations in order to enjoy better locality. As a result, new challenges emerge, and this chapter addresses some of these challenges.

10.1 Introduction

Our first OCP prototype system (chapter 8) features one reception unit and one central store, and all reactors running in the system are completely serializable and evaluated in a sequential manner. While this may be sufficiently good for some small applications, it is definitely not scalable when it comes to large applications connected to thousands of agents. For example, a stock market portfolio management software for the whole stock exchange expects tens of thousands of reactors to be submitted at the same time. Each of these reactors requires computation cycles and memory allocation in general and they are likely to stay in the system for substantial amount of time due to blocking. Thus it is compelling to introduce *parallel processing* to the implementation of OCP. In addition, when user bank accounts are linked to this portfolio management system, it is highly undesirable to store everybody's confidential information all together on one server. After all, much of this information is naturally distributed and protected locally. Therefore OCP system would be much more useful if it offers *distribution of data*.

One naive solution to the bottle neck of a single reception unit is to have multiple receptionists running in parallel. Although this may improve the efficiency of reactor input and output, it doesn't offer better overall throughput of the reactors because ultimately we only have one CLP($\mathcal{R}$) program and all goals are run sequentially. What is really desired is to have multiple OCP servers each with a CLP($\mathcal{R}$) system and a piece of CLP($\mathcal{R}$) code running concurrently, and agents can interact with different OCP server even though the distribution of the centralized

OCP server and store is *transparent* to the agents and the users.

In many applications, it is common to have reactors that require private predicates and don't interact with others reactors through shared predicates very frequently, therefore part or whole of the CLP program (the store) can be distributed to a local OCP server that serves a number of interested agents. As such, most of the processing can be done at the local OCP server until some shared information is needed. Only then the agent fetch that information from other OCP servers. In this respect, all the reactors are executing in true parallelism, while the semantics of the OCP framework remains unchanged. If the model and system we have presented in chapter 3 and chapter 8 are share-memory computing, then we are going to propose here in this chapter is a distributed shared memory (DSM) version of the OCP.

This chapter presents some preliminary ideas of a distributed OCP system, which extends the centralized OCP system we have described in chapter 8. We will first present the how to distribute CLP($\mathcal{R}$) programs in section 10.2, followed by the topology of interconnection of servers in section 10.3. Finally we discuss the concurrency control and triggering of the distributed OCP in section 10.4 and 10.5 before concluding the chapter.

10.2 Distribution of CLP($\mathcal{R}$) Program

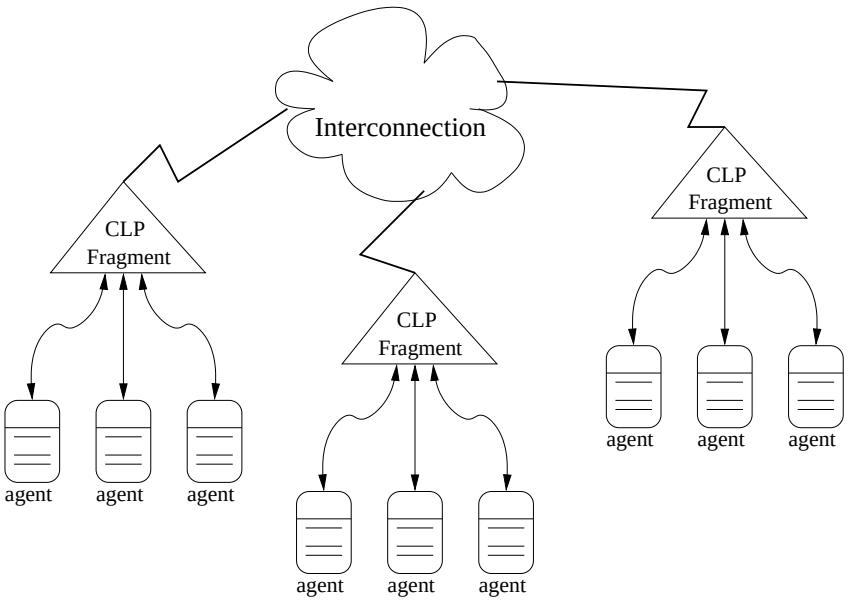


Figure 10.1: Distributed OCP scenario

Let us recall in chapter 8 which presents the OCP system as an agent based architecture with

a central OCP server which consists of a shared store and a registry that manages the reactors. In this system, we implement the store by a CLP($\mathcal{R}$) program. The agents synchronize with the store via conditions on the predicates in the CLP($\mathcal{R}$) program. In this model, all data are stored as predicates, and conditions and actions are CLP($\mathcal{R}$) goals. Both data storage and goal running are centralized at the server side. Our goal is to develop a computation model with concurrently running OCP servers, each with a fraction of the original CLP($\mathcal{R}$) program, so that reactors can be distributed to these servers and processed in parallel. Figure 10.1 depicts such a distributed OCP setting. This idea is analogous to the distributed database [OV99]. Essentially we are treating the CLP($\mathcal{R}$) program as a database, and the purpose of this section is to show how CLP($\mathcal{R}$) programs can be distributed that goals can be run in parallel. As a CLP($\mathcal{R}$) program is a collection of predicates, we first look at the types of predicates we have in any typical CLP($\mathcal{R}$) program.

System predicates System predicates are the common relations that govern the operation of the OCP system and its application. For example, predicates like `add_user`, `consult`, `append`, `findall`, etc. are considered system predicates. Some of these predicates, such as user management predicates are actually private to the administrator of the system. But this is a security issue which is orthogonal to our discussion here. System predicates are rules that can be used by anyone and they are not bound to change often. The arguments to these predicates are usually temporary or private and do not have global logical meanings. Therefore, these predicates and their code can be replicated at all local OCP servers, and all computation involving these predicates can be carried out locally. In rare cases when such rule is changed at one site (probably by the administrator of the system), it can be updated to all the connected servers using some existing update algorithms.

User predicates Most of these predicates are application-dependent facts that are defined for some or all agents in the system. They are both readable and updateable by the agents, e.g. `location(mobile_hq, X, Y)`, `money(johnsons, X_dollars)`, where the first predicate can be read or written by a group of war game agents in the same company, and the second accessible by all family members of the Johnsons. User predicates can also be locally defined rules that are applicable to only a small group of users. It is possible to fragment the user predicates and distribute them to different OCP servers.

The allocation of these fragments depends on the locality property of the application and user access patterns. For example, in a stock trading system, if we know the trading agents usually access stocks from the same industries frequently, then it makes sense to fragment the `stock_price` predicates according to industries. Furthermore, if agents that are currently interacting with a server repeated access or request some user predicates, it would be better if those predicates actually "live" at that server. User predicates are *shared* if they can be accessed by multiple OCP servers. Other user predicates which are only accessed by one OCP server where they reside are said to be *private*. Because of the volatile nature of the the user predicates, the distributed OCP system adopts the disjoint fragmentation of the CLP($\mathcal{R}$) program, so that data consistency is easier to maintain. Even so, concurrency control of the shared predicates are needed to ensure that data are consistent within a reactor, and the semantics of sustain, atomic transaction are preserved. To simplify matters, we assume that all tuples of the same predicate are stored in one single fragmentation.

Environment predicates These are special facts coming from the external environment, usually beyond the control of the users or system administrators. Examples of environment predicates are real time stock prices, trading volumes, movements of the enemy platoons, weather conditions, etc. Because these predicates often come with temporal properties and they are changing over time, somewhat similar to video streaming, they are also called *streaming predicates*. These data can be characterized as spontaneous, uncontrollable, observable but not changeable, and time-related. We assume that environment predicates can only be altered externally by a "magic hand" and are visible to all agents simultaneously. Because these predicates are for read-only, and we make no assumption about the synchronization of clocks in all servers, it is possible to make multiple copies of the streaming predicates to all servers. In the words, our architecture shall support the streaming feeds to all servers constantly, though no assumption is made about the timeliness of these streaming feeds. However for a single server, the data will come in sequence, so the order of arrival is preserved. Agent programs are free to store a history of the streaming data for historical analysis, or they can choose to disregard past data and focus on the current ones. The decision is made locally. However if a piece if data is lost, it's lost forever and irretrievable.

To summarize, system and environment predicates are duplicated to all sites, and we assume they are not to be changed by the agents, and hence they can be used as if they are just *local predicates*. User predicates are disjointly fragmented and distributed to different sites depending on the application. For any agent, a user predicate that is located in an OCP server other than the one it is currently communicating to is called a *remote predicate*. When a remote predicate is requested, it will be fetched from a remote server. For example, consider the following simple CLP($\mathcal{R}$) program:

```
q(5).
q(10).
p(X, Y):- q(X), r(Y), X<Y.
```

where `p/2` and `q/1` are predicates defined at store Δ_i, and `r/1` is a predicate defined at store Δ_j. Suppose we issue a CLP($\mathcal{R}$) goal at Δ_i:

```
p(X, Y).
```

At some point during the resolution, predicate `r(Y)` is requested. Realizing that this predicate is not local on Δ_i, the OCP server at Δ_i makes a global request and fetches `r(Y)` from Δ_j eventually and continue with the goal running.

At this point, we propose two ways to fetch a remote predicate. One is via *broadcast*, the other is with a *lookup table*. With broadcast, the local server sends a broadcast request for a predicate p to every other server on the interconnection. The server that has a local copy of p will respond with the a correct copy. With a lookup table, a server can readily check where a particular predicate is located on the network, and hence contact that server directly. The second method is less costly in terms of communication, but on the other hand is not as storage efficient as the first method because if the CLP($\mathcal{R}$) program is big, the lookup table can be very big too. In both cases, distributed backtracking such as in [FM87] may be required to ensure that the correct tuple is returned from a remote server during the backtracking.

10.3 Network Topology

We have thus far hidden the details of the interconnection of the distributed OCP servers in figure 10.1. In fact, the distributed OCP system we are presenting in this chapter does not impose any particular network topology. The decision of what type of connection to use is largely application specific. For example, if the application is inherently hierarchical, one may

fragment the program in a hierarchical way and distribute the fragments to a tree network of OCP servers. If an application involves a number of predicates that is frequently accessed by all agents, then it is the best to organized the servers in a star network and put the most frequently accessed predicated in the center server. Furthermore, if fault tolerance is a required feature, then having a ring topology makes sense as it makes failure detection and recovery easier. The interconnect may or may not involve a master control. To make the system more scalable, it is desirable, however, to have no master coordination of all the servers.

10.4 Concurrency Control

As we distribute the CLP($\mathcal{R}$) fragments to various OCP servers over the network, agents are interacting with different OCP servers in true parallelism. Thus reactors are also executed in parallel. As shown in section 8.2, every OCP reactor is decomposed into a primitive reactor before submitting to the OCP server. The primitive reactor is simply (*condition*, *action*) pair, where *condition* is the sustain condition, and *action* is an atomic update δ or an atomic transaction $\langle r \rangle$. Both condition and action are CLP($\mathcal{R}$) goals, and to be executed together atomically if condition is true. Unless otherwise noted, in the remainder of this section, we are only considering primitive reactors and just call them reactors for simplicity.

Since CLP($\mathcal{R}$) system is sequential, there is no interleaving between running of two independent goals. Therefore, if running both condition and action goals does not involve any remote access, i.e. all subgoals in the resolution process can be answered locally, consistency is always preserved, and there is no issue with concurrency. On the other hand, if one of the goals involves a remote access, and the values read or written are used again later in the same reactor, then the values must not be changed by other reactors. For example, if the execution of a reactor r consists of the following sequence of action.

$$rb(X, p(X));\ rb(Y, q(Y); X == Y;\ ub(p(X), p(X+1))$$

where p is a remote predicate and q is a local predicate. If there exists `q(3)` in the local CLP($\mathcal{R}$) program, then the expected outcome of this reactor is that `p(4)` is written to the remote store, and not any other value. This is because the operation semantics of the centralized OCP system dictates that every reactor is run atomically.

We generalize this example and give the following definition of the serializability of a dis-

tributed OCP system.

Definition 12 (Serializability) *Given a number of concurrent reactors, a distributed OCP system is serializable if the result of executing these reactors is the same as some serial execution of these reactors on a centralized OCP system.*

Here, the result of executing the reactors means the answers to the reactors as well the final state of the store (union of the fragments).

A remote access to a shared predicate whose argument values are used more than once in any reactor is called a *critical access*. From here onwards, we are only interested in critical accesses. And we use the word access to mean critical access unless otherwise stated. Naively, when a reactor makes a critical access, we can suspend all remote OCP servers until this reactor runs to completion. While this ensures serializability, the efficiency of the system is unacceptably low. Therefor the challenge is to allow maximum interleaving when a shared predicate is critically accessed by concurrent reactors while preserving data consistency of sustain and transaction in reactors.

To achieve serializability of a distributed OCP system, we propose the two kinds of concurrency control for shared predicates: *locking* and *timestamp ordering*. Due to the nature of fragmentation, only user predicates are shared, hence we only provide concurrency control for user predicates.

10.4.1 Locking

Predicates	Lock owners	Lock status	Lock queue
p_1	r_1, r_3	R	$\{r_5, r_{11}, ...\}$
p_2	r_4	W	$\{r_5\}$

Table 10.1: An Example Lock Table

In a lock-based approach, each shared user predicate is attached with a lock and called a *lock unit*. A shared predicate can be specially marked by a tag as one of its arguments. Every OCP server maintains a lock table like the one in table 10.1. Each row of the lock table is dedicated to one shared predicate. It also records which reactors currently hold on to the lock, what type of lock and which reactors are waiting to acquire the locks. In general, sustain condition c involves reading lock units and action a involves writing to lock units. Here we apply a *two-phase locking (2-PL)* mechanism commonly used in distributed database.

In the *growing phase* of the algorithm, the local OCP server analyzes the reactor (including goals and subgoals) for all lock units. Then it blocks the reactor and tries to acquire all the necessary locks contact the owners of these locks. The reactor will remain blocked until all locks have been obtained. When the reactor is finished, it returns all the locks to their respective owners in different OCP servers. The lock tables on these OCP servers will be updated accordingly. This is known as the *shrinking phase*. Suppose there are two accesses of shared predicate p at almost the same time. Depending on whether the accesses are read or write and their order of arrival, there are 4 combinations of outcome, namely, read-read, read-write, write-read and write-write.

read-read Both accesses are granted. The lock table records the two agents.

read-write Read is granted and write is blocked, write is queued.

write-read Write is granted and read is blocked, read is queued.

write-write First write is granted and the second is blocked, second write is queued.

To prevent the deadlock situation where one reactor acquired all the locks it needs but is somehow dead and not able to return the locks, OCP servers record the time of giving the locks to a particular reactor and attaches a deadline to each lock. If the agent fails to return the locks by due time, the reactor is excluded from the lock table and the locks are confiscated.

10.4.2 Timestamp ordering

The optimistic concurrency control comes from the belief that it is very rare for two remote reactors to mess with the one shared predicate and cause inconsistency. Therefore all reactors should be allowed to proceed anyway they like. To implement this approach, we need to augment our distributed OCP architecture a little bit. First, we need a global synchronization master which is connected to all OCP servers. The purpose of the global master is to serialize the operations at local OCP servers. The global master contains a master copy of the entire CLP($\mathcal{R}$) program. Second, we attach a clock to each local OCP server and the global master. The clocks do not have to be in sync but the master knows how much each local clock is skewed by. Each time a reactor is run locally, it is recorded in a local reactor sequence table along with its entry time. All computations, including delayed reactors must be book-kept and stored in a stack. Notice all reactors have to be run sequentially on the local server.

Once in a while, the master sends a broadcasting message to initiate a global synchronization. All OCP servers will then complete their pending reactors and send their sequence tables to the master. The master, once received all tables, will compare the table entries and sequence the operations on the master program copy. If there's no conflicts among the operations, the master program will be updated and a restart message sent to all servers. The local servers will clear off their sequence table and continue with their queued reactors.

However, if one shared predicate p is critically access by both reactors r_1 and r_2, then we have a *complication*. The master will compare the timestamp of these two reactors and allow the operation to proceed by the earlier reactor, say r_1, and reject r_2. It then discard the sequence table sent by r_2's original OCP server and notify the server of the reactor that fails to serialize. Normal synchronization will continue with the rest of the sequence tables. Now the r_2 server has do *roll-back*. It will reverse and delay all the reactors it has done until r_2, and wait for the restart message from the master. This is really nasty if many operations have been done, a trade-off for the optimistic approach.

The scheme we describe here is a basic one and can be improved. For examples, instead of rolling back all the reactors after r_2, we could unfold only those reactors that rely on the result from r_2. But this requires the master to reload the sequence table from this site and skip over reactors. This method involve more communication and better synchronization algorithm.

10.5 Triggering

The triggering mechanism in distributed OCP is complicated by the fact that a local sustain condition may rely on some remote facts, thus triggering must be done remotely as well. Recall that in the centralized OCP, when a reactor is blocked, the sustain condition is compiled into indexables and are inserted to index trees of all the base predicates and/or views that the condition depends on. Here we extend the idea and distribute the index trees to multiple OCP servers on the network. Because the user predicates are recorded disjointly on the system, there is only one possible index tree for each base user predicate. When a condition fails during the test, the condition goal is again compiled into indexables. For local predicates, that can be readily done and inserted to the local index tree as usual. If a predicate/view is remote, then the indexable is sent to the remote OCP server where this predicate or view is located, and, along with the pointer to this reactor, inserted into the index tree there.

For every index tree, a local update may produce a list of woken reactors. Where the reactor is a remote one (pointer to a remote OCP server), the trigger message is sent over and the remote server will wake up that reactor from the delay list and queue it for execution.

10.6 Conclusion and Future Work

This chapter presents some working ideas on the implementation of a distributed OCP system. The distributed OCP system allows parallelism and enables large applications with massive reactors. We have highlighted the implementation issues such as fragmentation of predicates, system architecture, concurrency control and triggering.

There are a few possible future extension to this research. First, as the current CLP($\mathcal{R}$) system is sequential, concurrency is achieved at reactor level, which means the distributed OCP offers only very coarse-grain parallelism. It would be much better if we can do fine-grained parallelism where instructions within a CLP($\mathcal{R}$) goal can be interleaved with other goals. In order to do this, we need to develop a parallel CLP($\mathcal{R}$) system. Second, if we employ "look-ahead" in the lock-based concurrency control, then not all remote access are "critical", e.g. if we read a variable in a remote predicate only once and never use the value elsewhere in a reactor, then this read is not considered a critical access and no lock is required. To realize the look-ahead, program analysis of the reactor and the CLP($\mathcal{R}$) goals involved is required. Third, more study can be done on the policy of distributing CLP($\mathcal{R}$) fragments and program agents. Usually, the data distribution is inherent to the application, but it can also rely on the usage pattern of the agents. Is the usage pattern changes, the data can be redistributed. On the other hand, if an agent connected to a server makes too many remote accesses to another server, it can be "migrated" to that server.

Part V

Finale

Chapter 11

Concluding Remarks

This thesis has presented an overview of a new reactive and open programming framework for distributed agents. The key contributions of this thesis are:

- the use of a *highly structured* updateable knowledge base as a communication and synchronization medium and environment for the program agents interact with;
- the reactivity for distributed programs in a *timely fashion*;
- the advanced speculative model which is a powerful way to allow the agents to *speculate* against the future so that the chances of achieve a solution for a given problem is maximized;
- a trigger framework on complex logical conditions with the notion of views and abstraction;
- the novel main memory spatial index structure: RC-tree.

As a support for the above main contributions, we have implemented a rudimentary OCP system based on the basic model for the Python language and tested it with a number of simple applications.

A number of open problems remain unsolved in this thesis and could become future research.

First, we have yet to suggest a systematic way to abstract composite views which may eventually allow us to automate the abstraction process. We also need to develop a tool for verifying the correctness of user defined abstractions. This could be done by program analysis.

Second, most of the experiments and simulations done for the RC-tree and triggering were in low dimensions. It is interesting to further investigate the behavior and practicality of the RC-tree in higher dimensions. Certain modifications to the RC-tree may be necessary to make it work better under those circumstances.

Third, we have only offered a few optimization ideas in speculative OCP. Simulation of these concepts under a number of reasonable scenarios will be helpful. Furthermore, it is more convincing if some analysis of these techniques can be done.

And last but not least, the implementation of a speculative OCP system with support of GCC will be extremely interesting. A speculative OCP system will basically consists of a (dynamic) number of reactive OCP systems, as the number of worlds increases and decreases.

References

[ACD+96] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[AdBHY04] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The priority R-Tree: A practically efficient and worst-case optimal r-tree. In *Proceedings of the 2004 ACM SIGMOD Conference*, pages 347–358. ACM, 2004.

[Air] Air War College. War games, simulations and exercises. http://www.au.af.mil/au/awc/awcgate/awc-sims.htm.

[And99] Arne Andersson. General balanced trees. *Journal of Algorithms*, 30(1):1–18, 1999.

[AR00] Iliès Alouini and Peter Van Roy. Fault-tolerant mobile agents in Mozart. In *2nd International Symposium on Agent Systems and Applications (ASA2000) and 4th International Symposium on Mobile Agents (MA2000)*, Zurich, Switzerland, September 2000. Poster presented at the conference.

[Bal91] Hentri E. Bal. *Programming Distributed Systems.* Prentice Hall International, 1991.

[BDFC00] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science, FOCS 2000*, pages 399–409. IEEE Computer Society, 2000.

[Ben75] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[BG89] Reem Bahgat and Steve Gregory. Pandora: Non-deterministic parallel logic programming. In *Proceedings of the ICLP*, pages 471–486, 1989.

[BG92] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BJL+95] Robert D. Blumofe, Christopher F. Joerg, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95, Santa Barbara, California, July 19-21, 1995*, pages 207–216. ACM, 1995.

[BK93a] Henri E. Bal and M. Frans Kaashoek. Object distribution in ORCA using compile-time and run-time techniques. In *OOPSLA*, pages 162–177, 1993.

[BK93b] Anthony J. Bonner and Michael Kifer. Transaction logic programming. In *Proceedings of the Tenth International Conference on Logic Programming (ICLP)*, pages 257–279. MIT Press, 1993.

[BK96] Anthony J. Bonner and Michael Kifer. Concurrency and communication in transaction logic. In *Joint International Conference and Symposium on Logic Programming*, pages 142–156, 1996.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 322–331. ACM, 1990.

[BKT92] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. ORCA: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual IEEE Conference on Foundations of Computer Science (FOCS'94)*, 1994.

[BPH+90] D. Beard, M. Palaniappan, A. Humm, D. Banks, A. Nair, and Y.-P. Shan. A visual calendar for scheduling group meetings. In *Proceedings of the CSCW '90 Conference on Computer-Supported Cooperative Work*, pages 279–290. ACM Press, 1990.

[BR91] D. J. Bertsimas and G. V. Ryzin. Stochastic and dynamic vehicle routing problem in the euclidean plane. *Operations Research*, 39:601–614, 1991.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3):261–322, 1989.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *SOSP*, pages 152–164, 1991.

[CG86a] Nicholas Carriero and David Gelernter. The s/net's linda kernel. *ACM Transactions on Computer Systems (TOCS)*, 4(2):110–129, 1986.

[CG86b] Keith L. Clark and Steve Gregory. Parlog: Parallel programming in logic. *ACM Trans. Program. Lang. Syst.*, 8(1):1–49, 1986.

[CL99] Yves Caseau and François Laburthe. Heuristics for large constrained vehicle routing problems. *Journal of Heuristics*, 5(3):281–303, 1999.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 178–188. ACM Press, 1987.

[dBGHO03] Mark de Berg, Joachim Gudmundsson, Mikael Hammar, and Mark H. Overmars. On R-trees with low query complexity. *Computational Geometry*, 24(3):179–195, 2003.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In H. Garcia-Molina and H. V. Jagadish, editors, *SIGMOD Conference 1990*, pages 202–214. ACM Press, 1990.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Inc. Englewood Cliffs, N.J., 1976.

[DL99] Don Dailey and Charles E. Leiserson. Using cilk to write multiprocessor chess programs. In *Proceedings of the 9th International Conference on Advances In Computer Chess*, 1999.

[FFMM94] Timothy W. Finin, Richard Fritzson, Don McKay, and Robin McEntire. Kqml as an agent communication language. In *CIKM*, pages 456–463, 1994.

[FKN80] H. Fuchs, Z.M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. In *Proceedings of SIGGRAPH '80, Computer Graphics*, pages 124–133, 1980.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40st Annual Symposium on Foundations of Computer Science, FOCS 1999*, pages 285–298. IEEE Computer Society, 1999.

[Fly] Flying distance between 325 cities in the world. http://www.etn.nl/distanc4.htm.

[FM87] Raphael Finkel and Udi Manber. DIB - a distributed implementation of backtracking. *ACM Transactions of Programming Languages and Systems*, 9(2):235–256, April 1987.

[FRS93] Françoise Fabret, Mireille Régnier, and Eric Simon. An adaptive algorithm for incremental evaluation of production rules in databases. In *VLDB*, pages 455–466, 1993.

[Fuj00] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems.* John Wiley & Sons Inc., 2000.

[Gae98] Volker Gaede. Multidimensional access methods. *ACM Computing Survey*, 30(2):170–231, 1998.

[GB82] David Gelernter and Arthur J. Bernstein. Distributed communication via global buffer. In *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 10–18, 1982.

[GC05] Jim Gray and Mark Compton. A call to arms. *ACM Queue*, 3(3), April 2005.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[GG87] Thierry Gautier and Paul Le Guernic. SIGNAL: A declarative language for synchronous programming of real-time systems. In *FPCA*, pages 257–277. Springer, 1987.

[GG98] Volker Gaede and Oliver Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[GGPT99] M. Gendreau, F. Guertin, J.-Y. Potvin, and E. Taillard. Parallel tabu search for real-time vehicle routing and dispatching. *Transportation Science*, 33(4):381–390, 1999.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the ACM SIGMOD 1987 Annual Conference*, pages 249–259. ACM Press, 1987.

[GPA+01] Gopal Gupta, Enrico Pontelli, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of prolog programs: a survey. *Transaction of Programming Languages and Systems (TOPLAS)*, 23(4):472–602, 2001.

[Gut84] Antonin Guttman. R-trees - a dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 47–57. ACM, 1984.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Har93] John V. Harrison. Active rules in deductive databases. In *Proceedings of Conference of Information Knowledge Management, CIKM*, 1993.

[Has00] Wilhelm Hasselbring. Programming languages and systems for prototyping concurrent applications. *ACM Comput. Surv.*, 32(1):43–79, 2000.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[HJ90] Seif Haridi and Sverker Janson. Kernel andorra prolog and its computation model. In *Proceedings of the ICLP*, pages 31–46, 1990.

[HJM+92] N.C. Heintze, J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) programmer's manual version 1.2, 1992.

[HLN+90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transaction on Software Engineering*, 16(4):403–414, 1990.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[HMJH05] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of PPoPP*, 2005.

[HMSY89] Nevin Heintze, Spiro Michaylov, Peter Stuckey, and Roland Yap. On meta-programming in CLP($\mathcal{R}$). In *Logic Programming: Proceedings of the North American Conference, 1989*, pages 52–68. MIT Press, 1989.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st International Conference of Very Large Data Bases, VLDB*, pages 562–573, 1995.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[HPY+93] S. Hariri, J. B. Park, F.-K. Yu, M. Parashar, and G. C. Fox. A message passing interface for parallel and distributed computing. In *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*, pages 84–91, 1993.

[HRS97] Seif Haridi, Peter Van Roy, and Gert Smolka. An overview of the design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*, pages 176–187, Maui, Hawaii, USA, July 1997. ACM Press.

[HS96] Pascal Van Hentenryck and Vijay A. Saraswat. Strategic directions in constraint programming. *ACM Comput. Surv.*, 28(4):701–726, 1996.

[Hul87] M. Elizabeth C. Hull. Occam - a programming language for multiprocessor systems. *Comput. Lang.*, 12(1):27–37, 1987.

[ILO] ILOG. http://www.ilog.com.

[ILO01] ILOG. ILOG solver 5.1: Reference manual, 2001.

[Jag90] H.V. Jagadish. Spatial search with polyhedra. In *Proceedings of 6th IEEE International Conference on Data Engineering, ICDE*, pages 311–319, 1990.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of POPL*, pages 111–119, 1987.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(R) language and system. *ACM Transaction on Programming Languages and Systems*, 14(3):339–395, 1992.

[JMY91] Joxan Jaffar, Spiro Michaylov, and Roland Yap. A methodology for managing hard constraints in CLP systems. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 306–316. ACM, 1991.

[JSYZ04] Joxan Jaffar, Andrew E. Santosa, Roland H.C. Yap, and Kenny Q. Zhu. Scalable distributed depth-first search with greedy work stealing. In *Proceedings of 16th International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 98–103. IEEE Computer Society, 2004.

[JYZ05] Joxan Jaffar, Roland H.C. Yap, and Kenny Q. Zhu. Coordination of many agents. In *Proceedings of the 21st International Conference on Logic Programming, ICLP*, 2005.

[JYZ06] Joxan Jaffar, Roland H. C. Yap, and Kenny Q. Zhu. Indexing for dynamic abstract regions. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE*, pages 98–112, 2006.

[KCDZ94] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operation systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, 1994.

[KCK01] Kihong Kim, Sang K. Cha, and Keunjoo Kwon. Optimizing multidimensional index trees for main memory access. In *Proceedings of the 2001 ACM SIGMOD Conference*. ACM, 2001.

[KS97] Norio Katayama and Shin'ichi Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 369–380. ACM, 1997.

[KSN00] Mamadou Tadiou Kone, Akira Shimazu, and Tatsuo Nakajima. The state of the art in agent communication languages. *Knowledge and Information Systems*, 2(3):259–284, 2000.

[KZ88] Richard Karp and Yanjun Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the ACM Annual Symposium on Theory of Computing*, pages 290–300. ACM, 1988.

[Leo01] Claudia Leopold. *Parallel and Distributed Computing. A survey of models, paradigms and approaches.* John Wiley & Sons, 2001.

[LLM98] Georg Lausen, Bertram Ludäscher, and Wolfgang May. On active deductive databases: The statelog approach. In B. Freitag, M Kifer H. Decker, and A. Voronkov, editors, *Transactions and Change in Logic Databases.* Springer, 1998.

[Llo87] John Wylie Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[MHN84] Takashi Matsuyama, Le Viet Hao, and Makoto Nagao. A file organization for geographic information systems based on spatial proximity. *International Journal of Computater Vision, Graphic and Image Processing*, 26(3):303–318, 1984.

[Mic98] Sun Microsystems. Javaspace specification, 1998.

[MNPT] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. R-trees have grown everywhere.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, September 1992.

[NAH98] Vasudevarao Nugala, Stephen J. Allan, and James W. Haefner. Parallel implementation of individual-based models in biology: Bulletin- and non-bulletin-board approaches. *BioSystems*, 45(2):87–97, 1998.

[OMSD87] B. Ooi, K.J. Mcdonell, and R. Sacks-Davis. Spatial kd-tree: An indexing mechanism for spatial databases. In *Proceedings of the IEEE Computer Software and Applications Conference*, pages 433–438, 1987.

[Ore89] Jack A. Orenstein. Redundancy in spatial databases. In *Proceedings of the ACM SIGMOD Conference*, pages 294–305, 1989.

[Ore90] Jack A. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. In *Proceedings of the ACM SIGMOD Conference*, pages 343–352, 1990.

[OV99] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 2 edition, 1999.

[Ove83] Mark H. Overmars. *The design of dynamic data structure.* Springer-Verlag, 1983.

[PJO95] W. B. Powell, P. Jaillet, and A. Odoni. Stochastic and dynamic network and routing. *Handboosk in Operations Research and Management Science*, 8:141–295, 1995.

[Psa95] H. N. Psaraftis. Dynamic vehicle routing: Status and prospects. *Annals of Operations Research*, 61:143–164, 1995.

[R-T] R-Tree Portal. http://www.rtreeportal.org.

[RBD+03] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz, and Christian Schulte. Logic programming in the context of multiparadigm programming: the oz experience. *TPLP*, 3(6):715–763, 2003.

[Rev02] Peter Revesz. *Introduction to Constraint Databases.* Springer-Verlag New York, Inc., 2002.

[Rey87] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, July 1987.

[RG02] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* McGraw-Hill, 3rd edition, 2002.

[RHB+97] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.

[RSS01] Kenneth A. Ross, Inga Sitzmann, and Peter J. Stuckey. Cost-based unbalanced R-trees. In *Proceedings of the 13th International Conference on Scientific and Statistical Database Management*, pages 203–212, 2001.

[RT95] E. Rochat and E. Taillard. Probabilistic diversification and intensification in local search for vehicle routing. *Journal of Heuristics*, 1:147–167, 1995.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[SBD02] Lamjed Ben Said, Thierry Bouron, and Alexis Drogoul. Agent-based interaction analysis of consumer behavior. In *Proceedings of AAMAS*, pages 184–190. ACM, 2002.

[Sch00] Christian Schulte. Parallel search made simple. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, 2000.

[SD98] S. Sen and E. H. Durfee. A formal study of distributed meeting scheduling. *Group Decision and Negotiation*, 7:265–289, 1998.

[Sha89] Ehud Y. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[Smo95] Gert Smolka. The Oz programming model. In *Computer Science Today*, pages 324–343. Springer, 1995.

[Sol87] Marius M. Solomon. Algorithms for vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2), 1987.

[Sos94] Rok Sosic. A parallel search algorithm for the n-queens problem. In *Proceedings of Parallel Computing and Transcomputer Conference, Wollonggong*, pages 162–172. AOI Press, 1994.

[SRF87] Timos Sellis, Nick Roussopoulos, and Christos Faloutosos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th VLDB Conference*, pages 507–518. VLDB, 1987.

[ST83] Ehud Y. Shapiro and Akikazu Takeuchi. Object oriented programming in concurrent prolog. *New Generation Comput.*, 1(1):25–48, 1983.

[ST98] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, 1998.

[Stu97] Peter J. Stuckey. Constraint search tree. In *Logic Programming, Proceedings of the Fourteens International Conferences on Logic Programming, ICLP*, pages 301–315. MIT Press, 1997.

[Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4):315–339 (or 315–340??), December 1990.

[SZD98] K.P. Sycara, D. Zeng, and K. Decker. Intelligent agents in portfolio management. In Nicholas R. Jennings and Michael J. Wooldridge, editors, *Agent Technology: Foundations, Applications and Markets*, chapter 14, pages 267–282. UNICOM/Springer, 1998.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.

[Tes02] Leigh Tesfatsion. Agent-based computational economics: Growing economics from the bottom up. *Artificial Life*, 8(1):55–82, 2002.

[TLZO01] Kay-Chen Tan, Loo-Hay Lee, Kenny Q. Zhu, and Ke Ou. Heuristic methods for vehicle routing problem with time windows. *Artificial Intelligent in Engineering*, 3(281–295), 2001.

[Tra] Trade Station. http://www.tradestation.com.

[Ued86] Kazunori Ueda. Guarded horn clauses. In *Logic Programming '85, Proceedings of the 4th Conference*, LNCS 221, pages 168–179, 1986.

[VB98] I. Vlahavas and N. Bassiliades. *Parallel, Object-Oriented, and Active Knowledge-Base Systems.* Kluwer International Series on Advances in Database Systems. Kluwer Academic Publisher, 1998.

[vRFLD03] Guido van Rossum and Jr. Fred L. Drake. *An Introduction to Python.* Network Theory Ltd, 2003.

[WC96] Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann Publishers, Inc., 1996.

[Wei99] Gerhard Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* The MIT Press, 1999.

[ZO00] Kenny Q. Zhu and Kar-Loon Ong. A reactive method for real time dynamic vehicle routing problems. In *10th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2000.

[ZS02] Kenny Q. Zhu and Andrew E. Santosa. A meeting scheduling system based on open constraint programming. In *14th International Conference on Advanced Informational System Engineering, CAiSE'02.* Springer, 2002.

Vita

Kenny Qili Zhu was born in 1976, in Nanjing, China, where he received his elementary and secondary education. He received Bachelor of Engineering (Computer Engineering) with honors from Department of Electrical Engineering, National University of Singapore in 1999. His research interests are concurrent/distributed programming languages and systems, multi-agent systems, heuristics for combinatorial optimization and evolutionary computing. In his spare time, Kenny likes to read, go swimming, do photography, listen to music and play the trumpet.