Chapter 1

Introduction

In this chapter, Section 1.1 reviews related research work in the field of mobile computing, mobile agent, intelligent agent and multi-agent systems, and addresses existing issues and open problems. Section 1.2 discusses our approaches for addressed issues and compares these approaches with previous work in the field. Section 1.3 provides related background knowledge. Section 1.4 discusses the structure of thesis.

1.1 Literature Review

Mobile computing, involving the movement of physical devices such as laptop and palmtop computers, has matured rapidly as a field of computer science. The fast growing technology of cellular communication, wireless LANs, and satellite services will ultimately make information accessible anywhere and at any time. The advent of widespread portable computers has led to a wide variety of interesting hardware and software issues. Regardless of size, most mobile computers are equipped with a wireless connection to the fixed part of the network, and perhaps to other mobile computers. The resulting computing environment is often referred to as mobile or nomadic computing.

Owing to the specific features of mobile environments, there are many research issues that need to be discussed in the mobile computing field: data access; location and mobility management including mobility of users and resources; energy efficient data management; disconnection management; transaction processing; correctness theory and verification for communication protocols; data replication and migration.

Transaction processing in mobile environments is a very challenging research area. The first key issue is weak connectivity: that is, how a computer can operate when
disconnected from the network, intermittently connected, or connected over very slow communication links [Milojičić, 1999]. Kistler and Satyanarayanan were among the first to recognize the problem of disconnection and provided a solution in the Coda system [Kistler, 1992; 1993; Satyanarayanan, 1990; 1993]. The central idea behind their work is that caching of data, traditionally widely used for enhancing performance, can also be exploited to improve availability in mobile environments. Mummert, and others, extended the Coda system to support weak connectivity [Mummert, 1994; 1995; 1996]. They modified their transport protocol to be more efficient over slow networks. They added volume-level cache consistency so that rather than revalidating every file upon reconnection, a client could first determine if anything at all on a file system had been modified since disconnection. Demers, and others, focused on weak connectivity in their Bayou system [Demers, 1994; Terry, 1995]. The Bayou storage system provides an infrastructure for collaborative applications that manages the conflicts introduced by concurrent activity while relying only on the weak connectivity available for mobile computing. In this system, clients can read and write any replica, and hosts can interact in a pair-wise fashion to exchange updates, so that updates are eventually propagated to all hosts. Also, application-specific mechanisms handle any conflicts that arise. Barron and Housel’s WebExpress system [Chang, 1997; Housel, 1996] performs several optimizations specific to web access in a wireless environment.

The second challenging aspect of transaction processing in a mobile environment deals with wireless connectivity when a computer moves between “cells” in a wireless network [Milojičić, 1999]. The Mobile IP protocol, which was initially proposed by Ioannidis, Duchamp and Maguire [Ioannidis, 1993], has been improved, expanded upon, and evaluated in numerous ways. Support for mobility within the Internet Protocol has been made possible through a long series of research initiatives, beginning with work by Ioannidis, and others, in the early 1990s [Ioannidis, 1991]. In one of their early Mobile IP papers, Ioannidis, Duchamp and Maguire [Ioannidis, 1993], presented the design, implementation, and evaluation of Mobile IP, a set of IP-based protocols and mechanisms to support host mobility throughout the Internet. Charles Perkins provided substantial detail on the latest modifications to the protocol, as well as additional research issues [Perkins, 1998]. Balakrishnan et al. examined the
interactions of TCP and wireless networks [Balakrishnan, 1995, 1997a, b]. They found that link-level recovery is useful and splitting the TCP connection gives good performance. Jain and Krishnakumar discussed the problem of supporting mobile transactions which access distinguishable instances, for which site escrow methods are typically not appropriate [Jain, 1994].

The attributes of the typical stationary environment have guided the development of classical distributed computing techniques for building client-server applications. These applications are usually unaware of the actual state of the environment, therefore they make certain implicit assumptions about the location and availability of resources. Such mobile-transparent applications can be used unmodified in mobile environments by having the system shield or hide from applications the differences between the stationary and mobile environments. The Ficus file system is a mobile-transparent, user-level file system supporting disconnected operation and peer-to-peer data sharing [Reiher, 1994]. The Little Work project caches files to smooth disconnection from the AFS file system [Huston, 1995]. Conflicts are detected and reported to the user. The BNU project implements an RPC-driven mobile-transparent application framework on mobile computers. It enables code shipping by downloading scheme functions for interpretation [Watson, 1993].

Although the mobile-transparent approach is appealing in that it offers to run existing applications without alteration, it is fundamentally limited in that it hides the information needed to allow applications to remain correct and to perform well in an intermittently-connected environment. This mobile-transparent approach simplifies mobile applications, but sacrifices functionality and performance.

The alternative to hiding environmental information from applications is to expose information and involve applications and users in decision-making. This alternative yields the class of mobile-aware applications. The mobile-aware argument can be viewed as applying the end-to-end argument to mobile applications [Joseph, 1997] that is, communication functionality can be implemented only with the knowledge and help of the application standing at the endpoints of the communications system [Saltzer, 1984]. The need for mobile-aware applications and complementary system
services to expose mobility to applications was identified concurrently by several groups. Katz noted the need for the adaptation of mobile systems to a variety of networking environments [Katz, 1994]. Davies et al. cited the need for protocols to provide feedback about the network to applications in a vertically integrated application environment [Davies, 1994]. Kaashoek et al. created a web browser that exposed the mobile environment to code implementing mobile-aware web pages [Kaashoek, 1994]. Baker identified the dichotomy between mobile-awareness and mobile-transparency in general applications and system design [Baker, 1994]. The InfoPad [Le, 1995], Daedalus [Narayanaswamy, 1996], GloMop [Fox, 1996], and W4 [Bartlett, 1994] projects focused on mobile-aware wireless information access. The Rover toolkit [Joseph, 1995; 1997] of Joseph, and others, attempted to support both mobile-aware and mobile-transparent applications. It provided two abstractions: the relocatable dynamic object and queued remote procedure call.

**Open Problem One: Notions and Concepts of Data Transaction in Mobile Environments.**

All of these works above are concentrated on mobile computing system modeling and implementation, however a lot of fundamental classical concepts and protocols of transaction processing may not be accurate and true in mobile environments, and need to be rechecked and redefined. These include the concepts of transaction, transaction history, equivalent history, history serial, serialization graph, notion of serializability, locking protocols, commit and abort protocols, rollback protocols, and log protocols.

**Open Problem Two: Serializability Criteria and Concurrency Control Algorithms in Mobile Environments.**

Apart from a range of basic notions and protocols, a number of theory algorithms and techniques have to be reconsidered, modified or completely recreated, to apply to mobile environments. The algorithms and techniques used to ensure serializability (concurrency control algorithms) become much more complex in mobile environments, wherein, when users modify local copies of global data, consistency
Chapter 1. Introduction

becomes an issue. Optimistic concurrency control [Kung, 1981] is argued to be more useful in such mobile environments because pessimistic methods may be inappropriate (a disconnected user cannot acquire or release locks), as pointed out by the designers of Coda [Kistler, 1992].

However, using an optimistic approach does involve some difficulties [Joseph, 1997]. In particular, long duration partitions will cause a greater incidence of apparent write-write conflicts than in stationary environments. Joseph suggested that it is therefore important to use application-specific semantic information to detect when such conflicts are false positives and can be resolved [Joseph, 1997]. He investigated how to build a mobile-transparent file system proxy for mobile computers by using optimistic concurrency control and pre-fetching [Kistler, 1992]. The Rover toolkit offered applications a distributed object system based on a client/server architecture with client caching and optimistic concurrency control [Joseph, 1995]. Gray et al. performed a thorough theoretical analysis of the options for database replication in a mobile environment and concluded that primary-copy replication with tentative updates is the most appropriate approach for mobile environments [Gray, 1996]. Krishnakumar’s paper discussed a method to perform mobile sales transactions using site-transaction escrow methods [Krishnakumar, 1996], but data type used for the discussion is very limited.

Due to the characteristics of real-time broadcast environments, the realization of “instant” information access over a mobile network relies on real-time processing of transactions, and it makes the timeliness of data accesses an important issue. As a result, research on processing soft real-time transactions in mobile distributed real-time database systems is receiving growing attention [Datta, 1996; Kayan, 1999; Leong, 1997; Ulusoy, 1998; Xuan, 1997]. In his paper [Lam, 2000], Lam et al. proposed a distributed real-time locking protocol based on the High Priority Two Phase Locking scheme [Abbott, 1992]. Dang and Liu presented a correct criterion of serializability called weak serializability for mobile real-time transaction [Dang, 2003]. Bhalla [Bhalla, 2003] studied an embedded concurrency control technique to provide support on transaction updates by mobile clients.
Serializability is too strong as a correctness criterion and not suitable for mobile transactions [Daniel, 1999; Lam, 1999; Lee, 1999, Ulusoy, 1998]. The relaxed serializability and concurrency control methods are more desirable and required in mobile environments, to improve performance or availability, than in stationary environments. The investigation on relaxed serializability and relaxed concurrency control method applicable to mobile environments therefore remains a very important research topic. A relaxed concurrency control method that can support mobile transaction with relaxed serializability, application independent, and with good performance, is especially highly desired.

**Open Problem Three: Knowledge Representation and Knowledge Transaction in Mobile Environments.**

In the mobile computing field, studying knowledge transaction in mobile environments is a novel research topic. Currently there is a separation between the intelligent agent community and the mobile system community. Within the intelligent agent community, a lot of frameworks/models have been developed for problem solving, knowledge representation and reasoning, for example the stable model/answer set [Gelfond, 1991], SMODEL, DLV and XSB model [Eiter, 1997; Nemela, 1996; Rao, 1997]. Logic programming has been proved to be one of the most promising logic based formulations for problem solving, knowledge representation and reasoning. It has great advantages on both declarative semantics and efficient proving procedures (e.g. SMODEL, DLV and XSB), which make this method more applicable in real world problem domains. These models are knowledge oriented with declarative semantics, and their specification language can specify the details of knowledge transaction. But discussion of these models is limited to conventional environments, and has not been extended to mobile environments.

Within the mobile system community, on the other hand, a lot of efforts have been invested in developing mobile agent systems. These efforts have effectively led the development of mobile agent systems, with General Magic at the forefront with Telescript [White, 1996]. Telescript is similar to Java even though it is targeted for a closed network. General Magic’s effort was followed by other industrial
developments, such as IBM’s Aglets [Lange, 1998a], Concordia [Walsh, 1998] and Voyager [Glass, 1998], and university research, such as Agent TCL [Kotz, 1997] and Tacoma [Johansen, 1995]. Aglets is one of the best known and most widespread mobile agent systems today [Lange, 1998a]. It is one of the first agent systems to jump on the Java bandwagon. Mole is one of the first academic agent systems written in Java [Baumann, 1997; 1998]. The work in [Bettini, 2002; Deugo 2001; Lange, 1998b] has presented KLAVA, a Java package for implementing distributed applications that can exploit mobile code and run over a heterogeneous network environment. All the above approaches have the following disadvantages: (1) They are not suitable for knowledge-oriented processing; (2) They have no declarative semantics. They are low-level algorithms for “how to do” and have no intelligent high-level “what to do” functionality; (3) The abstract transaction semantics is hard to be specified in those systems.

In mobile environments, current research for transaction processing concentrates on data rather than knowledge transaction [Ahmad, 1995; Barbara, 1994; Imielinski, 1996; Mirghafori, 1995; Bhalla, 2003; Dang, 2003]. So far no knowledge transaction has been formally studied and no model has been formalized to process knowledge transaction in mobile environments. In addition, no framework has been formalized for multi-agent systems in mobile environments. These remain as open questions for mobile computing. There is no doubt that it is both necessary and meaningful to formalize a knowledge transaction model and a multi-agent system framework in mobile environments, because this will provide a foundation to further the study of knowledge base and intelligent agents in mobile systems.

1.2 Thesis Contributions

Compared with previous works, this thesis contributes to the field of mobile computing in the following aspects:

1. A range of classical notions and protocols of transaction processing and concurrency control have been rechecked and redefined in mobile environments. In addition, a criterion for a serial history is given. As the major outcome, two
Chapter 1. Introduction

concurrency control theorems, wormhole theorem and locking theorem, are produced and proved in mobile environments. It is both important and necessary to recheck and redefine these concepts and protocols because they are fundamental to the study of transaction processing in mobile environments. To my best knowledge, no prior work has been done in this area.

2. A relaxed serializability correctness criterion – epsilon serializability is formally defined for mobile transaction. A relaxed concurrency control method for mobility support is proposed in this thesis, so that rapidly changing data objects can be processed in mobile environments with limited inconsistency. Compared with other concurrency control algorithms, the proposed approach has features that are applicable for mobile transaction, is high-level and application-independent, and has improved transaction performance by adopting epsilon serializability in mobile environments.

3. A rule-based knowledge transaction model is presented and formalized in mobile environments, which integrates the features of both mobile environments and intelligent agents. The resulting knowledge transaction language and model can be used for knowledge transaction representation, formalization and knowledge reasoning in mobile environments. This is an early effort to study and formalize a knowledge transaction model in mobile environments.

4. A framework/model for a mobile logic programming multi-agent system is formalized in mobile environments. This formalization can be used to study knowledge transaction in mobile multi-agent systems. Again, this is an early effort to formalize a multi-agent system in mobile environments.

1.3 Background Knowledge

1.3.1 Mobile Computing
As Imielinski stated [Imielinski, 1996], in the near future, tens of millions of people will carry a portable palmtop or laptop computer. Smaller units, often called personal digital assistants or personal communicators, will run on batteries and may have only a small memory; larger ones will be powerful laptop computers with large memories and powerful processors. Mobile computing environments no longer require users to maintain a fixed and universally known position in the network, and enable almost unrestricted mobility. Mobility and portability have created an entire new class of applications and new massive markets combining personal computing and consumer electronics. The falling cost of both communication and mobile devices (laptop computers, palmtop computers, etc.) is making wireless computing affordable not only to business users but also to consumers.

Information will be easily accessible from virtually any place and time, and will be stored in a highly decentralized, distributed information infrastructure often termed the “information superhighway”. A wide variety of information servers will be accessible to mobile computers. We are already seeing the beginnings of this with the rapidly growing popularity of the World-Wide Web across a broad range of computer users. As the mobile infrastructure develops, it will become what is referred to as the “first wireless mile” or “wireless on-ramp” for the information superhighway. In some applications, mobile computers themselves may contain data, or data may be stored on flash-memory “smart cards”.

The general abstract view of a mobile system consists of Mobile Hosts (MHs) interacting with the fixed network via Mobile Support Stations (MSSs) [Imielinski, 1996]. The connection between the MH and the MSS is via a wireless link. Each MSS is in charge of a cell. Cells can have sizes ranging from pico cells of approximately one hundred meters in diameter to macro cells and perhaps even global satellite cells. The capabilities of mobile hosts will vary from “dumb” terminals to complex “walkstations” which essentially have the capabilities of desktop computers. We expect that users of mobile computers will want to run a variety of applications, depending on the physical size and power of their machines. These range from standard desktop applications like word processors, spreadsheets, and electronic mail, to remote information access via database system applications or
Web browsers. Additionally, there may be location-dependent applications specific to the particular location of the user at a particular time.

As stated by Milojićić [Milojićić, 1999], mobile computing is about physical migration or the migration of the hardware. This is different with process migration and mobile agents that are about logical migration, or migrating bits (code and data). Several differences between the mobile environment and the stationary environment must be addressed. Issues that represent minor inconveniences in stationary distributed systems are significant problems for mobile computers. This distinction requires a rethinking of the classical distributed systems techniques normally used in stationary environments. Computers in a stationary environment are usually very reliable. Relative to their stationary counterparts, mobile computers are quite fragile: a mobile computer may run out of battery power, be damaged in a fall, be lost, or be stolen. A mobile computer often has fewer computational resources available compared with most stationary computers. A stationary environment can distribute an application’s components and rely upon the use of high-bandwidth, low-latency networks to provide good interactive application performance. Mobile computers operate primarily in limited bandwidth, high-latency, and intermittently-connected environments. In mobile environments, applications will face frequent, lengthy network partitions. Some of these partitions will be involuntary (e.g., due to a lack of network coverage), while others will be voluntary (e.g., due to a high dollar cost). Mobile applications should handle such partitions gracefully and as transparently as possible. In addition, users should be able, as far as possible, to continue working as if the network is still available.

1.3.2 The Basics of Transaction Processing

Transaction and transaction processing systems have been widely studied by worldwide researchers. Ullman described the transactions as follows [Ullman, 1982]: “A transaction is a collection of one or more operations on the database that must be executed atomically; that is, either all operations are performed or none are.”
Claybrook gave the following definition of a transaction [Claybrook, 1992]: “Informally, a transaction is the execution of a program (or program segment) that performs some function on one or more resources such as shared, online databases or files. A transaction may be considered as an abstraction that is managed by a transaction processing (TP) system. A transaction is an atomic unit of execution that can be modeled as

\[ T = a_1a_2...a_n \]

Where the \( a_i \) are actions (operations) such as read, write, delete, rewrite, open, close, start_trans, commit_trans, abort_trans, and so on.”

Gray explained the transaction and transaction processing as follows [Gray, 1993]: The term transaction processing system is generally used to mean a complete system. A Transaction Processing system includes application generators, operations tools, one or more database systems, utilities, and networking and operating system software. A transaction is a collection of operations on the physical and abstract application state. Transaction processing systems pioneered many concepts in distributed computing and fault-tolerant computing. They introduced distributed data for reliability, availability and performance; they developed fault-tolerant storage and fault-tolerant processes for availability; and they developed the client-server model and remote procedure call for distributed computation. Most important, they introduced the transaction ACID properties - atomicity, consistency, isolation, and durability – that have emerged as the unifying concepts for distributed computation. A transaction can be considered as a collection of actions with the following properties:

**Atomicity:** A transaction’s changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.

**Consistency:** A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state. This requires that the transaction be a correct program.

**Isolation:** Even though transactions execute concurrently, it appears to each transaction, \( T \), others are executed either before \( T \) or after \( T \), but not both.
**Durability:** Once a transaction completes successfully (commits), it changes to the state ‘survive failures’.

As an example, a banking debit transaction is atomic if it both dispenses money and updates your account. It is consistent if the money dispensed is the same as the debit to the account. It is isolated if a transaction program can be unaware of other programs reading and writing your account concurrently (such as, your spouse making a concurrent deposit). And it is durable if, once the transaction is complete, the account balance is sure to reflect the withdrawal.

The isolation is variously called consistency (the static property), concurrency control (the problem), serializability (the theory), or locking (the technique). The system state consists of objects related in certain ways. A transaction that is started with a consistent system state may make the state temporarily inconsistent, but it will terminate by producing a new consistent state. Transactions, then, are units of consistency. This is the consistency of the transaction ACID property. If some action of the transaction fails, the system can automatically undo the actions of the transaction to return to the original consistent state. Defining transaction boundaries within application programs is a major part of application design. If transactions are run one at a time in isolation, each transaction sees the consistent state left behind by its predecessor. However, if several transactions execute concurrently, the inputs and consequent behavior of some may be inconsistent, even though each transaction executed in isolation is correct. Concurrent transaction execution must be controlled so that correct programs do not malfunction. Many approaches providing automatic isolation have been tried. There is consensus on one feasible solution - locking. The simplest lock protocol associates a lock with each object. Locking is a serialization mechanism that assures that only one transaction accesses an object at any given time. By refining the granularity of the lock (how much data the lock covers), increasing amounts of concurrency are allowed. The main contribution of transaction systems to concurrency control has been to refine these ideas to include automatic locking and to combine the locking algorithms with the transaction undo/redo algorithms.
Chapter 1. Introduction

1.3.3 Multi-Agent System and Knowledge Representation

As stated by Weiss [Weiss, 1999], distributed artificial intelligence (DAI) has evolved and diversified rapidly since its inception in the mid to late 1970s. Today it is an established and promising research and application field which brings together, and draws upon, results, concepts and ideas from many disciplines, including artificial intelligence (AI), computer science, sociology, economics, organization and management science, and philosophy. Its broad scope and multi-disciplinary nature make it difficult to precisely characterize DAI in a few words. The following definition is intended to serve as a starting point for exploring this arena:

DAI is the study, construction, and application of multi-agent systems, that is, systems in which several interacting, intelligent agents pursue some set of goals or perform some set of tasks.

An agent is a computational entity, such as a software program or a robot, that can be viewed as perceiving and acting upon its environment and that is autonomous, in that its behavior at least partially depends on its own experience. As an intelligent entity, an agent operates flexibly and rationally in a variety of environmental circumstances given its perceptual and effectual equipment. Behavioral flexibility and rationality are achieved by an agent on the basis of key processes such as problem solving, planning, decision-making, and learning. As an interacting entity, an agent can be affected in its activities by other agents and perhaps by humans. A key pattern of interaction in multi-agent systems is goal- and task-oriented coordination, both in cooperative and in competitive situations. In the case of cooperation several agents try to combine their efforts to accomplish as a group what the individuals cannot, and in the case of competition several agents try to get what only some of them can have. Multi-agent systems have the capacity to play a key role in current and future computer science and its application. Modern computing platforms and information environments are distributed, large, open and heterogeneous. Computers are no longer stand-alone systems. The increasing complexity of computer and information systems goes together with an increasing complexity of their applications. To cope with such applications, computers have to act more as “individuals” or agents, rather than just
“parts.” Also, multi-agent systems have the capacity to play an important role in developing and analyzing models and theories of interactivity in human societies.

As stated by Wooldridge [Wooldridge, 2001], multi-agent systems have only gained widespread recognition since about the mid-1990s. However, since then international interest in the field has grown enormously. This rapid growth has been spurred at least in part by the belief that agents are an appropriate software paradigm through which to exploit the possibilities presented by massive open distributed systems – such as the Internet. Although they will certainly have a pivotal role to play in exploiting the potential of the Internet, there is a lot more to multi-agent systems than this. Multi-agent systems seem to be a natural metaphor for understanding and building a wide range of what we might crudely call artificial social systems. The ideas of multi-agent systems are not tied to a single application domain, but, like objects before them, seem to find currency in a host of different application domains.

As stated by Baral [Baral, 2003], representing knowledge, and reasoning with it, are important components of an intelligent system, and are two important facets of artificial intelligence. Another important expectation from intelligent systems is their ability to accept high level requests, as opposed to detailed step-by-step instructions, and their knowledge and reasoning ability are used to figure out the detailed steps that need to be taken. To have this ability, intelligent systems must have a declarative interface whose input language must be based on logic. Among other characteristics, an intelligent entity – whether an intelligent autonomous agent, or an intelligent assistant – must have the ability to go beyond just following direct instructions while in pursuit of a goal. This is necessary to be able to behave intelligently when the assumptions surrounding the direct instructions are not valid, or there are no direct instructions at all. At one end of the spectrum the request is a detailed algorithm that spells out how to satisfy the request, which no matter how detailed it is may not be sufficient in cases where the assumptions inherent in the algorithm are violated. At the other end of the spectrum the request spells out what needs to be done, and the entity has the knowledge – again in the what form rather than the how form – and the knowledge processing ability to figure out the exact steps (that will satisfy the request) and execute them. When it does not have the necessary knowledge it either
knows where to obtain that knowledge, or is able to gracefully get around its ignorance through its ability to reason in the presence of incomplete knowledge. The languages for spelling out how are often referred to as procedural, while the languages for spelling out what are referred to as declarative. Thus our initial thesis, that intelligent entities must be able to comprehend and process a description of what, leads to the necessity of inventing suitable declarative languages and developing support structures around those languages to facilitate their use. We consider the development of such language to be fundamental to knowledge based intelligence, perhaps similar to the role of the language of calculus in mathematics and physics.

Baral stated that [Baral, 1994] knowledge representation is one of the most important sub-areas of artificial intelligence. If we want to design an entity (a machine or a program) capable of behaving intelligently in some environment, then we need to supply the entity with sufficient knowledge about this environment. To do that, we need an unambiguous language capable of expressing the knowledge, together with some precise and well-understood way of manipulating sets of sentences of the language which will allow us to draw inferences, answer queries, and update both the knowledge base and the desired program behavior. Around 1960, McCarthy first proposed the use of logical formulas as a basis for a knowledge representation language for this type [McCarthy, 1959]. His idea has been further developed by many researchers with various backgrounds and interests. First, the classical logic of predicate calculus served as the main technical tool for the representation of knowledge. It has a well-defined semantics and a well-understood and powerful inference mechanism. It was soon realized that this tool is inadequate. The difficulty is rather deep and related to the so-called “monotonicity” of theories based on predicate calculus. A logic is called monotonic if the addition of new axioms to a theory based on it never leads to the loss of any theorems proved in this theory. Commonsense reasoning is non-monotonic, new information constantly forces us to withdraw previous conclusions. This observation has led to the development and investigation of new logical formalisms, non-monotonic logics [McCarthy, 1980; McDermott, 1980]. Another direction of research combined the idea of logic as a representation language with the theory of automated deduction and constructive logic. This led Kowalski and Colmerauer to the creation of logic programming
Chapter 1. Introduction

[Lloyd, 1987] and the development of the first logic programming language, Prolog [Colmerauer, 1973]. With time, Prolog evolved to incorporate some non-classical, non-monotonic features, which make it closer in spirit to the non-monotonic logics mentioned above. The most important non-monotonic feature of modern Prolog is negation as failure [Clark, 1978; Reiter, 1978].

The language of a logic program [Baral, 1994], like a first-order language, is determined by its object constants, function constants, and predicate constants. Terms are built as in the corresponding first-order language; atoms have the form $p(t_1, ..., t_n)$, where the $t$s are terms and $p$ is a predicate symbol of arity $n$. A rule is an expression of the form

$$A_0 \leftarrow A_1, ..., A_m, \text{not } A_{m+1}, ..., \text{not } A_n.$$ 

Where each $A_i$ is an atom and not is a logical connective called negation as failure. The left-hand side of the rule is called the rule’s head or conclusion; the right-hand side is called the rule’s body or premise. A collection of rules is called a general logic program, they are also referred to as normal logic programs. General logic programs that do not have not are called definite programs. Formulas and rules not containing variables are called ground.

1.4 Thesis Structure

Following the introduction in this chapter, the remainder of the thesis is organized as follows. In Chapter 2, a number of classical notions and protocols of transaction processing are examined and redefined in mobile environments: concepts of transaction, transaction history, equivalent history, history serial and serialization graph, notion of serializability, and locking protocols. The structure of transactions running on mobile hosts is also examined. In addition, a criterion for a serial history is given. Two new concurrency theorems, wormhole theorem and locking theorem, are produced and proved in mobile environments as the outcome of study.

In Chapter 3, a relaxed serializability - epsilon serializability - is formally defined for mobile environments. A relaxed concurrency control method is then proposed to
process rapidly changing data objects for mobile transactions. The proposed approach adopts epsilon serializability to tolerate bounded inconsistency and therefore have improved transaction performance. Additionally, the system architecture and experimental configuration details are given, to demonstrate the feasibility and applicability of implementing concurrent transactions in mobile environments.

In Chapter 4, a rule based knowledge transaction model is presented and formalized to study knowledge transaction in mobile environments. The formalization starts by defining the knowledge transaction representation language, and then imposes a set of rules to capture the features of knowledge transaction in mobile environments. Lastly, a knowledge transaction model is formally defined. A study case is illustrated to show this knowledge transaction model is applicable for practical problem domains in mobile environments.

In Chapter 5, a framework/model for an extended logic programming-based mobile multi-agent system is formalized for mobile environments. Such a system consists of a number of agents connected via wire or wireless communication channels, and the interactions between agents are also modeled in this formalization. Furthermore, knowledge transaction is defined and modeled in such a mobile multi-agent system for knowledge study. A few detailed study cases are presented to demonstrate how to process a knowledge transaction in specific problem domains by using a formalized mobile multi-agent system model.

In Chapter 6, the contributions of this research to the field of mobile computing are summarized and concluded.
Chapter 2

Transaction and Serializability in Mobile Environments

In this chapter, section 2.1 describes the salient features and operation model of mobile environments. Section 2.2 redefines the notion of transaction in mobile environments and examines the structure of transactions running on mobile hosts. Section 2.3 briefly reviews serializability in stationary environments and then discusses a range of concepts of serializability in mobile environments and shows that these concepts still work after proper modifications. A criterion for a serial history is given in mobile environments as well. Section 2.4 proves two concurrency control theorems in mobile environments. Section 2.5 summarizes the work presented in this chapter.

2.1 The Mobile Environment Model

The salient features of mobile environments can be described and summarized as follows [Ahamad, 1995; Mirghafori, 1995]. A Home Server (HS) acts as a permanent storage of Mobile Hosts' (MH) files. There are Mobile Support Stations (MSS) providing services to a MH when it is within their cells (the broadcasting range of MSS). The MSS is connected to the HS via hardwires. The MH is continuously connected to a MSS via a wireless link while accessing data. It may, however, become disconnected either voluntarily or involuntarily. When a MH registers with a MSS, a proxy is created on its behalf. It then performs various services for the MH including caching of profile, broadcasting of sub-profile, acquiring and releasing of locks and management of messages and page validations.

**Home Server (HS):** permanent storage of the mobile host's files.

**Mobile Support Station (MSS):** provides services to the Mobile Host (MH) such as caching, RPC, etc. It also communicates with the MH while it is within its cells. A
MH is registered with a MSS when it enters into its range. The MSS is connected to the HS via hardwires.

Connectivity: the MH is continuously connected to a MSS via a wireless link while accessing data. It may, however, become disconnected either voluntarily or involuntarily.

Proxy: a proxy is created for a MH when it registers with a MSS. The proxy contacts the HS to obtain the profile of the MH. It then caches this profile on the MSS and broadcasts the sub-profile to the MH. It acquires and releases locks for the MH and manages its page invalidation and messages. The proxy buffers messages and invalidation until the MH is ready to receive them.

Centralized data manager: the HS is the central node. It maintains a list of readers and writers of a page. In particular, it must know who has a copy of a valid page. It is the responsibility of the reader to acquire the new page or know that it is reading an old page.

The operational model of the environment is shown in Figure 2.1.

![Figure 2.1: Mobile environment model.](image-url)

There is a centralized database residing in the HS. On each mobile host, MH, there resides a transaction manager which preprocesses transaction operations; a scheduler which controls the relative order in which transaction operations are executed; a recovery manager which is responsible for commitment and abortion management and a cache manager. We assume that the effect of the scheduling is equivalent to an
assignment of a local execution time. The MSS of the MH holds a local clock which synchronizes with the local execution time. The recovery manager updates the list of valid pages by having the MH contact its proxy, which then broadcasts an invalidation report. To have control over the problem of cell migration by the MH, the HS holds a global clock, against which all local clocks held by the MSS synchronize via some master-slave algorithm [Levi, 1990].

2.2 Transaction Notions in Mobile Environments

To define a transaction in mobile environments, firstly recall the definition of a transaction given in Bernstein’s book [Bernstein, 1987] as Definition 1. A transaction is a partial order of operations on a set of data objects. Normally we use letters x, y, … to denote data objects and use the symbol "<<" to denote the ordering relation. We also reserve the letter "r" for the operation "read", "w" for the operation "write", "a" for "abort" and "c" for "commit"…:

a) $T$ is a subset of $\{r[x], w[x] | x \text{ is a data item} \} \cup \{a, c\}$;

b) $a \in T$ iff $c \not\in T$;

c) if any operation $z$ is $c$ or $a$ (whichever is in $T$), for any other operation $p \in T$, then $p << z$;

d) if $\{r[x], w[x]\} \in T$, then either $r[x] << w[x]$ or $w[x] << r[x]$, i.e., conflicting operations are ordered.

In mobile environments, it is imagined that each proxy holds a local clock for time stamping, while the HS holds a master clock against which all local clocks synchronize [Levi, 1990]. With this in mind, a transaction is defined in mobile environments in the following way.

**Definition:** A transaction $T$ is a function from a totally ordered set $\mathcal{T}$ (the local time of a proxy) to the set $\zeta = \text{OP} \times \text{DATA} \cup \text{SYM}$ inducing a partial ordering relation $\ll$ in $\zeta$ defined by $p \ll q$ iff $T(t_1) = p$ and $T(t_2) = q$ such that $t_1 < t_2$, where $<$ is the ordering relation in $\mathcal{T}$.
The sets \( \text{OP}, \text{DATA}, \) and \( \text{SYM} \) are defined as follows:

\( \text{OP} = \{r, w\} \)

\( \text{DATA} = \text{set of data objects} \)

\( \text{SYM} = \{\text{contact-proxy, page-invalidation, sleep, involuntary-sleep, wakeup, commit, abort, move}\} \)

They are subjects to the following conditions:

a) \( T(t) = \text{abort} \) for some \( t \in \mathcal{T} \) iff \( T(t) \neq \text{commit} \) for \( t \in \mathcal{T} \);

b) \( \forall p, q \in \text{OP}, x \in \text{DATA}, \) either \( p[x] \ll q[x] \) or \( q[x] \ll p[x] \);

c) \( \forall p \in \text{OP} \) such that \( T(t) = p \) for some \( t \in \mathcal{T} \), we have \( p \ll a \) or \( p \ll c \);

d) if \( T(t') = w \), and if there is a \( t'' \in [t, t_0] \) such that \( T(t'') = \text{contact-proxy} \) then there is a \( t''' > t'' \) such that \( T(t''') = \text{page-invalidation} \), otherwise \( T(t''') = \text{involuntary-sleep or move} \) for all \( t''' > t_0 \); 

e) if \( T(t) = \text{sleep or involuntary-sleep} \) and \( T(t') = \text{wakeup} \), then \( t < t' \);

Condition a) says that a transaction is aborted only if it is not committed. Condition b) says that conflicting operations of a transaction are ordered. Condition d) says that after each write there is a prescribed time period for the MH to contact the proxy, which would then send out a page invalidation on its behalf. If this does not happen, the MH is deemed to have gone into involuntary sleep. This is an implementation of the invalidation report broadcasting technique of Barbara [Barbara, 1994].

**Example:** A typical transaction \( T \) in mobile environments will look like the following sequence:

\( \text{Contact-proxy} \rightarrow r[x] \rightarrow w[x] \rightarrow \text{contact-proxy} \rightarrow \text{page-invalidation} \rightarrow \ldots \rightarrow \text{contact-proxy} \rightarrow \text{sleep} \rightarrow \text{wakeup} \rightarrow \ldots \rightarrow \text{commit} \)

in which the proxy first acquires the appropriate set of locks for the MH. Once a write is executed, the proxy is asked to broadcast a report of invalidation to proclaim the existence of a new version of \( x \). In this example, the MH, after serving a number of operations of the \( T \), decides to go to sleep voluntarily. This is done by first contacting the proxy, then flushing all its dirty pages (to the MSS) and releasing all the write-
locks it holds. Thus the process of voluntary sleep is equivalent to a partial commit. It is a partial one because there might be operations still not executed. Upon waking up, the MH goes through a process similar to start-up, i.e. there will be fetches of data objects (possibly through the proxy) and lock requests through the proxy. The remaining operations are then executed under the covering of the locks. Finally, the transaction is committed through delayed writes [Ahamad, 1995] to the MSS.

Thus an embedded voluntary sleep causes a transaction to acquire a structure as shown in Figure 2.2. Here the numbers 1,2,3… are used to denote the sequence of activities that happened during transaction. Figure 2.2 illustrates transaction activities for a transaction with an embedded voluntary sleep.

1: fetch of data object, acquisition of locks  
2: execution of transaction operations  
3: flushing of dirty pages to MSS and release of write-locks  
4: repetition of 1  
5: repetition of 2  
6: repetition of 3

Figure 2.2: A transaction with voluntary sleep.

In the case of involuntary sleep, the proxy is not contacted and therefore no broadcasting of the invalidation report occurs. Hence any update done by MH is lost. This is equivalent to there being another write to undo the previous write. If the MH holds write-locks while it goes to sleep involuntarily, and if it does not wake up beyond a certain system prescribed time interval (as measured from the master clock of the HS), the write-locks are cancelled. Thus the overall effect is that of a partial abort. If the wake-up is soon enough, the locks are still with the MH and the
execution of the rest of the operations can proceed. Diagrammatically, Figure 2.3 illustrates the transaction activities for a transaction with an embedded involuntary sleep.

```
1: fetch of data object, acquisition of locks
2: execution of transaction operations
3: canceling of locks by the HS
4: wake-up, fetch data, acquisition of locks
5: execution of transaction operations
6: release of locks
```

It is also possible for a transaction to be started with the MH being in one cell and completed with the MH in another cell. In this case, the transaction T becomes

```
contact-proxy → r[x] → w[x] → contact-proxy → page-invalidation → ... → move → contact-proxy → ... → commit
```

Here we have lumped all the activities related to hand-off and the creation of a new proxy by a new MSS into the operation *move*. The upshot of the discussion is: transactions executed by the mobile hosts always take on a form consisting of several blocks, as shown in Figure 2.4.

```
L U L U L U
```

**Figure 2.3: A transaction with involuntary sleep.**

1: fetch of data object, acquisition of locks
2: execution of transaction operations
3: canceling of locks by the HS
4: wake-up, fetch data, acquisition of locks
5: execution of transaction operations
6: release of locks

**Figure 2.4: A transaction structure.**
The blocks are separated by either voluntary, or involuntary sleeps and the direction of local time runs from left to right. Portions marked by L represent data fetches and lock acquisitions while those marked by U represent lock releases. Hence from now on, we will always represent a transaction $T$ by $\{T_\alpha\}$ where $T_\alpha$ is one of the blocks made up of $T$.

Aside from the notion of transaction, the notions of *transaction histories* and *equivalence of histories* will be continuously checked here in mobile environments.

Classically, let $T = \{T_1, \ldots, T_n\}$ be a set of transactions where each is associated with an ordering relation $\ll_i$. Then a complete history $H$ over $T$ is defined to be a partial order $\prec_H$ such that

a) $H = \bigcup_{i=1}^{n} T_i$

b) $\prec_H = \bigcup_{i=1}^{n} \ll_i$

c) for any two conflicting operations, $p$ and $q$ either $p \prec_H q$ or $q \prec_H p$

In mobile environments, each of the partial ordering $\ll_i$ is induced by a local time $t_i$ maintained by the $i$th MSS. Since it is envisaged that the home server HS maintains a master clock which synchronizes with all the local clocks, it can be imaged that $\ll_H$ is the partial order induced by the global time and hence condition b) continues to be true. This also shows that the concept of complete history remains meaningful.

In a fully connected environment, two histories $H1$ and $H2$ are said to be equivalent if i) they are defined over the same set of transactions; and ii) they order conflicting operations of non-aborted transactions in the same way, i.e. for any conflicting operations, $p_i$ and $q_j$ belonging to transactions $T_i$ and $T_j$, $p_i \prec_{H1} q_j \iff p_i \prec_{H2} q_j$. As noted above, in mobile environments the history $H$ represents a merging of the operations of a set of transactions according to a master clock held by HS. Any two of these mergences are deemed equivalent if they are defined over the same set of transactions and preserve the relative order of the operations.
2.3 Serializability in Mobile Environments

2.3.1 Serializability in Conventional Environments

The transaction dependencies are classically stated in the following way in Gray’s book [Gray, 1993]: A particular object accepts one action at a time. Each action of a transaction is either a read or a write of an object. To reiterate, the definition of object includes not just storage objects but also display objects (windows, menus, and keyboards) and real objects (printers, drill presses and reactor rods). Reading such objects implies sensing their states, while writing them implies changing their states. Objects go through a sequence of versions as they are written by these actions. Reads do not change the object version, but each time an object is written, it gets a new version. If a transaction reads an object, the transaction depends on that object version. If the transaction writes an object, the resulting object version depends on the writing transaction. When a transaction aborts and goes through the undo logic, all its writes are undone. These cause the objects to get new-new versions. For example, imagine that there is a record with version 23 having value A. If your transaction sets the variable to B, the version will increment to 24. If your transaction makes a further update, the version will become 25. If your transaction aborts after that, the record value will be updated back to A. Because versions only increase, the undo update will set the version to 26 (not to 23).

Thinking in these terms suggests a data flow or dependency graph, fragments of which are shown in Figure 2.5.

The transaction execution sequences

T1 READ <o,1>  T1 WRITE <o,2>  T1 WRITE <o,2>
T2 WRITE <o,2>  T2 READ <o,2>  T2 WRITE <o,3>

The dependency graphs

![Figure 2.5: The three forms of transaction dependencies.](image-url)
Figure 2.5 shows three possible execution sequences of reads and writes by two transactions, T1 and T2, operating on versions of an object named o. Originally the object has version <o,1>, but as it is written it acquires versions <o,2>, and <o,3>. The corresponding dependency graphs are shown below the execution sequences. The middle column shows transaction T1 writing version 2 of object o, and then transaction T2 reading that version. This creates a WRITE ->READ dependency between the two transactions. Figure 2.5 has no READ->READ dependencies, because transactions reading the same version of an object create no dependency on one another. Only write actions create versions and dependencies. The only subtle point in Figure 2.5 is the READ->WRITE dependency case. That dependency states that T1 read object o before T2 altered the object.

A dependency graph can be read as a time sequence. If there is an edge from transaction T1 to transaction T2, then T1 accessed an object later accessed by T2, and at least one of these accesses created a new version. In that sense, T1 ran before T2. In a purely sequential execution of the transactions – running T1 to completion, then running T2 to completion – all dependency arrows will point from T1 to T2. But in a parallel execution, dependency arrows can form an arbitrary graph.

The major result of the Gray’s isolation theorems is that any dependency graph without cycles implies an isolated execution of the transactions [Gray, 1993]. On the other hand, if the dependency graph has cycles, the transactions were not executed in isolation. This is fairly intuitive: if the dependency graph has no cycles, then the transactions can be topologically sorted to make an equivalent execution history in which each transaction ran serially, one completing before the next began. This implies that each transaction ran in isolation, as though there were no concurrency; it also implies that there were no concurrency anomalies. If there is a cycle, such a sort is impossible, because there are at least two transactions, such that T1 ran before T2, and that T2 ran before T1.

In the classic concurrency theory, a formal way to specify concurrency control is the serialization graph (SG), where each arc represents the precede relationship [Bernstein, 1987]. The serializability theorem [Bernstein, 1987] mentions that a log H
is of serializability (SR) if its serialization graph is acyclic, an acyclic SG implies an
SR log. That is, a cyclic SG implies a concurrency. But in fact it doesn't discuss how
to give a detail SG to describe the concurrency transaction and only defines a SG
notion to describe the precede relationship.

From the precede references [Gray, 1993] on the transaction concurrency control it is
known that in fact there are only three kinds of dependency cycles. Cycles take one of
only three generic forms, diagrammed in Figures 2.6.

<table>
<thead>
<tr>
<th>Lost Update</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2 READ &lt;o, 1&gt;</td>
<td>T2 WRITE &lt;o, 2&gt;</td>
<td>T1 READ &lt;o, 1&gt;</td>
</tr>
<tr>
<td>T1 WRITE &lt;o, 2&gt;</td>
<td>T1 READ &lt;o, 2&gt;</td>
<td>T2 WRITE &lt;o, 2&gt;</td>
</tr>
<tr>
<td>T2 WRITE &lt;o, 3&gt;</td>
<td>T2 WRITE &lt;o, 3&gt;</td>
<td>T1 READ &lt;o, 2&gt;</td>
</tr>
</tbody>
</table>

![Lost Update Diagram]

Lost Update: Transaction T1's write is ignored by transaction T2, which writes the
object based on the original value <o,1>. A \textit{READ-WRITE-WRITE} sequence is
shown in the diagram, but a \textit{WRITE-WRITE-WRITE} sequence forms the same graph
and is equally bad. To give a simple example of a lost update: suppose you and I are
writing a program together. We both make a copy of the program and change it
independently. We both make a copy of the program and change it
independently. We then return the program to the program library. If each of us is
unaware that the other has updated the program, and if there is no control on updates,
then the program will end up with only my changes or only your changes. One of our
updates will be lost.
Dirty read: $T_1$ reads an object previously written by transaction $T_2$, then $T_2$ makes further changes to the object. The version read by $T_1$ may be inconsistent, because it is not the final (committed) value of the object produced by $T_2$. To continue the programming example, suppose you tentatively install a version of the program in the library, and I use that program. Then you realize that the program is wrong and reverse the change, creating a new version of the program. My read of your tentative program was a dirty read.

Unrepeatable read: $T_1$ reads an object twice, once before transaction $T_2$ updates it and once after committed transaction $T_2$ has updated it. The two read operations return different values for the object. To continue the program example, suppose I use version 1 of your program, and you later install and commit a new version of it (version 2). If I read your program again, it will be the new version that I read. Thus, my first read was not repeatable.

If there was no concurrency, none of these anomalous cases would arise. If three kinds of dependencies can be prevented, then there will be no concurrency anomalies, and the transaction will appear to run in isolation.

Classically, Gray stated the degrees of isolation in the following way [Gray, 1993]. These options are generally called degrees of isolation. The definitions in the previous section were chosen to simplify the explanation of these options. The user’s definition of the four degrees of isolation are given in Gray’s book as below [Gray, 1993]:

**Degree 0:** An isolated transaction does not overwrite another transaction's dirty data if the other transaction is degree 1 or greater.

**Degree 1:** An isolated transaction has no lost updates.

**Degree 2:** An isolated transaction has no lost updates and no dirty reads.

**Degree 3:** An isolated transaction has no lost updates and has repeatable reads (which also implies no dirty reads). This is “true” isolation.
Surprisingly, most systems do not provide true isolation. Implementers make a compromise between correctness and performance. The typical modern system default is to guard against wormholes caused by WRITE->WRITE and WRITE->READ dependencies, but to ignore READ->WRITE dependencies. This goes under the name cursor stability in SQL systems. The ISO and ANSI SQL standards mandate true isolation as the default, but few vendors follow this aspect of the standard. Rather, they allow sophisticated users to request isolation as an option called repeatable reads. They also allow a third option to disable WRITE->READ isolation, called browse access, which allows queries to scan the database without acquiring locks and without delaying other transactions.

### 2.3.2 Serializability Notions in Mobile Environments

Let \( \{T_i\} = \{T_{\alpha}\} \) be a set of transactions in mobile environments, \( \alpha \) and \( \beta \) denote the block number of transaction. A history is assumed to be serial if it runs one block of a transaction at a time. Thus a serial history is one that is of the following form: \( [T_{i\alpha}, T_{j\beta}, \ldots] (\beta > \alpha \text{ for } i = j) \). The notion of serialization graph can also be defined similar way as in the case of a fully connected environment [Gray, 1993] in the following way: Given a history \( H \) over the set \( \{T_i\} \), the serialization graph (SG) for \( H \), denoted by \( SG(H) \), is a directed graph whose nodes are the blocks of the transactions in \( \{T_i\} \) that are committed in \( H \) and whose edges are all \( T_{i\alpha} \rightarrow T_{j\beta} (i \neq j) \), such that one of the operations of \( T_{i\alpha} \) precedes and conflicts with one of the operations of \( T_{j\beta} \) for some \( \alpha \) and \( \beta \). Furthermore, we identify blocks which are in the same transaction with the same value of the first index. Figure 2.7 illustrates a history example \( (H) \) in mobile environments.

![Figure 2.7: A history example in mobile environments.](image-url)
Here there is an operation in $T_{31}$ which is of the form, say, $r[x]$ and which precedes and conflicts with an operation in $T_{22}$, say, $w[x]$ and which in turn precedes and conflicts with an operation in $T_{11}$, say $r[x]$. Thus, in this case, the serialization graph is:

$$T_3 \rightarrow T_2 \rightarrow T_1 \cup T_3 \rightarrow T_2$$

Figure 2.8 illustrates a history having wormhole SG. Consider the following history $H$:

![Figure 2.8: A history of wormhole SG in mobile environments.](image)

This history gives rise to a serialization graph $SG(H)$ which is of the form as shown in Figure 2.9:

![Figure 2.9: A wormhole SG in mobile environments.](image)

This is an example of a wormhole graph which is not allowed in classical serializability but allowed in the mobile environment, since there is clearly no violation of causality with respect to the local times. In fully connected environments, transactions are represented in the serial graph by point-like objects and hence cycles cannot run away from being indicators of causality violation. On the other hand, in
mobile environments, transactions become extended objects and orderings with respect to the local times and are not comparable. This is why cycles without self intersections are allowed and a self intersection always indicates a violation of local causality.

Figure 2.10 illustrates a history having self-intersection wormhole SG. Considering a history of the following type:

![Figure 2.10: A history of self intersection wormhole SG in mobile environments.](image)

This history gives rise to a serialization graph $SG (H)$ which is of the following form as in Figure 2.11:

![Figure 2.11: A SG of a self-intersection wormhole in mobile environments.](image)

This history is of the following form:

$H = T_{34} \parallel T_{22} \parallel T_{12} \parallel \ldots \parallel T_{14} \parallel T_{32} \parallel \ldots$

Apparently it violates the causality of local time because of the form:

$\parallel T_{34} \parallel \ldots \parallel T_{32} \parallel \ldots$
So it is apparent that the history of the intersection wormhole SG is the history violating the causality of local time.

2.4 Serializability Theorems in Mobile Environments

2.4.1 Wormhole Theorem

Based on the discussion above two theorems, wormhole theorem and locking theorem, will be produced in mobile environments, which are the analogues of the wormhole theorem and locking theorem in the classical sense [Gray, 1993].

**Theorem 2.1:** Wormhole Theorem in mobile environments:

*A history is serial if and only if there is no wormhole in the serialization graph.*

Note: Here there is no wormhole in the serialization graph meaning there are no cycles with self-intersection of the above type in SG, and also there are no cycles between any two transaction blocks.

**Proof:** Let \( H \) be a serial history in the execution of a set of transactions \( \{T_i, i = 1, \ldots, n\} \) in mobile environments, each with \( k \) blocks, i.e. \( T_i \) is of the form \( \{T_{i\alpha_1}, \ldots, T_{i\alpha_k}\} \). By definition of a history being serial, there is a sequence of integers \( \{\mu_1, \ldots, \mu_j, \ldots\} \) such that:

\[
H = T_{\mu_1\alpha_1} \parallel T_{\mu_2\alpha_2} \parallel \ldots \parallel T_{\mu_j\alpha_j} \parallel \ldots
\]

\( \mu_j \in \{1, \ldots, n\} \), \( \alpha_j \in \{1, \ldots, k\} \) and for \( \mu_i = \mu_j \)

\( j > i \Rightarrow \alpha_j > \alpha_i \)

The last condition is necessary because of condition b) in the definition of the ordering relation for a history:

\[
\prec_H = \bigcup_{i=1}^{n} \ll_i
\]

In this history, the \( \alpha_1 \)-th block of \( T_{\mu_1} \) is first executed to completion, then \( \alpha_2 \)-th block of \( T_{\mu_2} \) and so on. For any \( \mu_i \) such that \( T_{i\alpha} \) occurs in \( H \), the value of its second index must increase as we traverse \( H \) from left to right. This is because of condition b) in the definition of \( \prec_H \).
Chapter 2. Transaction and Serializability in Mobile Environments

A wormhole with one self-intersection in SG would come from a history of the form:
\[
T_{\mu\alpha} \| \ldots \| T_{\mu\beta} \| \ldots \| T_{\mu\alpha} \| \ldots, \text{ with } \beta > \alpha
\]

A history with a cycle between transaction blocks in SG would come from a history of the form:
\[
T_{i\alpha} \| T_{j\beta} \| T_{i\alpha} \| \ldots
\]

In these two situations, the ordering \(<<_\mu\) is not obeyed.

The converse is clearly true that if there is no wormhole in the serialization graph of a history, the ordering \(<<_\mu\) must be true and therefore a history is serial. □

### 2.4.2 Locking Theorem

Before the derivation of the locking theorem in mobile environments is demonstrated, some familiar definitions need to be re-visited and modified in mobile environments.

A transaction block is well-formed if each \texttt{READ}, \texttt{WRITE}, and \texttt{UNLOCK} action is covered by a corresponding lock, and all locks are released by the end of the transaction block.

A transaction block is two-phase if it has a growing phase in which it only acquires locks, and then a shrinking phase in which it only releases locks.

A history is legal if it does not grant conflicting locks to two different transaction blocks at the same time.

**Theorem 2.2:** Locking Theorem in mobile environments:

*If all transaction blocks are well-formed and two-phase, then any legal history is serial.*

Suppose \(H\) is a legal history of the execution of the set of transaction blocks \(\{T_{\mu1\alpha1}, T_{\mu2\alpha2}, \ldots T_{\mu j \alpha j}, \ldots\}\), each of which is well-formed and two-phase in a classical sense. For each transaction block \(T_{i\alpha j}\), define \(\text{SHRINK}(T_{i\alpha j})\) to be the index of the first unlock step of \(T_{i\alpha j}\) in history \(H\). Formally \(\text{SHRINK}(T_{i\alpha j}) = \min(m | H[m] = <T_{i\alpha j})\).
UNLOCK, o) for some object. Since each block \( T_{\mu_0} \) is non-null and well-formed, it must contain an UNLOCK step. Thus SHRINK is well defined for each transaction block.

To prove locking theorem, the following lemma need to be first proven:

**Lemma 2.1:** if \( T_{\mu_0} <<< T'_{\mu_0} \), then SHRINK\((T_{\mu_0}) <<< SHRINK(T'_{\mu_0}) \)

**Proof of Lemma 2.1:** Suppose \( T_{\mu_0} <<< T'_{\mu_0} \), then suppose there is an object \( o \) and some steps \( m < n \) of history \( H \), such that \( H[m] = < T_{\mu_0}, a, o >, H[n] = < T'_{\mu_0}, a', o >; \) action \( a \) or action \( a' \) is a WRITE (this comes directly from the definition of dependency relation \( DEP(H) \)). Suppose that the action \( a \) of \( T_{\mu_0} \) is a write. Since \( T_{\mu_0} \) is well-formed, step \( m \) is covered by \( T_{\mu_0} \) doing an XLOCK on \( o \). Similarly, step \( n \) must be covered by \( T'_{\mu_0} \) doing an SLOCK or XLOCK on \( o \). \( H \) is a legal schedule, and these locks would conflict, so there must be a \( k_1 \) and \( k_2 \), such that:

\[
m < k_1 < k_2 < n \text{ and } H[k_1] = < T_{\mu_0}, \text{UNLOCK}, o > \text{ and either } H[k_2] = < T'_{\mu_0}, \text{SLOCK}, o > \text{ or } H[k_2] = < T'_{\mu_0}, \text{XLOCK}, o >
\]

Because \( T_{\mu_0} \) and \( T'_{\mu_0} \) are two-phase, all their lock actions must precede their first unlock action; thus, \( SHRINK(T_{\mu_0}) \leq k_1 < k_2 < SHRINK(T'_{\mu_0}) \).

This proves the lemma for the \( a=\text{WRITE} \) case. For the \( a'=\text{WRITE} \) case the proof is almost identical. The SLOCK of \( T_{\mu_0} \) will be incompatible with the XLOCK of \( T'_{\mu_0} \); hence there must be an intervening \( < T_{\mu_0}, \text{UNLOCK}, o > \) followed by a \( < T'_{\mu_0}, \text{XLOCK}, o > \) action in \( H \). Therefore, if \( T_{\mu_0} <<< T'_{\mu_0} \), then SHRINK \((T_{\mu_0}) <<< SHRINK(T'_{\mu_0}) \).

Now the proof of locking theorem is demonstrated as follows:

**Proof of Theorem 2.2:** This proof is by contradiction. For the sake of contradiction, assume \( H \) is not serial history. From the wormhole theorem in the mobile environment, there must be a self-intersection, so there must be a sequence of transaction block like:

\[
\{ T_{\mu_0}, ..., T_{\mu_\beta}, ..., T_{\mu_0}, ... \} \text{ and } \beta > \alpha
\]
Using the lemma, this in turn means that $\text{SHRINK}(T_{\mu\alpha}) \prec \ldots \prec \text{SHRINK}(T_{\mu\beta}) \prec \ldots \prec \text{SHRINK}(T_{\mu\alpha})$. But since $\text{SHRINK}(T_{\mu\alpha}) \prec \text{SHRINK}(T_{\mu\alpha})$ is a contradiction, so $H$ cannot have any self-intersection wormholes, so the history must be serial. □

It should be noted that locking theorem above does not apply to degenerate transactions, which are transactions with one or more blocks performing one of the following operations: locking away data objects that are never read or written upon, and unlocking data objects that have never been locked or terminating without unlocking some of the acquired locks.

### 2.5 Summary

This chapter has rechecked and redefined a range of notions of transaction and serializability in mobile environments. It has shown that classical concepts of transaction processing and concurrency control including notions of transaction, transaction history, equivalent history, history serial, serialization graph, well-formed transactions, two-phase transactions and legal history still work and remain meaningful in mobile environments after proper modifications.

The structure of transactions running on mobile hosts was also examined. A modified notion of a serial history and a criterion for a serial history were given, which is the analogue of the acyclic serial graph criterion in conventional environments. Lastly, two concurrency control theorems, wormhole theorem and locking theorem, were produced and proven in mobile environments.

Exploring those important concepts and protocols of transaction and serializability is believed to be the foundation of transaction and concurrency study in mobile environments. To my best knowledge, no prior work has been done in this area.
Chapter 3

Concurrency Control for Mobility Support

The methods and techniques used to ensure serializability become much more complex in mobile environments. This chapter addresses the issue of serializability (SR) and concurrency control for mobile transaction; defines a relaxed serializability correctness criterion - epsilon serializability (ESR) in mobile environments; and proposes a relaxed concurrency control method to process rapidly changing data objects for mobile applications. Compared with other concurrency control methods, the proposed approach has several major advantages.

Section 3.1 firstly reviews and compares different relaxed serializability criteria and concurrency control approaches in both stationary environments and mobile environments, and then introduces the notion of epsilon serializability and ESR concurrency control method in conventional environments. Section 3.2 formally defines and extends epsilon serializability to mobile environments. Section 3.3 proposes the general design of a relaxed concurrency control method for mobility support. Section 3.4 then presents the details of a concurrency method for discrete mobility support. Section 3.5 discusses the details of a concurrency method for continuous mobility support. Several global algorithms are used to maintain global consistency in distributed transaction processing. Section 3.6 provides the system architecture and experimental configuration used to implement concurrent mobile transactions in laboratory environments. Section 3.7 summarizes the work in this chapter.

3.1 Concurrency Control Overview

3.1.1 Relaxed Serializability Criteria and Concurrency Control

Serializability [Bernstein, 1987; Papadimitriou, 1979] is the standard notion of correctness in transaction processing. It maintains database consistency and has efficient implementation through concurrency control algorithms that require low
run-time overhead and relatively high transaction throughput. However, the notion has its limitations. For example, serializability requires that concurrent transactions be scheduled in a serializable order. When the number of concurrent transactions increases, the effective level of concurrency tends to decrease, forcing many transactions to wait or abort [Franaszek, 1985; Agrawal, 1987; Yu, 1991].

The investigation of correctness criteria weaker than SR is not new. Historically the earliest, Gray et al. have defined 4 degrees of consistency in terms of locking [Gray, 1993]. Garcia-Molina et al. have introduced the weak consistency class of read-only transactions [Garcia-Molina, 1982]. Their Wait-for List Centralized Algorithm (WLCA) gives each local query a consistent view of the data, but the union of all schedules may be inconsistent. Du et al. have introduced quasi-serializability (QSR) as a correctness criterion for federated databases [Du, 1989]. QSR may be seen as a generalization of read-only weak consistency, since QSR also maintains locally consistent views, but does not guarantee global SR. Korth et al. have introduced a formal model that allows transactions to specify pre-conditions and post-conditions to be satisfied before and after the transaction execution, respectively [Korth, 1988]. For distributed divergence control, a large body of literature exists concerning various approaches to effectively supporting concurrent transaction and query processing [Bober, 1992; Breitbart, 1991; Elmagarmid, 1990; Georgakopoulos, 1991; Herlihy, 1990] in conventional environments. These approaches typically employ multiple versions of data to eliminate Read/Write conflicts between update transactions and read-only queries. In the area of real-time database, Ho et al. have developed a notion called similarity [Ho, 1997], which also allows relaxed serializability. Another paper, by Bober et al. [Bober, 1992], presents an algorithm that allows weaker forms of consistency in a way similar to Read-Only transactions [Garcia-Molina, 1982]. However, their queries still see a transaction-consistent view of the database, possibly with some older version that does not contain all the updates. Krishnakumar’s paper [Krishnakumar, 1996] discusses a method to perform mobile sales transactions using site-transaction escrow methods [Barbara, 1992; Krishnakumar, 1992], but the data type used for discussion is very limited. Walborn et al. generalize the usage of escrow techniques by exploiting object semantics to facilitate autonomous and disconnected operations in mobile database applications [Walborn, 1995]. Gray et al. propose a
two-tier replication algorithm that allows mobile applications to propose tentative transactions to the mobile clients [Gray, 1996]. In [Barbara, 1997], the concept of certification reports is introduced as a way of supporting transactions in mobile environments. The transaction management model in [Bhalla, 2003] concentrates on embedded concurrency control for transaction updates by mobile clients. Research in [Dang, 2003; LAM, 2000; Datta, 1997; Kayan, 1999; Leong, 1997; Ulusoy, 1998; Xuan, 1997] studies the concurrency control in mobile distributed real-time database systems.

### 3.1.2 Epsilon Serializability in Conventional Environments

To alleviate the limitations of SR, the notion of epsilon serializability has been proposed in [Pu, 1990; Pu, 1991a; Pu, 1991b; Pu, 1992; Ramamrithan, 1994] for conventional environments. The purpose of ESR [Pu, 1990] is to explicitly allow some limited amount of inconsistency, called fuzziness, denoted by $\varepsilon$. ESR allows a limited amount of inconsistency in transaction processing. These inconsistencies are controlled by divergence control (DC) algorithms [Wu, 1992] that are extensions of classic concurrency control algorithms. An epsilon transaction (ET) is a traditional transaction plus a constraint specification, called $\varepsilon$-spec, which tells the transaction processing (TP) system how much inconsistency the ET can tolerate. For a typical case, a query ET (read-only) is allowed to view inconsistent data due to non-serializable interleavings of operations with concurrent update ETs, where an update ET may change the database status. Such non-serializable interleavings can increase the efficiency of a transaction processing system performance through added query ET concurrency, while update ETs still preserve database consistency. ESR alleviates the strictness of serializability in transaction processing by allowing for limited inconsistency. ESR provides well-defined correctness criteria, efficient algorithms, limited explicitly controlled inconsistency, and higher concurrency and availability.

The notion of *epsilon serializability* in conventional environments can be described as follows: two operations, $a$ and $b$, are assumed to conflict, denoted by $\text{conflict}(a, b)$, if both operations are on the same data item and one of them is a read operation. Assume that a set of transactions, $T$, is given. For two different transactions $t_i \in T$ and...
Chapter 3. Concurrency Control for Mobility Support

If \( t_i \in T \), we say that \( t_i \) conflicts with \( t_j \), denoted by \( t_i \mathcal{C}_{SR} t_j \), if an operation \( a \), issued by \( t_i \) and an operation \( b \), issued by \( t_j \), conflict, and \( a \) precedes \( b \) in the history. For a transaction \( t \in T \), we say that \( (t \mathcal{C}^*_{SR} t) \) is a conflict cycle, where \( C^*_{SR} \) is the transitive-closure of \( C_{SR} \). A history over \( T \) is serializable, if and only if there does not exist a conflict cycle. For two transactions \( t_i \in T \) and \( t_j \in T \), we say that \( t_i \) epsilon-conflicts with \( t_j \), denoted by \( t_i \mathcal{C}_{ESR} t_j \), if \( t_i \mathcal{C}^*_{SR} t_j \) and \( \neg \text{safe}(t_j) \). That is, \( t_i \) conflicts with \( t_j \) and the safe condition of \( t_j \) is violated after \( t_j \) invokes its operation. A conflict cycle that does not contain a \( C_{ESR} \) edge is a safe conflict cycle. Otherwise, it is an unsafe conflict cycle. A history is epsilon serializable if and only if there does not exist an unsafe conflict cycle.

A definition of epsilon serializability [Ramamrithan, 1994] is established by assuming existence of a safe condition for a transaction \( t \) (denoted by \( \text{safe}(t) \)):

\[
\begin{align*}
\text{Fuzziness}^\text{import}_t &\leq \text{Limit}^\text{import}_t \\
\text{Fuzziness}^\text{export}_t &\leq \text{Limit}^\text{export}_t
\end{align*}
\]

\( \text{Fuzziness}^\text{import}_t \) is the import fuzziness of \( t \), which could be an approximation of the "actual" import fuzziness by a mechanism (e.g., a divergence control method). \( \text{Fuzziness}^\text{export}_t \) (the export fuzziness of \( t \)) is defined in a similar way. \( \text{Limit}^\text{import}_t \) is the import fuzziness limit of \( t \) and \( \text{Limit}^\text{export}_t \) is the export fuzziness limit of \( t \).

Differing from WLCA [Garcia-Molina, 1982], which is a particular algorithm for replicated distributed databases, ESR is implemented by many divergence control methods for both centralized and distributed databases [Pu, 1991a; Pu, 1992]. QSR [Du, 1989] allows both updates and queries that may be globally inconsistent, but locally consistent. In contrast, ESR allows bounded local inconsistency for query Epsilon Transactions (ET). In Korth’s model [Korth, 1988], consistency specification is on data object states, while ESR allows users to specify the amount of inconsistency tolerated by the ETs. In contrast with multiple version approaches for distributed concurrency reviewed above in this chapter, ESR does not require multiple versions of data, thus does not incur extra storage overhead. However, ESR allows query ETs to access inconsistent data in a controlled way. ESR has three main advantages over previous concurrency models: (1) ESR is a general framework, applicable to a wide range of application semantics; (2) ESR is upward-compatible,
since it reduces to serializability as $\varepsilon$-spec $\to 0$; and (3) a number of efficient algorithms support ESR.

An important assumption in ESR is the existence of a distance function and an associated regular geometry in the database state space. A database state is the set of all object values. A database state space is the set of all possible database states. A database state space $S_{DB}$ is a metric space if it has the following properties:

- A distance function $dist(u,v)$, defined over every pair of states $u, v \in S_{DB}$ on real numbers.
- Triangle inequality. For every $u, v, w \in S_{DB}$, $dist(u,v) + dist(v,w) \geq dist(u,w)$.
- Symmetry. For every $u, v \in S_{DB}$, $dist(u,v) = dist(v,u)$.

Many of the real world database state spaces fulfill this requirement. For example, integers and real numbers in banking, airline and scientific data are spaces that have a natural definition of distance function and the regular geometry of a metric space.

### 3.1.3 ESR Divergence Control Algorithms

In ESR the limited inconsistency is automatically maintained by the divergence control (DC) methods in a way similar to serializability by concurrency control (CC) methods. However, DC for ESR allows more concurrency than CC for SR. The DC methods in ESR are designed in such a way that read-only transactions need not be serializable with other update transactions while update transactions must be serializable among themselves.

The feasibility of epsilon serializability has been demonstrated by showing the design of concrete, representative divergence control methods for ESR. The total number of non-serializable R/W operations that a query ET can import or an update ET can export is first used as the inconsistency specification to design DC methods. The amount of inconsistency that a query ET may see is bounded by $N\times K$, where $N$ is the total number of non-SR conflicts the query ET has imported, and $K$ is the maximum value change for each update to a data item by an update ET. One advantage of these
methods is that they are not application-specific. As a result, users can benefit from ESR through high-level and simple specification of inconsistency tolerance without reference to the specifics of the DC methods. Three such DC methods have been described in [Wu, 1992] for centralized transaction processing systems: two-phase locking DC, timestamp ordering DC and optimistic DC.

In Pu’s paper [Pu, 1993], distributed divergence control algorithms for epsilon serializability are presented for both homogeneous and heterogeneous distributed databases. For homogeneous distributed databases, it is assumed that the local orderings of sub-ETs are the same among all the component databases. Two distributed divergence control algorithms are described: Strict 2-Phase Locking Distributed Divergence Control algorithm (S2PLDDC), and Optimistic Distributed Divergence Control algorithm (ODDC) using weak locks. For heterogeneous distributed databases, the local orderings of all the sub-ETs of a distributed ET may not be the same. Hence the global orderings of distributed ETs may not be serializable even though all the sub-ETs are locally serializable. Thus, a global mechanism is needed to guarantee ESR. The Superdatabase architecture [Pu, 1988] is used as a general model and a corresponding distributed divergence control algorithm presented based upon it.

For each ET, inconsistency specification is divided into two parts: imported inconsistency denoted by $\text{Implimit}$, and exported inconsistency denoted by $\text{ExpLimit}$ respectively. When a conflict between a read operation in a query ET and a write operation in an update ET causes the execution of a diverge from a serial history, it is said that the query ET imports fuzziness from the update ET and the update ET exports fuzziness to the query ET. Import fuzziness ($Z_{\text{import}}$) is the amount of inconsistency that a query introduces from the database or other ETs through non-serializable operations. Export fuzziness ($Z_{\text{export}}$) is the amount of fuzziness that an ET introduces to other ETs. In the ESR method, $Z_{\text{import}}$ is requested to be less than a Limit amount of inconsistency imported into a transaction ($\text{ImpLimit}$), and $Z_{\text{export}}$ is requested to be less than a Limit amount of inconsistency exported from a transaction ($\text{ExpLimit}$), otherwise a query transaction or update transaction will be aborted. If the potential fuzziness is below each ET's specification, the lock is granted.
It has been shown by Wu et al. [Wu, 1992] that the ESR centralized divergence control method guarantees two following conditions:

\[ Z_{\text{import}} \leq \text{ImpLimit} \]
\[ Z_{\text{export}} \leq \text{ExpLimit} \]

In a distributed database, a distributed ET consists of one or more sub-ETs, each of which resides on one site. For homogeneous distributed databases, the local orderings of all sub-ETs are the same. Let local fuzziness, denoted by \( Z_{\text{local}} \), be the fuzziness detected and maintained locally for a sub-ET by a centralized divergence control algorithm, and total fuzziness, denoted by \( Z_{\text{total}} \), be the fuzziness of a distributed ET. In the ESR distributed divergence control algorithm [Pu, 1993], the total fuzziness of a distributed ET is equal to the sum of the local fuzziness of all its sub-ETs:

\[ Z_{\text{total import}} = \sum_{\text{sub-ET}} Z_{\text{local import}} \leq \text{ImpLimit} \]
\[ Z_{\text{total export}} = \sum_{\text{sub-ET}} Z_{\text{local export}} \leq \text{ExpLimit} \]

In the heterogeneous distributed databases, there is one more fuzziness in existence because the local orderings of the sub-ETs may be different. This is known as global fuzziness. The global fuzziness is presented as \( Z_{\text{global}} \) [Pu, 1993]:

\[ Z_{\text{total import}} = \sum_{\text{sub-ET}} Z_{\text{local import}} + Z_{\text{global import}} \leq \text{ImpLimit} \]
\[ Z_{\text{total export}} = \sum_{\text{sub-ET}} Z_{\text{local export}} + Z_{\text{global export}} \leq \text{ExpLimit} \]

### 3.2 Defining Epsilon Serializability in Mobile Environments

The characteristics of the mobile environments make serializability too strong as a correctness criterion and not suitable for mobile transactions [Daniel, 1999; Lam, 1999; Lee, 1999, Ulusoy, 1998]. The relaxed serializability and concurrency control methods are more desirable, and required, for mobile transactions to improve performance or availability than in conventional environments. In this section, epsilon serializability is formally defined in mobile environments as a relaxed serializability.
ACTA framework [Chrysanthis, 1991a; Chrysanthi, 1990; Chrysanthis, 1991b] is used here to introduce the notion of conflicts between operations and discuss the dependencies induced between transactions when they invoke conflicting transactions. For a given state \( s \) of a data item, \( \text{return}(s, a) \) denotes the output produced by operation \( a \), and \( \text{state}(s, a) \) denotes the state produced after the execution of \( a \). \( \text{Value}(s, P) \) denotes the value of predicate \( P \) in state \( s \). Given a history \( H \) of events relating to transactions in \( T \), \( H^{(x)} \) is the projection of the history containing the operation invocations on a data item \( x \). \( H^{(x)} = a_1 < a_2 < ... < a_n \), indicates both the order of execution of the operations, \( ( a_i \text{ precedes } a_{i+1} ) \), as well as the functional composition of operations. Thus, a state \( s \) equals the state produced by applying the history \( H^{(x)} \) corresponding to the sequence of operations on the data item’s initial state \( s_0 \) \( (s = \text{state}(s_0, H^{(x)})) \). For brevity, \( H^{(x)} \) is used to denote the state of a data item produced by \( H^{(x)} \), implicitly assuming initial state \( s_0 \).

The definition of operation conflict [Ramamritham, 1995] in conventional environments can be extended to mobile environments without any modification. This definition is recalled as Definition 1:

**Definition 1:** Two operations \( a \) and \( b \) conflict in a state produced by \( H^{(x)} \), denoted by \( \text{conflict}(H^{(x)}, a, b) \), iff

\[
\begin{align*}
\text{(state}(H^{(x)} &< a, b) \neq \text{state}(H^{(x)} < b, a)) \lor \\
\text{return}(H^{(x)}, a) &\neq \text{return}(H^{(x)} < b, a)).
\end{align*}
\]

As one of the upshots of the discussion in Chapter 2, it is known that transactions executed by the mobile hosts always take on a form consisting of several blocks. Given a set of transactions \( T \) in mobile environments, \( T \) can be denoted by \( T = \{t_i\} = \{t_{ia}\} \), where \( t_i \) is one of the transactions made up of \( T \) and \( t_{ia} \) is one of the transaction blocks made up of \( t_i \).

Epsilon serializability is formally defined and extended to mobile environments as follows:
Definition 2: Let $t_i$ and $t_j$ be transactions $\in T$ in mobile environments, whose events are recorded in history $H$. $C_{ESR}$, a binary relation on transactions in $T$, is defined as follows:

$$(t_i C_{ESR} t_j), \ t_i \neq t_j, \iff \exists a, b \ (\text{conflict}(H^{(x)}, a_{t_{i\alpha}}[x], b_{t_{j\beta}}[x]) \land (a_{t_{i\alpha}}[x] \rightarrow b_{t_{j\beta}}[x]) \land \text{value}(\text{state}(H^{(x)}) < a, b, \neg \text{Safe}(t_j, x))).$$

where $a_{t_{i\alpha}}[x]$ denotes operation $a$ invoked by transaction block $t_{i\alpha}$ on data item $x$, $b_{t_{j\beta}}[x]$ denotes operation $a$ invoked by transaction block $t_{j\beta}$ on data item $x$. $(a_{t_{i\alpha}}[x] \rightarrow b_{t_{j\beta}}[x])$ implies that $a_{t_{i\alpha}}[x]$ appear before $b_{t_{j\beta}}[x]$ in $H$.

In other words, $t_i$ and $t_j$ are related by $C_{ESR}$ if and only if one transaction block $t_{j\beta}$ of transaction $t_j$ has invoked an operation which conflicts with a previous operation invoked by one transaction block $t_{i\alpha}$ of transaction $t_i$, and they violate one of the invariants that constitute the predicate Safe. $C_{ESR}$ denotes the ordering requirements imposed by conflicts under epsilon serializability.

Definition 3: In mobile environments, A cycle formed by transactions $t_0, t_1, t_2, \ldots, t_{n-1}$, has a $C_{ESR}$ edge iff

$$\exists i, 0 \leq i < n, (t_i C_{ESR} t_{(i+1 \mod n)}).$$

If the serialization graph has a cycle consisting of a $C_{ESR}$ edge, then the history is not epsilon serializability.

Definition 4: In mobile environments, A history $H$ is epsilon serializability iff, in the serialization graph which corresponds to the $C_{ESR}$ relations induced by the history, there is no cycle that has a $C_{ESR}$ edge.

A formal definition of epsilon serializability in mobile environments is established by assuming existence of a safe condition for a transaction $t_i$ (denoted by $\text{safe}(t_i)$):

$$\text{Fuzziness}_{i_{\text{import}}} = \sum_{\alpha=1}^{\delta_i} \text{Fuzziness}_{i_{\alpha \text{import}}} \leq \text{Limit}_{i_{\text{import}}}$$
Chapter 3. Concurrency Control for Mobility Support

\[ Fuzziness_{ti}^{\text{import}} = \sum_{\alpha=1}^{n} Fuzziness_{t_{i\alpha}}^{\text{import}} \leq \text{Limit}_{ti}^{\text{import}} \]

Where \( Fuzziness_{ti}^{\text{import}} \) is the import fuzziness of transaction \( ti \). For a mobile transaction, \( Fuzziness_{ti}^{\text{import}} \) is the sum of import fuzziness of all transaction blocks in transaction \( ti \), \( Fuzziness_{t_{i\alpha}}^{\text{import}} \) is the import fuzziness introduced by transaction block \( t_{i\alpha} \). \( Fuzziness_{ti}^{\text{export}} \) is the sum of export fuzziness of all transaction blocks in transaction \( ti \), \( Fuzziness_{t_{i\alpha}}^{\text{export}} \) is the export fuzziness introduced by transaction block \( t_{i\alpha} \). \( \text{Limit}_{ti}^{\text{import}} \) is the import fuzziness limit of transaction \( ti \) and \( \text{Limit}_{ti}^{\text{export}} \) is the export fuzziness limit of transaction \( ti \).

3.3 A Relaxed Concurrency Control Method for Mobility Support

Here, a relaxed concurrency control method is proposed to process mobile transactions. This approach can provide concurrency control for rapidly changing data, while limited inconsistency is tolerated. An overview of the method is given in this section; section 3.4 and section 3.5 will provide the details. The discussion is based upon the generic form of mobile environments discussed in Chapter 2.

Our proposed concurrency control method is applicable to both discrete mobility situation and continuous mobility situation mentioned in [Krishnakumar, 1996]. Simply put, discrete mobility situation is one in which the mobile user will turn off his mobile host and disconnect from the current MSS, before connecting to another MSS cell. In this situation the mobile user does his transaction processing entirely in the new MSS cell. In the continuous mobility situation a mobile host user moves from a current MSS cell to a new MSS cell and continues his transaction processing without turning off the mobile host. In our proposed concurrency support approach, concurrency control for discrete mobility is achieved by modifying and extending the ESR centralized divergence control method [Wu, 1992] to every MSS cell of the mobile environment. Four global algorithms are adopted to achieve concurrency support for continuous mobility: mobile version of ESR distributed divergence control method; redistribution mobility scheme; two-phase commit protocol; and mobility check-pointing algorithm [Acharya, 1994].
The mobile version of ESR divergence control method is adopted in the proposed concurrency control approach. As a result, this method is particularly desirable for one class of application that involves rapidly changing data, which can be characterized by: (1) an enormous amount of data; (2) sharing of data among large numbers of simultaneous users (human and programs); (3) frequent changes to data outside the control of the system; (4) soft real-time constraint, where updates must be accepted within a time interval or they will be missed; and (5) the acceptance by application designers that decisions may be based on obsolete or inconsistent data [Kaiser, 1990]. The major application areas are queries in a stock market scenario and rapid estimations of total account balance in a bank.

3.4 Concurrency Support for Discrete Mobility

3.4.1 Discrete Mobility

The situation referred to as discrete mobility in Krishnakumar’s paper [Krishnakumar, 1996] can be explained with the help of a simple example. Consider a bank manager using his portable computer to query the daily bank balance at one moment. This mobile host user starts his query at one MSS cell. When he completes all transactions, he turns off his portable computer, disconnects from the MSS and moves to another MSS cell. Now the user can start another session with the new MSS. In this situation the mobile user handles the transaction processing in the separate MSS cells. In our proposed concurrency control method, since all the transactions are processed in one site, concurrency control on every site can be achieved by extending centralized ESR control method to every MSS cell of mobile environments. No matter whether he is in the old MSS cell or in the new MSS cell, the bank manager can query the bank balance, while at the same time other bank customers can execute banking activities such as deposits or withdrawals. The bank manager gives the balance limit that is the tolerated inconsistency limit for his query transaction.

The notion of epsilon serializability has been formally defined and extended to mobile environments above, thus ESR divergence control methods can be modified to
implement concurrency control in mobile environments with proper modification. When the query transaction is processed, the actual inconsistency is estimated and computed according to the mobile version of centralized ESR divergence control method. If the estimated inconsistency exceeds the inconsistency limit, the query transaction is aborted otherwise the transaction is committed. As discussed in section 3.2, the safety condition for tolerated inconsistency in mobile environments is as follows:

\[
Fuzziness_{i_t}^{import} = \sum_{\alpha=1}^{n} Fuzziness_{i_{t\alpha}}^{import} \leq \text{Limit}_{i_t}^{import}
\]

\[
Fuzziness_{i_t}^{export} = \sum_{\alpha=1}^{n} Fuzziness_{i_{t\alpha}}^{export} \leq \text{Limit}_{i_t}^{export}
\]

Each variable or constant herein has been specified in section 3.2.

Query transactions will be aborted when the estimated inconsistency exceeds the tolerated inconsistency limit. It is clear that update transactions can not be aborted because the bank cannot refuse the customers’ requests.

It is argued that optimistic divergence control is the best approach [Kistler, 1992] in mobile environments, However, using an optimistic approach does involve some difficulties, especially in mobile environments, in which long duration partitions will cause a greater incidence of apparent write-write conflicts than in stationary environments [Joseph, 1997]. By adopting epsilon serializability in mobile environments, the transactions have less wait and abort activities, and thus the two-phase lock becomes an acceptable approach. When a mobile host goes to voluntary sleep, it will release all holding locks before sleep. When a mobile host goes to involuntary sleep, the lock manager will release the mobile host’s lock eventually, after a time bound. In either case, the lock can be acquired and released. Two approaches, two-phase locking and optimistic divergence control, will be discussed to show the feasibility and flexibility in implementing the proposed concurrency control method.

3.4.2 Two-Phase Locking
The ESR two-phase lock compatibility matrix is modified in mobile environments as shown in Table 3.1:

<table>
<thead>
<tr>
<th></th>
<th>QL(ET_α)</th>
<th>RL(ET_α)</th>
<th>WL(ET_α)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QL(ET_β)</td>
<td>AOK</td>
<td>AOK</td>
<td>LOK-1</td>
</tr>
<tr>
<td>RL(ET_β)</td>
<td>AOK</td>
<td>AOK</td>
<td>—</td>
</tr>
<tr>
<td>WL(ET_β)</td>
<td>LOK-2</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 3.1 ESR Two-phase lock matrix in mobile environments.

QL(ET_α): denotes a read lock acquired by a block α of a query ET.
RL(ET_α): denotes a read lock acquired by a block α of an update ET.
WL(ET_α): denotes a write lock acquired by a block α of an update ET.

In the table, columns represent locks held and rows represent locks requested. The squares marked AOK are always compatible; those marked dash are always incompatible. LOK-1 (for limited OK, case 1) allows query ETs to read uncommitted data, while LOK-2 (for limited OK, case 2) allows update ETs to overwrite data that query ETs are reading.

In mobile environments, to control the amount of inconsistency allowed in a query ET, the lock management at LOK-1 and LOK-2 is further refined. The following procedure is performed:

- Every time, when a query ET block requests a QL(ET) lock under LOK-1, the Import_Accumu (accumulated import fuzziness) of the query ET and the Export_Accumu (accumulated export fuzziness) of the conflicting update ET are first checked to see if they would exceed their import and export limit after an increment of import and export fuzziness introduced by this epsilon transaction block.

- Each time, when an update ET block requests a WL(ET) lock under LOK-2, all the query ET holding the conflicting QL(ET) lock have their Import_Accumu
checked to see if they would exceed their import limits after an increment. However, the update ET must check its own Export_Accumu against an increment that comes from all query ETs holding the QL(ET) lock.

• If either the update ET’s Export_Accumu exceeds its export limit or any query ET’s Import_Accumu exceeds its import limit, then the requester should either wait or abort by returning a lock failure to the requesting ET or breaking some locks already held by other ET’s blocks.

3.4.3 Optimistic Divergence Control

Optimistic concurrency control allows the read or write operations of any transaction to proceed immediately when the requests arrive, thus it is arguably the best approach for mobile environments [Kistler, 1992]. However, a transaction cannot be committed until a validation is performed to ensure SR. If any non-serializable execution is detected during the validation, the transaction is aborted and restarted.

Using an optimistic divergence control (ODC) method to control concurrency for epsilon transactions in mobile environments, the following procedure is performed:

• The ODC manager first marks the query ET block as non-SR when resolving lock conflicts between a query ET block and an update ET block. Only when the inconsistency accumulation of either a query ET or an update ET exceeds its limit, is the query ET block marked for abort.

• Any non-SR conflict detected between update ET blocks will be marked for abort to guarantee the database consistency.

• To control the inconsistency, the ODC manager maintains a table of locks for each data item. When a weak lock on a data item x is requested by a query ET block, if a strong lock on x is currently held by an update ET block the Import_Accumu of query ET is incremented by a fuzziness imported by this query ET block.
• If the Import_Accumu of query ET has not reached its limit, then the Export_Accumu of update ET is also checked to see if it will reach its limit after an increment.

• If neither of them reaches limit, then this query ET block is marked as non-SR, otherwise it is marked for abort.

• When committing an ET (query or update), the ODC manager checks to see if any transaction block has been marked for abort. If not, it can be committed, otherwise it is aborted.

• In committing an update ET, the weak locks of the exclusive mode of the update ET are also changed into strong locks. During lock conversion on a data item x for an update ET, other update ET blocks holding a weak lock on x are marked for abort. However, for query ET blocks holding a weak lock on x, the query ET block is marked as non-SR if neither its Import_Accumu nor the Export_Accumu of the update ET exceeds its limit.

3.5 Concurrency Control for Continuous Mobility

3.5.1 Continuous Mobility

The case referred to as continuous mobility in Krishnakumar’s paper [Krishnakumar, 1996] is considered in this section. In this situation the mobile user is running a long transaction, such as a complicated bank balance query. When the mobile user moves from one MSS cell to another MSS cell, he continues and completes his balance query in the new cell. This means the transaction involves the different MSS cells. In this section only the situation where a transaction involves two different MSS cells is considered. This is to avoid making things unnecessarily complicated while at the same time keeping the discussion general.

To support continuous mobility, our proposed concurrency control method adopts four global algorithms to guarantee the global consistency: the mobile version of ESR distributed divergence control method; the redistribution mobility scheme; the commit protocol; and the check-pointing mobility algorithm [Acharya, 1994]. The
mobile version of ESR distributed divergence control method and redistribution mobility scheme are proposed for distributed concurrency control in mobile environments, while commit protocol and check-pointing are employed to guarantee the same transaction outputs and global orderings for implementation of commitment and abort.

A distributed ET consists of one or more sub-ETs, each of which resides on one site. For homogeneous distributed databases, the local orderings of all sub-ETs are the same. Let local fuzziness, denoted by $Fuzziness_{local}$, be the fuzziness detected and maintained locally for a sub-ET, and total fuzziness, denoted by $Fuzziness_{total}$, be the fuzziness of a distributed ET. In the ESR distributed divergence control algorithm [Pu, 1993], the total fuzziness of a distributed ET is equal to the sum of the local fuzziness of all its sub-ETs in homogeneous distributed system. The same safety condition is used in the proposed concurrency method for continuous mobility support as follows:

$$Fuzziness_{total\ import} = \sum_{sub-ET} Fuzziness_{local\ import} \leq Limit_{import}$$

$$Fuzziness_{total\ export} = \sum_{sub-ET} Fuzziness_{local\ export} \leq Limit_{export}$$

In mobile environments, transaction is made up of transaction blocks, so every sub-ET is actually made up of transaction blocks as well. The Fuzziness introduced by every sub-ET is the sum of fuzziness introduced by its every transaction block. As with the implementation of discrete mobility support, both two-phase locking and optimistic divergence control will be presented for continuous mobility support.

A Superdatabase architecture [Pu, 1988] is used in the paper [Pu, 1993] as a general model for heterogeneous distributed database to maintain global consistency in the ESR. The ESR Superdatabase divergence control algorithm provides substantial autonomy in a heterogeneous environment, however, finding a conflicting committed ET against a committing ET can be expensive unless further optimization is done. There are some alternative schedulers for heterogeneous concurrency control that enforce global serializability, such as one could distribute and order at transaction creation time [Elmagarmid, 1990]. Another possibility is the use of forced local
conflicts to obtain a rigorous schedule [Georgakopoulos, 1991]. Whether these algorithms can be easily modified to support heterogeneous distributed divergence control in mobile environments can be the subject of further research. In our proposed concurrency control method for mobility support, it is assumed that the local orderings of sub-ETs is simply the sum of the local fuzziness of all its sub-ETs, that is, only the homogeneous distributed system is considered.

3.5.2 Two-Phase Locking with Mobility Redistribution Scheme

In conventional environments, the ESR distributed divergence control method uses a demarcation protocol [Barbara, 1992] to dynamically reconfigure the inconsistency limit of different sites. In mobile environments, a new redistribution scheme is needed to dynamically redistribute the inconsistency limit between cell A and cell B, since the demarcation protocol cannot be suitably used when the mobile user moves from cell A to cell B. A mobility redistribution scheme applicable to a moving scenario is provided in our approach to dynamically redistribute the inconsistency limit between cell A and cell B. The scheme is motivated by Krishnakumar’s work [Krishnakumar, 1996]. However, different from this work, the redistribution scheme is used to reconfigure the tolerated inconsistency limit (import limit and export limit) in our method.

In order to explain clearly, it is assumed that the mobile user moves from cell A to cell B with the original allocated import limit Implimit in cell A and cell B being Implimit$_A$ and Implimit$_B$. Our proposed concurrency control method supports continuous mobility in the following way: in separate cells, i.e. cell A and cell B, the concurrency is supported by the discrete mobility support method discussed in section 3.4. When the mobile user moves from cell A to cell B, the mobility redistribution scheme is used to redistribute Implimit$_A$ and Implimit$_B$, meanwhile the transaction will be processed continuously. The redistribution scheme works as follows: when the mobile user moves from cell A to cell B, the transaction inconsistency he is using will be taken from cell A to cell B. This causes a redistribution of Implimit in cell A and cell B. Concretely, this transaction inconsistency is subtracted from Implimit$_A$ and added to Implimit$_B$. For example,
Implimit_A = 100 and Implimit_B = 100, when the bank manager starts to query the bank balance from cell A. While the manager is querying the balance, bank customers are withdrawing from or depositing into their accounts. It is assumed that the transaction inconsistency in cell A is 40 at this moment, which is estimated and computed by the discrete mobility support scheme. When the mobile user moves from cell A to cell B, the redistributed scheme method is used here to redistribute Implimit_A and Implimit_B according to the following formula:

\[\text{Implimit}_A = \text{Implimit}_A - \text{Inconsistency}\]
\[\text{Implimit}_B = \text{Implimit}_B + \text{Inconsistency}\]
\[\text{Inconsistency}_A = \text{Inconsistency}_A - \text{Inconsistency}\]
\[\text{Inconsistency}_B = \text{Inconsistency}_B + \text{Inconsistency}\]

Inconsistency_A: denotes transaction inconsistency existed in cell A.
Inconsistency_B: denotes transaction inconsistency existed in cell B.

Inconsistency: denotes transaction inconsistency the query transaction can "see".

In this example, the redistributed result is Implimit_A = 60 and Implimit_B = 140. In this way Implimit is redistributed from cell A to cell B, while in the meantime the mobile user continues his balance query. To the bank manager, the existence inconsistency seen is still 40. In this way, the long query transaction could be continuously processed while the mobile user moves from one cell to another cell, and Implimit will be redistributed between different sites to support mobility in mobile environments.

If the mobile host is still in cell A and has not moved to cell B when the estimated transaction inconsistency has been accumulated to Implimit_A, the demarcation protocol can still be used to let the sub-ETs negotiate between cell A and cell B to reconfigure Implimit_A and Implimit_B.

Figure 3.1 and Figure 3.2 illustrate the redistribution scheme and demarcation protocol used to negotiate the inconsistency limit in mobile environments.
3.5.3 Optimistic Divergence Control

Figure 3.1: Redistribution scheme when mobile host moves from cell A to B.

Figure 3.2: Demarcation protocol for negotiation inconsistency limit between cells.
Using an optimistic divergence control approach for continuous mobility support, a
distributed update ET is assigned a limit for export fuzziness and a distributed query
ET is assigned a limit for import fuzziness. The following procedure is performed:

- Each sub-ET executes the Optimistic Divergence Control in every cell locally.
  Unlike two-phase distributed divergence control, the local ODC uses ET’s entire
global $\epsilon$-spec, and optimistically runs to completion, assuming that the other sub-
ETs will not use their allowed fuzziness. If the ET’s $\epsilon$-spec is exceeded locally,
then it is marked for abort.

- At certification time, all sub-ET’s blocks are synchronized by the lock manager.
  If any sub-ET’s block is marked for abort, the ET must be aborted.

- If not, it first sums the local fuzziness introduced by all sub-ET’s blocks. If any
  local fuzziness exceeds its $\epsilon$-spec, the distributed ET is aborted.

- Otherwise, all weak locks held by transaction blocks of a distributed update ET
  are upgraded to strong locks, and then the distributed update ET can commit.

Note that optimistic distributed divergence control does not need to run a demarcation
protocol or redistribution scheme to negotiate for a greater inconsistency limit. Each
sub-ET is assigned a limit of the entire ET’s inconsistency limit.

### 3.5.4 Two-Phase Commit Protocol and Check-pointing Algorithm

The two-phase commit protocol [Bernstein, 1987; Blaybrook, 1992] and check-
pointing mobility algorithm [Acharya, 1994] are adopted in our proposed
concurrency control method to guarantee the global consistency between different
sites.

The two-phase commit protocol could still be suitably used for mobile transactions.
For the coordinator, the prepare phase and commit phase are the same as in the
classical environment. The only difference is the enhancement of the abort possibility
due to disconnection and cell migration. The details of the workings of the algorithm
are as follows:
• The coordinator sends a "prepare" to commit message to each subordinate host.

• The coordinator waits for a reply.

• If one or more nodes vote \textit{NO} or if no reply is received after a certain time bound, due to disconnection (maybe caused by the mobile host's voluntary or involuntary sleep), then the coordinator sends an abort message to all nodes involved in the transaction.

• If all votes are \textit{YES}, the coordinator sends a message instructing all other nodes to commit.

• The coordinator waits for an acknowledgment from all other nodes involved in the transaction. When all nodes have replied, this transaction is completed.

In mobile environments, timeouts occur more frequently so aborting becomes a more usual process than in classical environments, and therefore, check-pointing becomes a very important process. There are two key problems for check-pointing in mobile environments. One is due to catastrophic failures, e.g. loss, theft or physical damage. Disk storage on a MH is not acceptably stable for storing message logs or local checkpoints. Check-pointing schemes must therefore rely on a stable alternative repository for a MH's local checkpoints. The other problem is that disconnection of one or more of the MHs should not prevent recording the global state of an application executing on the MHs. In Acharya’s paper [Acharya, 1994], a check-pointing algorithm was designed to be used in mobile environments to complete the checkpoints in the distributed applications. This algorithm uses the MSS as the stable storage to maintain local checkpoints of MH and gives a method to make local checkpoints and form a consistent global checkpoint in mobile environments. This approach is adopted in our proposed concurrency control method to achieve global consistence when a transaction aborts.

In the bank balance query transaction, when the bank manager completes the query he requests to commit, the two-phase commit protocol is used to confirm that this transaction should be committed or aborted. If all sub-ETs in different cells agree to
commit then the transaction commits, otherwise the transaction aborts. The transaction aborts when estimated inconsistency exceeds the inconsistency limit, or timeouts happen when the two-phase commit protocol is used. When the transaction aborts the mobile host recovers back to the status before this transaction happens, that is, the mobile host recovers back to \textit{savepoint(0)}, which could be obtained by the mobility check-pointing algorithm described in Acharya’s paper [Acharya, 1994].

### 3.6 Implementing Concurrent Transactions in Mobile Environments

Our proposed concurrency control method above can process rapidly changing data objects in mobile environments with limited inconsistency and therefore improve transaction throughput. In this approach, the ESR divergence control algorithm is extended and modified to mobile environments for concurrency control.

The performance of ESR divergence control has been studied in [Wu, 1992] by using an analytical model similar to those used in [Yu, 1991a; Yu, 1990, Yu, 1991b]. Kamath et al. have refined the specification of inconsistency into a hierarchy, and have actually built a prototype system to do a more realistic appraisal of ESR [Kamath, 1994]. Their results show consistent and significant increases in TP system concurrency (and decreases in abort rate) when the epsilon bound is increased. This work strengthened the preliminary evaluation done on the original divergence control algorithm [Wu, 1992]. A group at Columbia University has evaluated the performance of centralized divergence control algorithms and divergence control algorithms using simulation [Pu, 1994]. The software is based on the CSIM package [Pu, 1993]. The performance of ESR has been studied by both simulation and prototype system in stationary environments, and can be evaluated in a similar way in mobile environments. The performance improvement in our proposed concurrency control method can be predicted since epsilon serializability is adopted as a relaxed serializability to process mobile applications. To evaluate performance in detail is one objective for an ongoing project in our ISL research group. The aim of this project is to demonstrate the feasibility and applicability of implementing concurrent
transactions in mobile environments and study the performance of our proposed concurrency control method in this chapter. The details on system architecture and experimental configuration for this project will be presented in the following sections.

### 3.6.1 System Architecture and Design

A system architecture is proposed in this section for implementing the concurrent transactions in mobile environments. This design merges the features of mobile environments [Krishnakumar, 1996] and the ESR system [Pu, 1994] into the transaction processing (TP) system described in Claybrook’s book [Claybrook, 1992]. It is a TP system architecture that can be used to study concurrency in mobile environments. The proposed system architecture is shown in Figure 3.3 as below:

![Figure 3.3: A system architecture for concurrency implementation in mobile environments.](image-url)
**Cache Manager:** In mobile environments, caching is used to reduce the communication in the network. Effective caching can reduce network traffic significantly. Database applications can benefit from caching in much the same way that small file applications can. The mobile host can only do simple work itself, and it can access data more quickly by using caching strategy. A two-level caching is adopted in mobile environments: the MSS caches a MH’s profile from the HS, while the MH caches the sub-profile from the MSS.

**Proxy Manager:** The proxy is created when a MH registers with a MSS. The MH merely communicates the request to the proxy that then becomes responsible for servicing it. The proxy services the request on the faster and more reliable hardwire links. This reduces the number of retries and timeouts that the MH would otherwise incur. The proxy manager is created on the MSS, and the proxy can provide three main services. The first is Remote Process Call (RPC) service. The proxy residing in the MSS acquires and releases locks, and retrieves data on behalf of the MH. The second service is the management of invalidation. The proxy knows what is in the MH cache; it buffers messages and invalidation until the MH is ready to receive them. Thirdly, the proxy manages data consistency by recording the timestamp when the MH contacts the proxy and then moves or sleeps. The data transfer over wireless link is slow and unreliable, and the transfer can be handled by a proxy to minimize the number of transfers.

**Lock Manager:** The HS maintains a scheduler that executes a locking scheme. The manager keeps track of all the locks given out and all the locks that are asked for. It regularly checks for deadlocks and informs the commit manager (CM) of their existence, and the identities (Ids) of the transactions involved. In mobile environments, transactions executed by the mobile hosts always take on a form consisting of several blocks. The blocks are separated by either voluntary or involuntary sleeps. It has been shown in Chapter 2 that two-phase locking still works in mobile environments. The lock manager in this system is designed to maintain two-phase locking in every block. During the growing phase, the transaction block obtains locks and during the shrinking phase the transaction block releases locks.
If the transaction commits before the MH becomes disconnected, the lock manager is no different with one in the classical wire network. If the MH becomes disconnected before the transaction commitment is completed, that is, the disconnection is voluntary or involuntary, then one of the following possibilities will occur: (1) In the case of voluntary disconnection, the MH will notice its disconnection and therefore the lock manager will unlock the granules locked by the transaction block of the MH; (2) In the case of involuntary disconnection, the MH will not notice its disconnection, and the lock manager has to be responsible for testing this situation. That is, if the MH is timed out before re-connection takes place, the lock manager will release the locks on its behalf and notice the MSS flushes the partial result from its cache.

To implement concurrent transactions in mobile environments using our concurrency control method proposed in this chapter, the lock manager should include an inconsistency calculator to calculate the inconsistency for the query transaction, and hold the ESR safeness pre-conditions that specify the inconsistency limits. The lock manager grants a write lock to an update transaction block and in the meantime grants a shared read lock to a query transaction block.

A lock manager may be implemented using six basic data structures [Claybrook, 1992]: transaction descriptor table, which is used to keep a list of the locks granted and/or requested by each transaction; lock table, which is used to keep a list of the lock-ids in use; lock element table, which is used to store the \textit{lm-tid} and the \textit{lock-mode} for each lock that has been granted or blocked; lock compatibility table, which is used to define the lock modes that are compatible with other lock modes; lock conversion table, which is used to define the new lock mode when two lock modes are to be converted to one; lock translation table, which is used to translate the valid lock modes into an internal representation for more efficient processing.

\textbf{Commit Manager:} The Commit Manager (CM) handles all commit, abort and recovery functions. The CM maintains three lists containing, respectively, the active, the committed and the aborted transactions. CM maintains a log in stable storage of redundant information such as the before- and the after-images of data granules.
updated by writes. Updates can be written either prior to the commit or as part of the commit. A transaction that aborts must have its updates undone by an undo algorithm. If there is a crash after a transaction is committed but before the data granule is written to storage, the CM will redo the update by copying the after-image of the granule from the log to the permanent database.

For the commit of a mobile transaction, the following steps are involved:

- The MH completes the updates and batches them in the cache to be transferred to the MSS.
- The MSS transfers the updates to the HS, which then writes them to storage and notifies other MHs of the commit and releases all the locks held.
- If the MH becomes disconnected before the transfer is completed, then one of the following three possibilities will occur: a) the disconnection is voluntary and the MH is not moving to another cell; b) the disconnection is voluntary and the MH is moving to another cell; c) the disconnection is involuntary. In the case of a), the MSS just waits. If the MH is timed out before re-connection takes place, the MSS will release the locks on its behalf and flush the partial result of the transfer from its cache. In the case of b), MSS will notify the new MSS to whose cell the MH is moving. It will also transfer the locks and flush the partial results. When the MH arrives at the new cell, it will resume transmission from where it left off. In the case of c), the MSS will broadcast the transaction Id to all the MSSs, informing them of the impending approach of a nomadic MH.

For transaction abort, local aborts (involving only one MSS) are simple. The CM just has to perform the undo actions and the HS (scheduler) releases all the locks. Cases with more than one MSS involved are also simple. A CM may require redo and / or undo logging depending on the particular recovery algorithm implemented. Four categories of algorithms can be designed [Claybrook, 1992]: undo but no redo; redo but no undo; undo and redo; no undo; and no redo.

### 3.6.2 Experimental Configuration
This section proposes an experimental configuration to implement the concurrent mobile transaction in a laboratory environment. It is proposed that the experiment should comprise two parts. In the first part, SUN workstations are adopted as the home server and the MSS. The aim of this part is to demonstrate the feasibility and applicability of implementing concurrent transactions in mobile environments. In the second part, it is proposed that Sun workstations are replaced by a gateway to an existent corporate system for more extensive applications in the network. The experiment configuration is shown in Figure 3.4 as below:

**Network Connection:** The network includes two parts: fixed network and wireless network. A SUN ultra10 workstation is used as the Home Server (HS); another two SUN ultra10 workstations are used as Mobile Support Stations (MSSs). They are connected with each other through a local ethernet, which is a fixed network. The oracle database is installed on the HS, and the SUN Solaris UNIX operating system is installed on the ultra 10 workstations for the HS and the MSSs.

The Telstra MobileData network [Telstra, 1999] is proposed to be used as the wireless network. Telstra’s MobileData is a packet-switched wireless data network for people on the move, which covers all capital cities and some major regional centres in Australia. The Telstra MobileData network is based on Motorola’s DataTAC 5000 system. It operates at 19.2 Kilobytes per second in the 800MHZ band.
The system is similar to other DataTAC networks in North America and Europe, and has become the standard in Australia and Asia, including Hong Kong, Singapore, Thailand, Malaysia, Indonesia, South Korea and China. Applications developed in Australia for the DataTAC 5000 system can be run without modification on other DataTAC 5000 networks. Data is segmented into packets of up to 512 bytes and then sent over a radio channel. This allows for extensive error checking and correction on each packet to ensure data is transferred across the network.

**Application Interface to the Mobiledata Network:** The application program interfaces (APIs) are available from the vendor Telstra.

**Wireless Terminal Options:** The wireless terminal consists of a radio packet modem (RPM) linked to a Toshiba Satellite laptop computer. The radio packet modem may need to be separately purchased depending on the model of laptop.

**Gateway as HS:** In the first part of implementation, the HS and the MSS are SUN Ultra 10 workstations in the laboratory. In a more general setting, as proposed in the second part of implementation, the HS may be a gateway to another network. The HS can connect through Radio Network Gateways (RNGs) located anywhere within the network. The Mobile Service Stations (MSSs) (known also as base stations in the case of wireless LANs) are connected to the HS through a Standard Context Router (SCR). The following Figure 3.5 is a simplified layout of the hardwire connections between the HS and the MSS. Information required by the MSS will be transmitted using the SCR protocol. The required physical connectivity can be provided by Telstra data communication services. The hub accepts X.25 packets, translates them to TCP/IP, and passes them on to the RNG. The Network Management Centre (NMC) collects traffic statistics and locates and determines the status of the Mobile users. The Radio Network Controller (RNC) performs data conversion to the Base Message Structure (BMS) protocol. It routes the BMS packets to the appropriate destination MSS and handles mobile user roaming among its MSSs. This is an interesting case since the connection directly links us up with a corporate establishment.
Figure 3.5: Fixed network: gateway as Home Server.

**System Coding:** As already mentioned above, the completed system coding is beyond the research scope of this thesis. To achieve a full implementation of our proposed system, another research project has been initiated towards this goal.

### 3.7 Summary

The chapter began by comparing different relaxed serializability criteria and methods of concurrency control in both conventional and mobile environments. Next, a relaxed serializability criterion - epsilon serializability is formally defined and extended to mobile environments. A relaxed concurrency control method for mobility support was then proposed and presented, so that rapidly changing data objects can be processed with limited inconsistency in mobile environments, and therefore yielding the improved transaction throughput. Compared with previous concurrency control approaches, five main advantages can be highlighted for our proposed methods: (1) It provides a concurrency control method applicable for mobile transactions, which is different from all other concurrency control algorithms that are only suitable for conventional environments [Bober, 1992; Breitbart, 1991; Elmagarmid, 1990; Georgakopoulos, 1991; Herlihy, 1990; Ho, 1997; Garcia-Molina, 1982]. (2) It adopts a mobile environment version of the epsilon serializability method, and thus it is high level and application independent, which differs from the work in [Joseph, 1999;
Katz, 1994; Davies, 1994; Kaashoek, 1994; Terry, 1995; Kistler, 1992; Kummar, 1994; Mummert, 1995; Reiher, 1994; Huston, 1995]. These works are limited to specific applications or problem domains. (3) It uses epsilon serializability as a weaker correctness criterion and tolerates the bounded inconsistency, and thus having improved transaction performance [Kamath, 1993; Wu, 1992; Pu 1993]. (4) It is feasible in that both two-phase locking and optimistic divergence control approaches can be employed for implementation. (5) It is particularly desirable for one class of application that involves a query for rapidly changing data, such as managers making queries in a stock market scenario, and rapid estimations of total account balances in a bank to do certain decision making.

The chapter ended with a presentation for a system architecture design and experimental configuration to demonstrate feasibility and applicability of implementing concurrent transactions in mobile environments.
Chapter 5

A Multi-Agent System Formalization in Mobile Environments

This chapter models and formalizes a mobile logic programming multi-agent system (MLPMAS) for mobile environments. Such a system consists of a number of agents connected via wire or wireless communication channels. Agents communicate with each other via passing answer sets obtained by updating the information received from connected agents with their own private information. The interactions between agents are also modeled in this formalization. Based on this model, knowledge based transaction can be studied in such a mobile multi-agent system. In addition, knowledge transaction is formally defined and modeled for these mobile multi-agent systems.

Section 5.1 presents and formalizes a mobile logic programming multi-agent system. Section 5.2 illustrates a study case to demonstrate how to specify a MLPMAS system in a particular problem domain. Section 5.3 formally defines and models the knowledge transaction in formalized MLPMAS systems. Then, three case studies are presented to address how to process a knowledge transaction in MLPMAS systems. Finally, section 5.4 summarizes the work in this chapter.

5.1 A Mobile Logic Programming Multi-Agent System Formalization

As discussed in Chapter 4, extended logic programming is a suitable tool for knowledge representation and study in mobile environments. In this chapter, extended logic programming is employed as a mathematical tool to formalize a mobile logic programming multi-agent system in mobile environments. The definition and formalization of MLPMAS systems are based on the three-layer
environment model proposed Chapter 4. This model is firstly recalled in Figure 5.1. In this model it is assumed that every Mobile Host (MH) has its own Knowledge Base (KB) and Intelligent Agent (A11, A12, A13, A21, A22, A23), and every MSS has a knowledge base and agent residing on it. It is also assumed that MSS1 and MSS2 represent different MSSs in different geographic areas. In the Home Server (HS) level, there is a knowledge base that possesses a set of rules, and there is an agent residing on it. Every intelligent agent on a MH will work on behalf of that MH, and all the agents in the same geographic area (i.e. controlled by the same HS) will negotiate, communicate, and cooperate with each other to achieve the goal for themselves and their systems.

A MLPMAS system is shown in Figure 5.2. A mobile logic programming multi-agent system includes the three levels, MH, MSS and HS, with the local knowledge base located on each level. The system consists of a set of agents in mobile environments. The agent resides on the MH, MSS and HS levels respectively, connected through communication channels. The agent on each level contains its own logic program representing its local information and reasoning method. Agents use information received from their incoming channels as input for their reasoning, where the received information may be overridden by other concerns represented in their programs. Agents produce output to their outgoing communication channels.
Definition 1: A mobile logic programming multi-agent system, or MLPMAS, is a pair $F = \langle A, C \rangle$, where $A$ is a set of agents: $A = A_{MH} \cup A_{MSS} \cup A_{HS}$, and $C \subseteq A \times A$ is a reflexive relation representing the communication channels between agents. For any $a_1, a_2 \in A$, if $\langle a_1, a_2 \rangle \in C$, then we say agents $a_1$ and $a_2$ have a communication channel. For each agent $a \in A$, there is an associated extended logic program $LocalKB(a)$, which represents agent $a$’s local knowledge base.

Example 1: Definition of the MLPMAS system above can be explained by Example 1, in which an investor agent resides on a MH, a group agent resides on a MSS, and a fund manager agent resides on a HS. The investor agent manages the local knowledge base and provides output to the group agent on behalf of the MH. The group agent collects information from all involved investor agents, manages the local knowledge base on the MSS and sends output to the fund manager agent. The fund manager agent collects information from all involved group agents, does the investment decision and manages the local knowledge base on the HS. The investor agent, group agent and fund manager agent are represented by $a_{MH}$, $a_{MSS}$ and $a_{HS}$ respectively.
Given a mobile logic programming multi-agent system $F = < A, C >$, with four mobile hosts MH1, MH2, MH3 and MH4, the investor agent resides on each MH:

$$A_{MH} = \{ a_{MH1}, a_{MH2}, a_{MH3}, a_{MH4} \}$$

There are two mobile support stations MSS1 and MSS2, with a group agent residing on each:

$$A_{MSS} = \{ a_{MSS1}, a_{MSS2} \}$$

There is one home server HS, with a resident fund manager agent:

$$A_{HS} = \{ a_{HS} \}$$

MH1 and MH2 are in the same geographic location as MSS1; MH3 and MH4 are in the same geographic location as MSS2. There is a wireless communication channel between the MHs and their MSS:

$$< a_{MH1}, a_{MSS1} > \in C , < a_{MH2}, a_{MSS1} > \in C ,$$

$$< a_{MH3}, a_{MSS2} > \in C , < a_{MH4}, a_{MSS2} > \in C$$

It is a wire communication channel between the MSS and the HS:

$$< a_{MSS1}, a_{HS} > \in C , < a_{MSS2}, a_{HS} > \in C$$

It is assumed there is a communication channel between the MH and the HS as well:

$$< a_{MH1}, a_{HS} > \in C , < a_{MH2}, a_{HS} > \in C ,$$

$$< a_{MH3}, a_{HS} > \in C , < a_{MH4}, a_{HS} > \in C$$

As mentioned earlier, each agent is associated with an extended logic program of its local knowledge base. If there is no answer set, it means the local knowledge base is not well designed. If there are multiple answer sets, each answer set represents a possible knowledge state.

The input and output of agents are defined in an MLPMAS as follows.

**Definition 2:** Let $F = < A, C >$ be a MLPMAS, where $A = A_{MH} \cup A_{MSS} \cup A_{HS}$. At the MH, MSS or HS level, for $\forall a \in A$, we have two parts to the input: message input and knowledge input, denoted by $MessageInput(a,X)$ and $KnowledgeInput(a,Y)$ respectively. That is,

$$Input(a) = < MessageInput(a,X), KnowledgeInput(a,Y) >$$
Here, \( X \subseteq A, Y \subseteq A \), \( X \) and \( Y \) are subsets of \( A \). Agent \( a \) collects message input from agents in \( X \), and collects knowledge input from agents in \( Y \), where

\[
\forall b \in X, \text{ we have } <a,b> \in C, \text{ or } <b,a> \in C \text{ and } \\
\forall b' \in Y, \text{ we have } <a,b'> \in C, \text{ or } <b',a> \in C.
\]

I.e. there is a communication channel between agent \( a \) and agent \( b \), and agent \( a \) and agent \( b' \) respectively.

Message input is the information that an agent sends to another agent for the purpose of communication, such as one agent informing another agent that it will move into another MSS geographic area. This information will not cause any influence to the other agent’s local knowledge base. However, knowledge input is the information produced by the other agent’s local knowledge base, and will be taken into the agent’s local knowledge base, i.e. the answer set of a logic program.

For \( \forall a \in A \), we have two parts to the output, message output and knowledge output, denoted by \( MessageOutput(a,X) \) and \( KnowledgeOutput(a,Y) \) respectively. That is

\[
Output(a) = <MessageOutput(a,X),KnowledgeOutput(a,Y)>
\]

here \( X \subseteq A, Y \subseteq A \). Agent \( a \) sends message output to agents in \( X \), and sends knowledge output to agents in \( Y \).

Message output is information output for communication purposes, this information will not cause any influence to the other agent’s local knowledge base. While knowledge output is the information that is produced by the agent’s local knowledge base and will have an impact for the other agent's knowledge base.

**Definition 3:** The knowledge input and output in MLPMAS systems are defined on the MH level as follows:

There is no input for MHs at the MH level because this is the first level in MLPMAS systems, i.e.

\[
KnowledgeInput(a_{MH}, Y) = \phi
\]
The knowledge output can be derived from the equation:

\[
\text{KnowledgeOutput}(a_{MH}, a_{MSS}) = \text{an answer set of } [\text{LocalKB}(a_{MH}) \cup \text{KnowledgeInput}(a_{MH}, Y)]
\]

I.e. knowledge output is an answer set of the program formed by the local logic program of agent \(a_{MH}\) with extension of knowledge input from \(Y\) for agent \(a_{MH}\).

\(\text{LocalKB}(a)\) is an extended logic program as defined in Definition 1. \(\text{KnowledgeInput}(a,Y)\) is a set of facts (beliefs). Note that \(\text{LocalKB}(a_{MH}) \cup \text{KnowledgeInput}(a_{MH}, Y)\) is viewed as a new logic program while fact \(e \in \text{KnowledgeInput}(a,Y)\) is treated as a rule \(e \leftarrow \).

**Definition 4:** The knowledge input and output in MLPMAS systems on the MSS level are defined as follows:

The knowledge input can be derived from the equation:

\[
\text{KnowledgeInput}(a_{MSS}, Y) = \text{cons} \left( \bigcup_{a_{MH} \in F} \text{KnowledgeOutput}(a_{MH}, a_{MSS}), S_F \right)
\]

where \(\text{cons}(X)\) returns a maximal consistent subset of knowledge output from \(Y\) to agent \(a_{MSS}\) with respect to the select function \(S_F\).

\(S_F\) is the *selection function* of the system. For knowledge output, \(\bigcup_{b \in F} \text{KnowledgeOutput}(b,a)\) may be inconsistent. \(S_F\) is introduced to solve such inconsistency by taking proper preference in the domain. Note that \(S_F\) is domain dependent. It can be a special logic programming rule for a specific problem domain.

The knowledge output can be derived from the equation:

\[
\text{KnowledgeOutput}(a_{MSS}, a_{HS}) = \text{an answer set of } [\text{LocalKB}(a_{MSS}) \cup \text{KnowledgeInput}(a_{MSS}, Y)]
\]

I.e. knowledge output is an answer set of the program formed by the local logic program of agent \(a_{MSS}\) with extending of knowledge input of agent \(a_{MSS}\).
**Definition 5:** Knowledge input and output in MLPMAS systems on the HS level is defined as follows:

The knowledge input can be derived from the equation:

\[
\text{KnowledgeInput}(a_{HS}, Y) = \text{cons}\left( \bigcup_{a_{MSS} \in Y} \text{KnowledgeOutput}(a_{MSS}, a_{HS}), S_F \right)
\]  

(5)

I.e. knowledge input of \( a_{HS} \) is the maximal consistent subset of knowledge output from \( Y \) to agent \( a_{HS} \) with respect to the select function \( S_F \).

The knowledge output can be derived from the equation:

\[
\text{KnowledgeOutput}(a_{HS}) = \text{an answer set of } [\text{LocalKB}(a_{HS}) \cup \text{KnowledgeInput}(a_{HS}, Y)]
\]

(6)

I.e. knowledge output is an answer set of the program formed by the local logic program of agent \( a_{HS} \) with extending of knowledge input of agent \( a_{HS} \).

**Example 2:** (Example 1 continued). Given the same scenario described in Example 1. To invest share1, MH1, MH2, MH3 and MH4 are involved investors. It is shown how input and output are derived on the MSS level based on their definition above.

On the MSS level, the group agent has input from involved investor agents on behalf of MH. Group agent \( a_{MSS1} \) on MSS1 has the input as below according to Definition 2:

\[
\text{Input}(a_{MSS1}) = \langle \text{MessageInput}(a_{MSS1}, X), \text{KnowledgeInput}(a_{MSS1}, Y) \rangle
\]

Here \( X = \{a_{MH1}, a_{MH2}\} \) is a subset of \( A \), including all investor agents who have message input for the group agent on MSS1. \( Y = \{a_{MH1}, a_{MH2}\} \) is a subset of \( A \), including all investor agents who have knowledge input for the agent on MSS1.

According to equation (3), knowledge input on MSS1 can be derived as below:

\[
\text{KnowledgeInput}(a_{MSS1}, Y) = \text{cons}\left( \bigcup_{a_{MH} \in Y} \text{KnowledgeOutput}(a_{MH}, a_{MSS1}), S_F \right)
\]

\[
= \text{cons}(\text{KnowledgeOutput}(a_{MH1}, a_{MSS1}) \cup \text{KnowledgeOutput}(a_{MH2}, a_{MSS1}), S_F)
\]
The group agent on MSS1 has message output and knowledge output according to Definition 2:

\[ \text{Output}(a_{\text{MSS1}}) = (\text{MessageOutput}(a_{\text{MSS1}}, X), \text{KnowledgeOutput}(a_{\text{MSS1}}, Y)) \]

Here \( X = \{a_{\text{HS}}\} \), \( Y = \{a_{\text{HS}}\} \) are subsets of A. The agent \( a_{\text{MSS1}} \) sends message output to the agent \( a_{\text{HS}} \) in \( X \) and sends knowledge output to the agent \( a_{\text{HS}} \) in \( Y \).

The knowledge output can be derived from equation (4):

\[ \text{KnowledgeOutput}(a_{\text{MSS1}}, a_{\text{HS}}) = \text{an answer set of } [\text{LocalKB}(a_{\text{MSS1}}) \cup \text{KnowledgeInput}(a_{\text{MSS1}}, Y)] \]

Thus, an agent sends its full set of belief outputs over all outgoing communication channels. On the other hand, an agent receives as input, the beliefs of all agents connected to its incoming channels.

### 5.2 A Case Study for MLPMAS Systems

Study case 1 is presented to illustrate how to specify a MLPMAS system in a specific problem domain based on the formalization above.

**Study Case 1**: This case study is presented in a specific investment problem domain. The MLPMAS system has shown in Figure 5.2. At the MH level, there are MH1, MH2, MH3 and MH4. MH1 and MH2 are in the cell of MSS1, MH3 and MH4 are in the cell of MSS2. MSS1 and MSS2 are connected to the same HS. At the MH level, each MH has a local knowledge base that includes a set of investment rules, and has an investor agent residing on it. At the MSS level, a MSS has its own knowledge base, and accepts the input from MHs and produces the output based on this input and its own belief. The HS has its own local knowledge base. It accepts input from the MSS level and makes the investment decision. For the initial status, it is assumed that MH1 and MH2 are all alive when a transaction is processed in a MSS1 cell. In a MSS2 cell, the MH3 is alive, while MH4 is in sleep at the moment the HS is requesting the transaction information from all related MH agents. The HS will need information from MH4 by the time it does the decision making. The logic programming rules for local knowledge bases on MH, MSS, and HS is specified.
when input and output of agents are discussed on every level. Note that, all the
equations and definitions quoted in this case study are those formalized in MLPMAS
systems in section 5.1 of this chapter.

MH Level:

Local knowledge base at MH level:
The logic programming rules of local knowledge base on MH level are as follow.
The local knowledge base of MH1 has rules \( r1-r3 \) related to the share1 investment:
\[
\begin{align*}
r1: & \text{ holds(profit(share1))} \\
r2: & \text{ holds(risk(share1))} \\
r3: & \text{ holds(cost(share1))} \\
\end{align*}
\]
The local knowledge base of MH2 has rules \( r4-r6 \) related to the share1 investment:
\[
\begin{align*}
r4: & \text{ holds(profit(share1))} \\
r5: & \text{ holds(risk(share1))} \\
r6: & \text{ holds(cost(share1))} \\
\end{align*}
\]
The local knowledge base of MH3 has rules \( r7-r9 \) related to the share1 investment:
\[
\begin{align*}
r7: & \text{ holds(profit(share1))} \\
r8: & \text{ holds(risk(share1))} \\
r9: & \text{ holds(cost(share1))} \\
\end{align*}
\]
The local knowledge base of MH4 has rules \( r10-r12 \) related to the share1 investment:
\[
\begin{align*}
r10: & \text{ holds(profit(share1))} \\
r11: & \text{ holds(risk(share1))} \\
r12: & \text{ holds(cost(share1))} \\
\end{align*}
\]

Input at MH level:
According to equation (2), there is no input for MHs at the MH level because this is
the first level in MLPMAS systems. We have
\[
\text{KnowledgeInput}(a_{MH}, Y) = \phi
\]
i.e.
\[
\text{KnowledgeInput}(a_{MH1}) = \phi, \quad \text{KnowledgeInput}(a_{MH2}) = \phi, \\
\text{KnowledgeInput}(a_{MH3}) = \phi, \quad \text{KnowledgeInput}(a_{MH4}) = \phi
\]
Output at MH level:

According to equation (2), we have

\[ KnowledgeOutput(a_{MH1}, a_{MSS}) = \text{an answer set of } \{ LocalKB(a_{MH1}) \cup KnowledgeInput(a_{MH1}, Y) \} \]

\[ = \text{an answer set of } \{ LocalKB(a_{MH1}) \cup \phi \} \]

\[ = \text{an answer set of } \{ LocalKB(a_{MH1}) \} \]

I.e. at the MH level, the knowledge output is an answer set of local knowledge base.

Based on rules \( r1 \)-\( r9 \) of the local knowledge base of MHs, the knowledge outputs are derived as below on MH1, MH2, MH3 and MH4.

\[ KnowledgeOutput(a_{MH1}, a_{MSS}) = \{ \text{profit}(\text{share}1), \text{risk}(\text{share}1), \neg \text{cost}(\text{share}1) \} \]

i.e. it is high profit, high risk and low cost to invest share1 on MH1.

\[ KnowledgeOutput(a_{MH2}, a_{MSS}) = \{ \text{profit}(\text{share}1), \neg \text{risk}(\text{share}1), \neg \text{cost}(\text{share}1) \} \]

i.e. it is high profit, low risk and low cost to invest share1 on MH2.

\[ KnowledgeOutput(a_{MH3}, a_{MSS}) = \{ \text{profit}(\text{share}1), \neg \text{risk}(\text{share}1), \neg \text{cost}(\text{share}1) \} \]

i.e. it is high profit, low risk and low cost to invest share1 on MH3. MH4 is at sleep at the moment the information is retrieved from it.

MSS level:

Local knowledge base at MSS level:

On MSS1, we have rule \( r13 \) related to this investment in its knowledge base:

\[ \{ r13: \text{holds}(\text{info} \rightarrow \text{requested}(\text{HS, MHi})) \leftarrow \text{holds}(\text{slept}(\text{MHi})) \} \]

On MSS2, we have rule \( r14 \) similar to \( r13 \) related to this investment in its knowledge base:

\[ \{ r14: \text{holds}(\text{info} \rightarrow \text{requested}(\text{HS, MHi})) \leftarrow \text{holds}(\text{slept}(\text{MHi})) \} \]

Rules \( r13 \) and \( r14 \) denote that if MHi is in sleep at the time the HS agent requests transaction information from MHs, HS will request information from MHi when HS does the decision making for the transaction.

Input at MSS level:

At MSS level, according to Definition 2 of MLPMAS systems, input of MSS1 agent equals:
Chapter 5. A Multi-Agent System Formalization in Mobile Environments

\[ Input(a_{MSS_1}) = \langle MessageInput(a_{MSS_1}, X), KnowledgeInput(a_{MSS_1}, Y) \rangle \]

where \( X = \{a_{MH_1}, a_{MH_2}\} \) is a subset of \( A \), including all investor agents who have message input for the group agent on MSS1. \( Y = \{a_{MH_1}, a_{MH_2}\} \) is a subset of \( A \), including all investor agents who have knowledge input for the group agent on MSS1.

According to equation (3), knowledge input on MSS1 is as below:

\[
KnowledgeInput(a_{MSS_1}, Y) = \text{cons}\left( \bigcup_{a_{MH} \in Y} KnowledgeOutput(a_{MH}, a_{MSS_1}, S_F) \right)
\]

\[
= \text{cons}( KnowledgeOutput(a_{MH_1}, a_{MSS_1}) \cup KnowledgeOutput(a_{MH_2}, a_{MSS_1}, S_F) )
\]

\[
= \text{cons}( \{ \text{profit}(\text{share}i), \text{risk}(\text{share}i), \neg \text{cost}(\text{share}i) \} \cup \{ \text{profit}(\text{share}i), \neg \text{risk}(\text{share}i), \neg \text{cost}(\text{share}i) \}, S_F )
\]

For agent \( a_{MSS_1} \), \( \text{risk}(\text{share}i) \) is a belief in output of \( a_{MH_1} \), while \( \neg \text{risk}(\text{share}i) \) is a belief in output of \( a_{MH_2} \), they are inconsistent. Here it is assumed that the selection function \( S_F \) takes the positive atom as the higher preference for investment risk, in which case, \( \text{risk}(\text{share}i) \) will become the input of \( a_{MSS_1} \).

Therefore, the knowledge input of \( a_{MSS_1} \) is:

\[ KnowledgeInput(a_{MSS_1}, Y) = \{ \text{profit}(\text{share}i), \text{risk}(\text{share}i), \neg \text{cost}(\text{share}i) \} \]

Without considering the selection function the knowledge input of \( a_{MSS_1} \) will be different:

\[ KnowledgeInput(a_{MSS_1}, Y) = \{ \text{profit}(\text{share}i), \neg \text{cost}(\text{share}i) \} \]

The different knowledge input is derived by considering the selection function in a specific problem domain, therefore a different answer set is derived for decision making due to the selection function.

Input to MSS2 agent equals:

\[ Input(a_{MSS_2}) = \langle MessageInput(a_{MSS_2}, X), KnowledgeInput(a_{MSS_2}, Y) \rangle \]

where \( X = \{a_{MH_3}, a_{MH_4}\} \) is a subset of \( A \), including all investor agents who have message input for the group agent on MSS2. \( Y = \{a_{MH_3}, a_{MH_4}\} \) is a subset of \( A \),
including all investor agents who have knowledge input for the group agent on MSS2.

The MH4 is at sleep at the moment, so knowledge input of MSS2 agent equals:

\[
\text{KnowledgeInput}(a_{MSS2}, Y) = \text{cons}(\bigcup_{a_{MSS2}} \text{KnowledgeOutput}(a_{MH}, a_{MSS2}, S_F))
\]

\[
= \text{cons}(\text{KnowledgeOutput}(a_{MH3}, a_{MSS2}) \cup \text{KnowledgeOutput}(a_{MH4}, a_{MSS2}, S_F))
\]

\[
= \text{cons}\{\text{profit}(share1), -\text{risk}(share1), -\text{cost}(share1)\} \cup \emptyset, S_F
\]

\[
= \{\text{profit}(share1), -\text{risk}(share1), -\text{cost}(share1)\}
\]

Output at MSS level:

Based on Definition 2 of a MLPMAS system, there are two parts to the output of the MSS, message output and knowledge output.

According to equation (4), the knowledge output can be derived:

\[
\text{KnowledgeOutput}(a_{MSS}, a_{HS}) = \text{an answer set of } [\text{LocalKB}(a_{MSS}) \cup \text{KnowledgeInput}(a_{MSS}, Y)]
\]

The knowledge output of MSS1 is derived as below:

\[
\text{KnowledgeOutput}(a_{MSS1}, a_{HS}) = \{\text{profit}(share1), \text{risk}(share1), -\text{cost}(share1)\}
\]

The knowledge output of MSS2 is derived as below:

\[
\text{KnowledgeOutput}(a_{MSS2}, a_{HS}) = \{\text{profit}(share1), -\text{risk}(share1), -\text{cost}(share1), \text{info-requested}(HS, MH4)\}
\]

A new belief, \text{info-requested}(HS, MH4), is added to the answer set on MSS2 because of rule \(r14\) in the local knowledge base of MSS2.

HS level:

Local knowledge base at HS level:

There are rules \(r15-r21\) in the local knowledge base of HS.
Chapter 5. A Multi-Agent System Formalization in Mobile Environments

\[
\begin{align*}
\text{r15: } & \text{holds} (\text{invest}(\text{share1})) \leftarrow \text{holds} (\text{profit}(\text{share1})), \neg \text{holds} (\text{risk}(\text{share1})), \\
& \neg \text{holds} (\text{cost}(\text{share1})), \text{holds} (\text{info} \rightarrow \text{get} (\text{MHi}), \text{res} (\text{request} \rightarrow \text{info} (\text{MHi}))) \\
\text{r16: } & \neg \text{holds} (\text{invest}(\text{share1})) \leftarrow \text{holds} (\text{risk}(\text{share1})) \\
\text{r17: } & \neg \text{holds} (\text{invest}(\text{share1})) \leftarrow \text{holds} (\text{cost}(\text{share1})) \\
\text{r18: } & \neg \text{holds} (\text{risk}(\text{share1})) \leftarrow \text{notholds} (\text{risk}(\text{share1})) \\
\text{r19: } & \neg \text{holds} (\text{cost}(\text{share1})) \leftarrow \text{notholds} (\text{cost}(\text{share1})) \\
\text{r20: } & \neg \text{holds} (\text{invest}(\text{share1})) \leftarrow \text{holds} (\text{info} \rightarrow \text{requested} (\text{HS}, \text{MHi})), \\
& \neg \text{holds} (\text{info} \rightarrow \text{get} (\text{MHi}), \text{res} (\text{request} \rightarrow \text{info} (\text{MHi}))), \\
\text{r21: } & \neg \text{holds} (\text{info} \rightarrow \text{get} (\text{MHi})) \leftarrow \text{notholds} (\text{info} \rightarrow \text{get} (\text{MHi})), \\
& \text{holds} (\text{info} \rightarrow \text{requested} (\text{MHi})), \text{holds} (\text{timeout} (\text{MHi}))
\end{align*}
\]

Rule r15 denotes that if it is high profit, low risk, low cost to invest share1, and the HS gets requested information from every sleeping MHi, the decision to invest share1 will be made. Rules r16, r17 and r20 denote that if share1 is high risk or high cost on any MHi, or information is not available from every sleeping MHi, then the HS will make the decision that share1 will not be invested. Rules r18 and r19 denote that if share1 hasn’t been specified to be high risk or high cost for any MHi, then it is considered to be low risk or low cost. Rule r21 denotes that if the HS hasn’t got requested information from sleeping MHi before time is out, then the HS will assume no information is available from MHi.

**Input at HS level:**

On the HS level, based on Definition 2 of a MLPMAS system, input of an HS agent equals:

\[
\text{Input}(a_{HS}) = \langle \text{MessageInput}(a_{HS}, X), \text{KnowledgeInput}(a_{HS}, Y) \rangle
\]

Where \(X = \{a_{MSS1}, a_{MSS2}\}\) is a subset of \(A\), including all involved agents who have message information input for the agent on the HS. \(Y = \{a_{MSS1}, a_{MSS2}\}\) is a subset of \(A\), including all involved agents who have knowledge input for the agent on the HS.

Based on equation (5), knowledge input of a HS agent equals:
KnowledgeInput($a_{HS}, Y$) 

$$= \text{cons}(\bigcup_{a_{MSS} \in Y} \text{KnowledgeOutput}(a_{MSS}, a_{HS}, S_F))$$

$$= \text{cons}\left(\text{KnowledgeOutput}(a_{MSS1}, a_{HS}) \cup \text{KnowledgeOutput}(a_{MSS2}, a_{HS}, S_F)\right)$$

$$= \text{cons}\left(\{\text{profit}(share1), \text{risk}(share1), \neg \text{cost}(share1)\}\cup\right.$$

$$\left.\{\text{profit}(share1), \neg \text{risk}(share1), \neg \text{cost}(share1), \text{info} - \text{requested}(HS, MH4)\}, S_F\right)$$

$$= \{\text{profit}(share1), \text{risk}(share1), \neg \text{cost}(share1), \text{info} - \text{requested}(HS, MH4)\}$$

$\text{risk}(share1)$ is a belief of input on the HS with consideration of the selection function.

Output at HS level:

Based on Definition 2 of a MLPMAS system, the output of a HS agent has two parts, message output and knowledge output. The knowledge output is derived as below according to equation (6):

\[
\text{KnowledgeOutput}(a_{HS}) = \text{an answer set of } [\text{LocalKB}(a_{HS}) \cup \text{KnowledgeInput}(a_{HS}, Y)]
\]

\[
= \text{an answer set of } [\text{LocalKB}(a_{HS}) \cup \{\text{profit}(share1), \text{risk}(share1), \neg \text{cost}(share1), \text{info} - \text{requested}(HS, MH4)\}]\]

\[
= \{\neg \text{invest}(share1), \text{profit}(share1), \text{risk}(share1), \neg \text{cost}(share1)\}
\]

Where $[\text{LocalKB}(a_{HS}) \cup \text{KnowledgeInput}(a_{HS}, Y)] \models \{\neg \text{invest}(share1)\}$.

The $\text{risk}(share1)$ is a part of the knowledge input of the HS, and according to rule r10 of the knowledge base on the HS, it is known that share1 can not be invested if there is a high risk to invest it. From the output of HS, the fund manager agent on HS makes the decision that share1 won’t be invested. In this case, no matter what information is received from MH4, HS will make the decision that share1 can’t be invested, otherwise the information from MH4 will be considered as input to HS, and the decision making will be made based on all inputs and the local knowledge base itself. It is also shown that the selection function will impact the decision making. If no selection function is used in this example, $\text{risk}(share1)$ will not be part of the input to the HS agent, therefore share1 may be invested.
After the HS has made the decision that share1 will not be invested, the transaction decision will be sent to the MSS, and all involved MHs will be notified by broadcasts from the MSS.

5.3 Transaction Processing in MLPMAS Systems

5.3.1 Defining Knowledge Transaction in MLPMAS Systems

Based on Definition 1 of a MLPMAS system presented above, a Mobile Logic Programming Multi-Agent System (MLPMAS) is a pair $F = <A,C>$, where $A$ is a set of agents: $A = A_{MH} \cup A_{MSS} \cup A_{HS}$, and $C \subseteq A \times A$ is a reflexive relation representing the communication channels between agents. Recall the formalization in chapter 4: the knowledge transaction model on domain $D$ is defined as a pair $\Sigma = (R, I(D))$ in mobile environments, where $R$ is a set of generic knowledge transaction rules, and $I(D)$ is a finite set of initial facts and rules with respect to the problem domain $D$. Thus knowledge transaction can be defined in mobile logic programming multi-agent systems as follows:

**Definition 6:** In mobile logic programming multi-agent systems, the knowledge transaction can be defined as $T = <A, C, R, I(D)>$, where $A$ is a set of agents: $A = A_{MH} \cup A_{MSS} \cup A_{HS}$, $C \subseteq A \times A$ is a reflexive relation representing the communication channels between agents, $R$ is a set of generic knowledge transaction processing rules, and $I(D)$ is a finite set of initial facts and rules with respect to problem domain $D$.

5.3.2 Transaction Processing Language in Mobile Environments

In chapter 4, a language $\mathcal{L}$ is formalized to process knowledge based transaction in mobile environments. Firstly recall the syntax of language $\mathcal{L}$ as follows. A language $\mathcal{L}$ contains variables of three sorts: situation variable $s$, fluent variable $f$, and action variable $a$. It has one predicate $\text{holds}(f, s)$, where $f$ is a fluent function, and $s$ is a
situation. The predicate \( \text{holds}(f, s) \) means that fluent \( f \) is true at situation \( s \). \( \mathcal{L} \) also has a resulting function \( \text{res}(a, s) \), which denotes a situation resulting from situation \( s \) by performing action \( a \). To begin with, \( \mathcal{L} \) has one initial situation constant \( S_0 \).

Based on language \( \mathcal{L} \), the following action functions are defined in MLPMAS systems:

- **Write** (MHi): denotes MHi has an update transaction request.
- **Submit** (MSSi): denotes MSSi submits the transaction request to HS.
- **acquire-lockc** (MHi): denotes MHi acquires lock for transaction updating.
- **do-trans** (HS): denotes HS starts the transaction.
- **request-info** (MHi): denotes MHi is requested to provide its information.
- **cal-output** (MHi): denotes MHi calculates its output.
- **sent-info** (MHi): denotes MHi sends its information as output.
- **cal-input** (MSSi): denotes MSSi calculates its input.
- **cal-output** (MSSi): denotes MSSi calculates its output.
- **cal-input** (HS): denotes HS calculates its input.
- **cal-output** (HS): denotes HS calculates its output.
- **make-decision** (HS): denotes HS makes the decision for the transaction.
- **abort-trans** (HS): denotes HS aborts the transaction.
- **notice-abort** (HS, MSSi): denotes HS notices MSSi transaction abort.
- **broadcast** (MSSi, MHi): denotes MSSi broadcasts transaction status to MHi.
- **update-knowledge** (MHi): denotes MHi updates its knowledge base.

The following fluent functions are defined in MLPMAS systems:

- **update-requested** (MHi): denotes MHi has an update transaction request.
- **registered** (MSSi, MHi): denotes MHi has registered in MSSi cell.
- **trans-submitted** (MSSi): denotes MSSi has submitted the transaction request to HS.
- **locked** (MHi): denotes MHi has got the lock for the transaction.
- **trans-start** (HS): denotes transaction start on HS.
- **info-requested** (MHi): denotes MHi has been requested to provide its information.
- **output-derived** (MHi): denotes MHi has derived its output.
- **output-sent** (MHi): denotes MHi has sent its output to MSSi.
**input-derived(MSSi):** denotes MSSi has derived its input.

**output-derived(MSSi):** denotes MSSi has derived its output.

**input-derived(HS):** denotes HS has derived its input.

**output-derived(HS):** denotes HS has derived its output.

**decision-made(HS):** denotes HS has made the transaction decision.

**trans-aborted(HS):** denotes HS has aborted the transaction.

**knowledge-update(HS):** denotes HS has updated its knowledge base.

**abort-noticed(HS, MSSi):** denotes HS has sent the transaction abort notice to MSSi.

**abort-broadcasting(MSSi, MHi):** denotes MSSi has broadcast the transaction abort to MHi.

**knowledge-update(MHi):** denotes MHi has updated its knowledge base.

### 5.3.3 Transaction Processing Rules in MLPMAS Systems

In chapter 4, a set of rules are specified and imposed to formalize a knowledge transaction processing model in mobile environments, which models all transaction processing activities, requests, results and constraints on MH, MSS, and HS levels. Firstly recall some of those transaction rules, which will be used in the study cases in later sections of this chapter.

The following rules are imposed in chapter 4 to specify knowledge transaction processing in mobile environments:

At the register stage of a transaction, when MH moves into MSS cell, it is registered. The rule for this is:

\[
\text{t1: holds(registered(MSS, MH), res(move(MSS, MH), s))} \leftarrow
\]

After an action \(\text{move(MSS, MH)}\) happens, \(\text{res(move(MSS, MH), s)}\) becomes the current situation, and \(\text{registered(MSS, MH)}\) is true.

Then MH requests to start an update transaction. The rule is:

\[
\text{t2: holds(update-requested(MH), res(write(MH), s))} \leftarrow
\]

After an action \(\text{write(MH)}\), i.e., the MH submits a write request, \(\text{res(write(MH), s)}\) becomes the current situation, and \(\text{update-requested(MH)}\) becomes true.
After the MH requests a query or update, the MSS submits this transaction request to the HS on behalf of the MH. If it is a write request, the lock needs to be acquired firstly to submit this transaction.

\[ t_3: \text{holds} (\text{trans-submitted(MSS)}, \text{res(submit(MSS), s)}) \leftarrow \text{holds(locked(MH), s), holds(update-requested(MH), s)} \]

If both update-required(MH) and locked(MH) are true, action submit(MSS) will happen and trans-submitted(MSS) then becomes true.

After the MH requests a query or update transaction and the MSS submits this transaction request to the HS, the HS starts the transaction. The rule is:

\[ t_4: \text{holds} (\text{trans-start(HS), res(do-trans(HS), s)}) \leftarrow \text{holds(trans-submitted(MSS), s)} \]

After transaction request is submitted by the MSS, i.e., trans-submitted(MSS) is true, action do-trans(HS) happens, and trans-start(HS) then becomes true, the transaction starts.

If the decision is made that the transaction will be aborted at the HS level, HS updates its knowledge base to reflect this accordingly. The rules to denote this are \( t_5 \) and \( t_6 \):

\[ t_5: \text{holds} (\text{trans-aborted(HS), res(abort-trans (HS),s)}) \leftarrow \text{holds(decesion-made(HS),s)} \]

\[ t_6: \neg \text{holds(knowledge-update(HS),s)} \leftarrow \text{holds(trans-aborted(HS),s)} \]

The HS sends transaction abort notice to the MSS. The rule is:

\[ t_7: \text{holds} (\text{abort–noticed (HS,MSS)}, \text{res(notice–abort(HS,MSS),s)}) \leftarrow \text{holds(trans–aborted(HS),s)} \]

The MSS broadcasts the transaction abort information to involved MHS. The rule is:

\[ t_8: \text{holds} (\text{abort–broadcasting(MSS,MH)}, \text{res(broadcast(MSS,MH),s)}) \leftarrow \text{holds(abort–noticed(HS,MSS),s)} \]

The MHS update their local knowledge base accordingly, the rule reflecting this is:

\[ t_9: \neg \text{holds(knowledge–update(MH),res(update–knowldg(MH),s))} \leftarrow \text{holds(abort–broadcasting(MSS,MH),s)} \]
A set of new rules are imposed in this chapter to specify the activities and capture the features of MLPMAS systems in mobile environments. The rules are specified as follows. Note that the equations mentioned below are those presented in section 5.1 for the formalization of a MLPMAS system.

After the MH agent is requested by the HS to provide problem domain related information to the HS, the MH agent calculates the MH’s output based on equation (2), then sends its output to the MSS if the MH is not at sleep. The rules specifying this are:

$$t_{10}: \text{holds}(\text{output-derivative}(MHi), \text{res}(\text{cal-output}(MHi), s)) \leftarrow \text{holds}(\text{info-requested}(MHi), s)$$

$$t_{11}: \text{holds}(\text{output-sent}(MHi), \text{res}(\text{sent-output}(MHi), s_0)) \leftarrow \text{holds}(\text{output-sent}(MHi), s_0)$$

Rules $t_{10}$ and $t_{11}$ denote: If the MH agent is requested to provide its information, action $\text{cal-output}(MHi)$ is taken to calculate the output, $\text{output-derivative}(MHi)$ becomes true. Then output is sent to the MSS by action $\text{sent-output}(MHi)$, i.e., $\text{output-sent}(MHi)$ becomes true.

The MSS agent collects information from all related MHs and calculates the input based on the equation (3). The rule is:

$$t_{12}: \text{holds}(\text{input-derivative}(MSSi), \text{res}(\text{cal-input}(MSSi), s)) \leftarrow \text{holds}(\text{output-sent}(MHi), s)$$

Rule $t_{12}$ denotes: After output of the MH is sent to the MSS, the MSS agent takes action $\text{cal-input}(MSSi)$. This derives the input of the MSS, i.e., $\text{input-derivative}(MSSi)$ is true.

Then the output of the MSS is calculated and derived based on equation (4) after its input is derived. The rule is:

$$t_{13}: \text{holds}(\text{output-derivative}(MSSi), \text{res}(\text{cal-output}(MSSi), s)) \leftarrow \text{holds}(\text{input-derivative}(MSSi), s)$$

Rule $t_{13}$ denotes: After input of the MSS is derived, the MSS agent takes action $\text{cal-output}(MSSi)$, then its output is derived, i.e., $\text{output-derivative}(MSSi)$ becomes true.
The HS agent gets the output of the MSS, and calculates the input of the HS based on equation (5). The rule is:

\[ t14: \text{holds(input-derived(HS),res(cal-input(HS),s))} \leftarrow \text{holds(output-derived(MSSI),s)} \]

Rule \( t14 \) denotes: After the output of the MSS is known to the HS, the HS agent takes action \( \text{cal-input(HS)} \), then the input of the HS has been derived, i.e., \( \text{input-derived(HS)} \) is true.

The output of the HS will be derived after the calculation base on equation (6) by considering the input of the HS and its local knowledge base. The rules are:

\[ t15: \text{holds(output-derived(HS),res(cal-output(HS),s))} \leftarrow \text{holds(input-derived(HS),s)} \]

Rule \( t15 \) denotes: After input of the HS is derived, the HS agent takes action \( \text{cal-output(HS)} \), then its output is derived, i.e., \( \text{output-derived(HS)} \) becomes true.

The HS will do the decision-making based on its output. The rule is:

\[ t16: \text{holds(decision-made(HS),res(make-decision(HS),s))} \leftarrow \text{holds(output-derived(HS),s)} \]

Rule \( t16 \) denotes: After \( \text{output-derived(HS)} \) becomes true, the HS will take action \( \text{make-decision(HS)} \), then the decision is made, i.e. \( \text{decision-made(HS)} \) becomes true.

### 5.3.4 Transaction Study Case in MLPMAS systems

Study case 2 demonstrates how a knowledge transaction is processed in a specific problem domain in MLPMAS systems.

**Study Case 2:** Given the same scenario and investment domain as study case 1 in this chapter. The initial status is the same: MH1, MH2 and MH3 is alive, while MH4 is at sleep. At the time the HS agent requests a transaction, MH1 initializes an update transaction by requesting to invest a share (\textit{share1}). Here, we will present, step by step, the activities involved in this transaction together with the generic transaction rules and the specific rules that are applied in this investment domain.
As the initial facts, MH1 and MH2 are registered with MSS1, while MH3 and MH4 are registered with MSS2. Based on rule \( t1 \) specified in section 5.3.3, the rules \( i1-i4 \) below denote the initial status:

\[
i1: \text{holds}(\text{registered}(\text{MSS1}, \text{MH1}), s_0) \leftarrow
i2: \text{holds}(\text{registered}(\text{MSS1}, \text{MH2}), s_0) \leftarrow
i3: \text{holds}(\text{registered}(\text{MSS2}, \text{MH3}), s_0) \leftarrow
i4: \text{holds}(\text{registered}(\text{MSS2}, \text{MH4}), s_0) \leftarrow
\]

The rules for the local knowledge base in the MH, MSS and HS are domain related rules. Recall these rules as below.

Local knowledge base of MH:

The local Knowledge base of MH1 has rules \( r1-r3 \) related to share1 investment:

\[
\begin{align*}
\{ r1: \text{holds}(\text{profit}(\text{share1}), s_0) \leftarrow \\
r2: \text{holds}(\text{risk}(\text{share1}), s_0) \leftarrow \\
r3: \lnot \text{holds}(\text{cost}(\text{share1}), s_0) \leftarrow 
\end{align*}
\]

The local Knowledge base of MH2 has rules \( r4-r6 \) related to share1 investment:

\[
\begin{align*}
\{ r4: \text{holds}(\text{profit}(\text{share1}), s_0) \leftarrow \\
r5: \lnot \text{holds}(\text{risk}(\text{share1}), s_0) \leftarrow \\
r6: \lnot \text{holds}(\text{cost}(\text{share1}), s_0) \leftarrow 
\end{align*}
\]

The local Knowledge base of MH3 has rules \( r7-r9 \) related to share1 investment:

\[
\begin{align*}
\{ r7: \text{holds}(\text{profit}(\text{share1}), s_0) \leftarrow \\
r8: \lnot \text{holds}(\text{risk}(\text{share1}), s_0) \leftarrow \\
r9: \lnot \text{holds}(\text{cost}(\text{share1}), s_0) \leftarrow 
\end{align*}
\]

The local Knowledge base of MH4 has rules \( r10-r12 \) related to share1 investment:

\[
\begin{align*}
\{ r10: \text{holds}(\text{profit}(\text{share1}), s_0) \leftarrow \\
r11: \lnot \text{holds}(\text{risk}(\text{share1}), s_0) \leftarrow \\
r12: \lnot \text{holds}(\text{cost}(\text{share1}), s_0) \leftarrow 
\end{align*}
\]

The local knowledge base of the MSS:

On MSS1, rule \( r13 \) is related to this investment in its knowledge base:

\[
\{ r13: \text{holds}(\text{info-requested}(\text{HS}, \text{MHi}), s_0) \leftarrow \text{holds}(\text{slept}(\text{MHi}), s_0) \}\]
On MSS2, rule \( r14 \) is related to this investment in its knowledge base:

\[
\begin{align*}
\{ r14: & \text{ holds(info-requested(HS, MHi), } s_0) \leftrightarrow \text{ holds(slept(MHi), } s_0) \} 
\end{align*}
\]

Rules \( r13 \) and \( r14 \) denote that if MHi is at sleep at the time the HS agent requests transaction information from the MHs, the HS will request information from MHi when it does the decision making for the transaction.

The local knowledge base of the HS:

There are rules \( r15-r21 \) in local knowledge base of HS:

\[
\begin{align*}
\{ r15: & \text{ holds(invest(share1), } s_0) \leftrightarrow \text{ holds(profit(share1), } s_0), \neg\text{ holds(risk(share1), } s_0), \\
& \neg\text{ holds(cost(share1), } s_0), \text{ holds(info-get(MHi), res(request-info(MHi), } s_0)) \\
r16: & \neg\text{ holds(invest(share1), } s_0) \leftrightarrow \text{ holds(risk(share1), } s_0) \\
r17: & \neg\text{ holds(invest(share1), } s_0) \leftrightarrow \text{ holds(cost(share1), } s_0) \\
r18: & \neg\text{ holds(risk(share1), } s_0) \leftrightarrow \neg\text{ holds(risk(share1), } s_0) \\
r19: & \neg\text{ holds(cost(share1), } s_0) \leftrightarrow \neg\text{ holds(cost(share1), } s_0) \\
r20: & \neg\text{ holds(invest(share1), } s_0) \leftrightarrow \text{ holds(info-requested(HS, MHi), } s_0), \\
& \neg\text{ holds(info-get(MHi), res(request-info(MHi), } s_0)) \\
r21: & \neg\text{ holds(info-get(MHi), } s_0) \leftrightarrow \neg\text{ holds(info-get(MHi), } s_0) \\
& \text{ holds(info-requested(MHi), } s_0), \text{ holds(timeout(MHi), } s_0) 
\end{align*}
\]

In the following discussion of transaction processing, all the rules have been formally specified in section 5.3.3.

**Step 1**: MH1 requests to start an update transaction to invest share1. The rule is:

\[
g1: \text{ holds(update-requested(MH1), res(write(MH1), } s_0)) \leftarrow
\]

**Step 2**: After MH1 requests an update transaction and the lock is acquired, MSS1 submits this transaction request to the HS. The transaction rule is:

\[
g2: \text{ holds(trans-submitted(MSS1), res(submit(MSS1), } s_0)) \leftarrow \\
\text{ holds(update-requested(MH1), } s_0), \\
\text{ holds(timeout(MHi), res(acquire-lock(MH1), res(write(MH1), } s_0))}
\]

**Step 3**: After the MSS submits the transaction to the HS, the HS starts the transaction. The transaction rule is:
Step 4: Based on its knowledge base, the HS will know which MH agents are related to this investment request, then the HS sends this investment request to all involved MH agents and asks these agents to provide their related investment information. The rule is:

\[ g_4: \text{holds}(\text{info-requested}(MHi), \text{res}(\text{request-info}(MHi), s_0)) \leftarrow \text{holds}(\text{invest-involved}(MHi), s_0) \]

Step 5: After the MH agent receives the request from the HS, via MSS, the agent calculates the MH’s output and then sends its output to the MSS if the MH is not at sleep. The rules are:

\[ g_5: \text{holds}(\text{output-derived}(MHi), \text{res}(\text{cal-output}(MHi), s_0)) \leftarrow \text{holds}(\text{info-requested}(MHi), s_0) \]

\[ g_6: \text{holds}(\text{output-sent}(MHi), \text{res}(\text{sent-output}(MHi), s_0)) \leftarrow \text{holds}(\text{output-derived}(MHi), s_0) \]

The outputs of MH1 and MH2 are sent to MSS1, the output of MH3 is sent to MSS2, while MH4 is at sleep at the moment.

Step 6: The MSS agent collects information from all related MHs and calculates the input. MSS1 collects input from MH1 and MH2, while MSS2 collects input from MH3 and MH4. The rule is:

\[ g_7: \text{holds}(\text{input-derived}(MSSi), \text{res}(\text{cal-input}(MSSi), s_0)) \leftarrow \text{holds}(\text{output-sent}(MHi), s_0) \]

The inputs of MSS1 and MSS2 are derived accordingly.

Step 7: The output of the MSS is derived by considering its input and its own knowledge base. The rule is:

\[ g_8: \text{holds}(\text{output-derived}(MSSi), \text{res}(\text{cal-output}(MSSi), s_0)) \leftarrow \text{holds}(\text{input-derived}(MSSi), s_0) \]
Step 8: The HS agent gets the output of MSS1 and MSS2, and calculates the input of
the HS, the rule is as below:

\[
g_9: \text{holds(input-derived(HS), res(cal-input(HS), s_0))} \leftarrow
\text{holds(output-derived(MSSI), s_0)}
\]

In this study case, MH4 is at sleep at the moment MSS2 collects information from it,
MSS2 sends this information to the HS as a belief of its output and the HS reflects
this in its input.

Step 9: The output of the HS will be derived by considering its input and its own
knowledge base. The rules are:

\[
g_{10}: \text{holds(output-derived(HS), res(cal-output(HS), s_0))} \leftarrow
\text{holds(input-derived(HS), s_0)}
\]
\[
g_{11}: \text{holds(decision-made(HS), res(make-decision(HS), s_0))} \leftarrow
\text{holds(output-derived(HS), s_0)}
\]

From the discussion of study case 1 of this chapter, we know that \(\neg \text{invest/share1}\) is
in every answer set, which means the HS will make the decision that share1 can’t be
invested.

Step 10: After the decision has been made (that share1 will not be invested), the
transaction will be aborted at the HS level and the HS will update its knowledge base
to reflect this accordingly. The rules ti2 for this are:

\[
g_{12}: \text{holds(trans-aborted(HS), res(abort-trans(HS), s_0))} \leftarrow
\text{holds(decision-made(HS), s_0)}
\]
\[
g_{13}: \neg \text{holds(knowledge-update(HS), s_0)} \leftarrow \text{holds(trans-aborted(HS), s_0)}
\]

Step 11: The HS sends a transaction abort notice to the MSS. The rule reflecting this
is:

\[
g_{14}: \text{holds(abort-noticed(HS, MSSi), res(notice-abort(HS, MSSi), s_0))} \leftarrow
\text{holds(trans-aborted(HS), s_0)}
\]

Step 12: The MSS broadcasts the transaction abort information to involved MHs The
rule for this is:
Step 13: The MHs update their local knowledge base accordingly. The rule reflecting this is:

\[ g_{16} : \neg \text{holds(knowledge-update(MHi),res(update-knowledg(MHi),s_0))} \leftarrow \text{holds(abort-broadcasting(MSSi,MHi),s_0)} \]

As defined previously, a knowledge transaction in a MLPMAS system can be denoted as \( T = \langle A,C,R,I(D) \rangle \). Study case 2 presented the steps in how a transaction is processed in a MLPMAS systems showing how agents, rules, initial facts, input and output work together allowing the HS to make a decision. In this study case, agents are on each MH, MSS and HS level, and reflexive communication channels are between the MH, MSS, and HS. The initial fact and domain related rules are reflected as \( I(D) = \{i1-i4,r1-r21\} \), which includes initial facts of the transaction and every local knowledge base on the MH, MSS and HS. The generic transaction rules are reflected as \( R = \{g1-g16\} \), therefore the following fact \( \phi \) is in answer set:

\[ \phi = \{ \neg \text{holds(invest (share1), s_0)}, \]
\[ \text{holds( trans-aborted (HS), s_0)}, \]
\[ \neg \text{holds(knowledge-updated(MH), s_0) } \}

i.e. the facts that share1 isn’t invested, the transaction is aborted on the HS, and the knowledge base isn’t updated on the MHs are concluded from the Answer set of the logic programs.

5.3.5 Disconnection and Mobility in MLPMAS Systems

In Comparison to stationary environments, disconnection and mobility are two distinguished features of mobile environments. The Mobile Host can be disconnected from the network due to voluntary or involuntary sleep, or network connection issues. Also, the Mobile Host can migrate to another MSS cell after a physical movement. Study cases 3 and 4 are presented to demonstrate how these two features can be
addressed for knowledge transaction processing in the specific problem domain in the formalized MLPMAS systems.

**Study Case 3:** This study presents how a transaction is processed when MH4 is at sleep. Consider the same scenario as study case 1, but now the select function $S_F$ will take a negative atom as high preference for investment risk. Thus, $\neg \text{invest}(\text{share1})$ will become the input of $a_{MSS1}$, and the input of $a_{HS}$ becomes:

$$\text{KnowledgeInput}(a_{HS}, Y) = \text{cons}(\bigcup_{a_{MSS} \in Y} \text{KnowledgeOutput}(a_{MSS}, a_{HS}), S_F)$$

$$= \text{cons}(\text{KnowledgeOutput}(a_{MSS1}, a_{HS}) \cup \text{KnowledgeOutput}(a_{MSS2}, a_{HS}), S_F)$$

$$= \{ \text{profit(share1)}, \neg \text{risk(share1)}, \neg \text{cost(share1)}, \text{info-requested(HS, MH4)} \}$$

In this case, the HS requests information from MH4. The HS needs to know if $\text{holds}(\text{info-get(MH4)})$ is true, and if $\text{holds}(\text{profit(share1)}), \neg \text{holds(risk(share1))}$ and $\neg \text{holds(cost(share1))}$ are still true when considering the input from MH4. It is assumed that MH4 is still at sleep after the time bound. Thus $\neg \text{holds}(\text{info-get(MH4)})$ becomes true, according to the rule $r20$ in local knowledge base of the HS. In this case, $\neg \text{invest(share1)}$ is always true in every answer set, that is, the HS will make the decision that share1 cannot be invested and the transaction will be aborted accordingly. In this study case, the rules for transaction processing are almost the same as in study case 2, the only difference being an extra rule, added after rule $g9$, for the HS to request the information from MH4:

$$f1: \text{holds(info-requested(MH4), res(request-info(MH4), s_0))} \leftarrow $$

$$\text{holds(input-derived(HS), s_0)}$$

Rule $f1$ denotes: after the input of the HS is derived, the HS takes the action $\text{request-info(MH4)}$ to request information from MH4, then $\text{info-requested(MH4)}$ becomes true.

**Study Case 4:** This study case addresses how a transaction is processed after a MH’s migration from one cell to another. Given the same scenario as study case 1, the difference is MH4 will move to another cell managed by MSS3 during sleep, then
wake up and register with a new mobile support station, MSS3. In this case, when the HS requests information from MH4 via MSS2 for decision making, MSS2 tells the HS that MH4 has left its cell. Thus the HS needs to find in which cell MH4 is currently located using one of the following algorithms: broadcast [Ioannisdis, 1991; Imieliński, 1993], central service [Ma, 1992], home bases [Teraoka, 1993], and forwarding pointer [Ioannisdis, 1991], which are four basic mechanisms for determining the current address of a mobile host. After knowing that MH4 is located in cell MSS3, the HS will request MH4’s information from MSS3, and eventually it will get MH4’s information from MSS3. After the transaction is processed according to rules g1-g9 as in study case 2, the following extra rules are added to address the migration scenario:

\[ f2: \text{holds}(\text{info-requested}(\text{MH4}), \text{res}(\text{request-info}(\text{MH4}), s_0)) \leftarrow \text{holds}(\text{input-derived}(\text{HS}), s_0) \]

Rule \( f2 \) denotes: based on its derived input, the HS takes action request-info(MH4) to request information from MH4, then info-requested(MH4) becomes true.

\[ f3: \text{holds}(\text{cell-located}(\text{HS}, \text{MH4}), \text{res}(\text{locate-cell}(\text{HS}, \text{MH4}), s_0)) \leftarrow \text{holds}(\text{cell-migrated}(\text{HS}, \text{MH4}), s_0) \]

Rule \( f3 \) denotes: After the HS realises that MH4 has migrated from MSS2, i.e., cell-migrated(HS, MH4) is true, the HS takes action locate-cell(HS, MH4) to locate MH4 according to some algorithm and, then, the location of MH4 is known, i.e., cell-located(HS, MH4) becomes true.

\[ f4: \text{holds}(\text{info-get}(\text{MH4}), \text{res}(\text{request-info}(\text{MH4}), s_0)) \leftarrow \text{holds}(\text{cell-located}(\text{HS}, \text{MH4}), s_0) \]

Rule \( f4 \) denotes: after the location of MH4 is established, i.e., cell-located(HS, MH4) becomes true, the HS takes action request-info(MH4) to request information from MH4 via the new MSS whereupon information from MH4 becomes available, i.e., info-get(MH4) is true.

From the local knowledge base of MH4, given in study case 1, it is known that MH4’s output is \{\text{profit}(\text{share1}), \text{risk}(\text{share1}), \text{cost}(\text{share1})\}, and by considering the output from MH4, the input of the HS becomes:
\[ \text{KnowledgeInput}(a_{\text{HS}}, Y) = \{ \text{profit}(\text{share1}), \neg \text{risk}(\text{share1}), \neg \text{cost}(\text{share1}) \} . \]

\( \text{info-requested}(\text{HS, MH4}) \) is no longer a belief in the HS’s input.

Thus, \( \text{holds} (\text{invest(share1)}, s_0) \) is entailed by the HS’s logic programs. That is, the HS will make the decision that share1 will be invested, and the transaction will be committed, whereupon the knowledge will be updated accordingly in both the HS and the MH level.

### 5.4 Summary

This chapter presented and formalized a mobile logic programming multi-agent system (MLPMAS) for mobile environments. This formalization can be used to process knowledge transactions in such kinds of mobile multi-agent systems. Extended logic programs was employed as a specification method so this model is knowledge oriented and has declarative semantics. The formalized MLPMAS system is very useful for modeling decision-making problems for mobile applications, not just the solutions of the problem but also the evolution of the beliefs of and the interactions between the agents in mobile environments. Based on the formalized MLPMAS system, this chapter formally defined the knowledge transaction in this kind of multi-agent systems. A few study cases were illustrated to demonstrate how knowledge transactions can be processed in a particular problem domain in MLPMAS systems.

With respect to previous works, major advantages of the formalized MLPMAS system can be characterized as follows: (1) This model can be used to process knowledge transactions in multi-agent systems for mobile environments. This differs from many investigations on multi-agents and intelligent agents in that currently developed languages and models for knowledge representation, reasoning and problem solving are only limited in conventional environments and haven’t been extended to mobile environments, such as stable model/answer set, SMODEL, DLV and XSB model in [Eiter, 1997; Nemela, 1996; Rao, 1997]. (2) Extending logic programs is adopted in this formalization so this model is knowledge oriented and has
declarative semantics. (3) It can specify details of knowledge transactions, input/output and the knowledge base using knowledge rules. (2) and (3) make the formalization different from most other works on mobile agent systems in that they are not knowledge oriented processing, such as Telescript [White, 1996], Aglets [Lange, 1998], Mole [Baumann, 1998] and KLAVA [Bettini, 2002; Deugo 2001; Lange, 1998b].
This chapter formalizes a rule based knowledge transaction model for mobile environments. Logic programming is used as a mathematical tool and formal specification method to study knowledge transactions in mobile environments. This formalized model can be used for knowledge transaction representation, formalization and knowledge reasoning in mobile environments.

Section 4.1 proposes a new environmental model. Section 4.2 describes the transaction processing in mobile environments and gives background knowledge on logic programming. Then mobile semantics for some logic programming concepts and formulas are given. Section 4.3 starts from knowledge transaction representation language, and imposes a set of rules for knowledge transaction in mobile environments. Lastly, it formalizes a knowledge transaction model. Section 4.4 illustrates a study case to demonstrate that the proposed knowledge transaction model is applicable in practical scenarios. Finally section 4.5 summarizes the work in this chapter.

4.1 An Environment Model

Firstly, let us recall the three-layer mobile environment model presented in Chapter 2. There is a Home Server (HS) acting as permanent storage for Mobile Hosts' (MH) Files. There are Mobile Support Stations (MSS) providing services to a MH when it is within its cell. The MSS is connected to the HS via hardwires. The MH is continuously connected to a MSS via a wireless link while accessing data. It may, however, become disconnected either voluntarily or involuntarily. The mobile environment model is recalled and shown in Figure 4.1:
When the intelligent agent is studied in conventional environments, the following environment model [Weiss, 1999; Wooldridge, 2002] is usually used, as shown in Figure 4.2.

The idea is that function *see* captures the agent’s ability to observe its environment whereas function *action* represents the agent’s decision making process. Fundamentally, an agent is an active object with the ability to perceive, reason and act. It is assumed that an agent has explicitly represented knowledge and a mechanism for operating on, or drawing inferences from, its knowledge. Also, it is assumed that an agent has the ability to communicate. This ability consists of
perception (the receiving of messages) and action (the sending of messages). In a distributed computing system, agents need to communicate with each other in order to achieve their goals. For this purpose, the intelligent agent has been introduced for more flexible and efficient problem solving.

To study knowledge transaction in mobile environments, a new environment model is proposed, which contains intelligent agents, as shown in Figure 4.3.

Figure 4.3: New environment model.

In the environment model above, it is assumed that every Mobile Host (MH) has its own knowledge base (KB) and intelligent agent (A11, A12, A13, A21, A22, A23). Every MSS has a knowledge base and an agent residing on it as well. MSS1 and MSS2 represent different MSS in different geographic areas. At the Home Server (HS) level, there is a knowledge base that has a set of rules in it and there is an agent residing on it. Every intelligent agent on a MH will work on behalf of the MH that it resides on. All the agents in the same geographic area (i.e. controlled by same HS) will negotiate, communicate, and cooperate with each other to achieve goals for themselves and their systems. Agents can do decision making based on the rules in every knowledge base.

4.2 Mobile Transactions and Logic Programs
4.2.1 Transactions in Mobile Environments

Firstly recall a typical transaction $T$ in mobile environments, as presented in Chapter 2. It looks like the following [Chan, 1999]:

$$Contact-proxy \rightarrow r[x] \rightarrow w[x] \rightarrow contact-proxy \rightarrow page-invalidation \rightarrow \ldots \rightarrow contact-proxy \rightarrow sleep \rightarrow wakeup \rightarrow \ldots \rightarrow commit$$

The proxy first acquires the appropriate set of locks for the MH. Once a write is executed, the proxy is asked to broadcast a report of invalidation. In this example, the MH decides to go to sleep voluntarily after serving a number of operations of $T$. This is done by first contacting the proxy and then flushing all its dirty pages and releasing all the write-locks it holds. Upon waking up, the MH goes through a process similar to start-up, i.e. there will be fetches of data objects and lock requests through the proxy. Finally, the transaction is committed.

In the case of involuntary sleep the proxy is not contacted and, therefore, no invalidation report is broadcasted. Hence, any update done by the MH is lost. If the MH holds write-locks while it goes to sleep involuntarily and, if it does not wake up beyond a certain prescribed time, the write-locks are cancelled. Thus the overall effect is that of a partial abort. If the wake-up is soon enough, the locks are still with the MH and the execution of the rest of the operations can proceed.

It is also possible for a transaction to be started with the MH being in one cell and completed with the MH in another cell. In this case, the transaction $T$ becomes:

$$contact-proxy \rightarrow r[x] \rightarrow w[x] \rightarrow contact-proxy \rightarrow page-invalidation \rightarrow \ldots \rightarrow move \rightarrow contact-proxy \rightarrow \ldots \rightarrow commit$$

4.2.2 Extended Logic Program Semantics in Mobile Environments

The classical logic programming has top-down semantics based on SLD resolution, which was first introduced by Robinson [Robinson, 1965] in the mid-sixties. This semantics does not fit well with programs with negation. General logic program is a
finite set of general clauses that contain both positive and negative literals in bodies.
The resulting proof-technique of general programs is called SLDNF-resolution that is
a result of combing SLD-resolution with the negation-as-finite-failure rule [Nilsson,
200]. An important limitation of general logic programming as a knowledge
representation tool is that it does not allow us to deal directly with incomplete
information, and therefore only either yes or no answer can be got from a query. This
is because in general logic programming, closed world assumption [Reiter, 1978] is
automatically applied to all predicates [Gelfond, 1991], and each ground atom is
assumed to be false if it does not been certified to be true in the program. The query
evaluation methods of general logic programming give no answer to every query that
does not succeed, it provides no counterpart for undecided situations which mean the
incompleteness of information in classical axiomatic theories.

Although general logic programming provides a powerful tool for knowledge
representation in conventional environments, we observe it is not suitable as a
knowledge representation language for mobile environments. When knowledge
transaction is studied in mobile environments, it should be clearly understood that
there is a major different between the scenario that the transaction fails and the
transaction hangs on due to mobile user’s sleep. The first scenario is transaction fails
in the sense of its negation succeeds, it is a no answer for a query. The second
scenario is that transaction does not succeed because of incomplete information, the
answer is unknown for a query transaction, but may become a definite answer yes or
no after sometime. Therefore, in mobile environments, a method is required to deal
with incomplete information explicitly, and this method should handle the transaction
fails in the sense of its negation succeeds and the transaction that does not succeed in
the mobile situation. Extended logic programs [Baral, 1994; Gelfond, 1991; Pereira,
1992; Wagner, 1991] can overcome the limitation of general logic programming. It
contains classical negation \( \neg \) in addition to negation-as-failure not. The semantics of
such extended logic programs is based on the method of stable models. General logic
programs provide negative information implicitly through closed-world reasoning,
extended logic program can include explicit negative information. In the language of
extended logic programs, it is distinguishable between a query which fails in the
sense that it does not succeed and a query which fails in the stronger sense that its
negation succeeds. By extending extended logic program semantics to mobile environments, this method can be used to study knowledge transaction and deal with the incomplete information explicitly in mobile environments. The applicability of extended logic programs for formalization of reasoning with incomplete information has been demonstrated in paper [Baral, 1994; Gelfond, 1991; Kowalski, 1990; Pereira, 1993].

Here, extended logic program semantics are given in mobile environments: classical negation \( \neg \) is defined as explicit negative information, is explicit no when mobile transaction is explicit fail. In the situation the mobile host is in voluntary or involuntary sleep, and therefore the information is incompleteness, we say it is absent of atom \( A \), noted by not \( A \), therefore it is negation-as-failure not.

The closed world assumption in mobile environments is described as follows:
If a mobile host is in voluntary or involuntary sleep (i.e. no evidence to show \( p \) is true so it is negation-as-failure not) and the sleep time is beyond the specified limit, we assume not \( p \) to be \( \neg p \) at this time. The closed world assumption can be expressed by the rule:

\[
\neg p(x) \leftarrow \text{not } p(x), \ T \leq t
\]
Here T is the time limit for the mobile hosts’ sleep, \( t \) is mobile hosts’ sleeping time;

An extended logic program [Gelfond, 1991] is a set of rules of the form

\[
L_0 \leftarrow L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n \quad (I)
\]

Where \( 0 \leq m \leq n \), and each \( L_i \) is a literal. In the body of rule (I), both classical negation \( \neg \) and weak negation not are allowed to be presented. The meaning of (I) is as follows: if \( L_1, \ldots, L_m \) hold, and there is no explicit evidence to show that \( L_{m+1}, \ldots, L_n \) hold, then \( L_0 \) holds.

The answer set of an extended logic program is normally defined as follows [Baral, 1994; Gelfond, 1991; Baral 2003]:

Firstly, consider programs without negation-as-failure. The answer set of \( \Pi \) not containing not is the smallest subset \( S \) of Lit such that
(i) for any rule $L_0 \leftarrow L_1, \ldots, L_m$ from $\Pi$, if $L_i, \ldots, L_m \in S$, then $L_0 \in S$;
(ii) if $S$ contains a pair of complementary literals, then $S = \text{Lit}$.

Obviously, every program $\Pi$ that does not contain negation-as-failure has a unique answer set which will be denoted by $b(\Pi)$.

Let $\Pi$ be an extended program without variables. For any set $S$ of literals, let $\Pi^S$ be the logic program obtained from $\Pi$ by deleting

(i) each rule that has a formula not $L$ in its body with $L \in S$, and
(ii) all formulas of the form not $L$ in the bodies of the remaining rules.

Clearly, $\Pi^S$ does not contain not, so that its answer set is already defined. If this set coincides with $S$, then we say that $S$ is an answer set of $\Pi$. In other words, the answer sets of $\Pi$ are characterized by the equation

$$S = b(\Pi^S).$$

We say $\Pi$ entails a literal $L$ if $L$ is always true in all answer sets of $\Pi$, this is denoted by $\Pi \models L$.

As stated in [Baral, 2003], when dealing with logic programs that may have function symbols the answer sets may not have a finite cardinality and hence we need a way to finitely express such answer sets. Such an attempt was made in [Gottlob, 1996]. An alternative approach, which is also useful when we are only interested in computing entailment and not in computing one or all of the answer sets is to develop derivation methods that compute the entailment. Several such methods have been proposed in [Apt, 1994; Lifschitz, 1996; Chen, 1996].

4.3 A Logic Programming Based Transaction Model

The formalization of the logic programming based knowledge transaction model starts with defining a knowledge transaction language $\mathcal{L}$, which contains necessary components for specifying knowledge transactions associated with the MH, MSS and HS. By adopting language $\mathcal{L}$, a set of rules are then imposed to capture knowledge transaction features in mobile environments.
4.3.1 Transaction Processing at Three Levels

Let us firstly go over a transaction stage by stage to see what activities happen on the MH, MSS and HS at every transaction stage [Mirghafori, 1995].

Startup is the initial powering up of the mobile computer, after registration the MH can start a query or update transaction. Voluntary or involuntary sleep can happen during the transaction. After the sleep, the MH can wake up to continue the transaction. Also, the MH can start the transaction in one MSS cell and move to another MSS cell to continue the transaction. The last activity of the transaction is a transaction commit or abort. All the possibilities of transaction activity are shown in Figure 4.4. The transaction activity can be: startup → start → sleep → wakeup → commit; Or startup → start → move → commit. The transaction can follow the arrow in the figure to process the transaction activities.

![Figure 4.4: Transaction activities in mobile environments.](image-url)

There are different steps involved at every stage of the transaction.

**Startup:**
- Step 1: When the MH is powered on, it registers with MSS. Upon registration, the MH notifies the MSS of its HS address.
- Step 2: The MSS then creates the proxy process which retrieves the MH profile from HS.
- Step 3: The HS sends the pages in the MH profile, marks the MH as a valid reader of those pages, and notes where to contact the MH.
- Step 4: The proxy receives and caches the MH profile.
Step 5: The proxy then broadcasts the sub-profile to the MH, and marks the in-MH-cache bit for those pages.
Step 6: The MH receives and caches the sub-profile pages.

Start Transaction
Step 1: The MH requests a query or update transaction.
Step 2: The MH acquires a lock if it is an update transaction.
Step 3: The MSS submits the transaction request to the HS.
Step 4: The HS does the transaction after the MSS submits the transaction request.

Sleep: There are essentially two types of sleep – voluntary and involuntary. Voluntary sleep is a planned power down, while involuntary sleep is an unplanned power down, i.e., the system crashes or runs out of battery power.

Voluntary Sleep:
Step 1: The MH flushes its dirty pages, gives up any write locks it holds, and informs the proxy of its intention to sleep.
Step 2: The proxy updates MH-sleep-time and buffers messages and invalidations for the MH until the MH wakes up and is ready to receive them.

Involuntary Sleep:
In this case, the proxy does not know that the MH is not listening and continues to broadcast invalidations as normal. The MH will recover missed messages upon wakeup by asking for all messages sent after its disconnection time. Suppose the MH is holding a write lock when it goes to sleep involuntarily and, in the mean time, another writer asks the HS for the write lock. The HS forwards the request to the proxy, and the proxy forwards it to the MH. If the proxy does not receive the lock from the MH in a limited amount of time, it invalidates the lock and sends it back to the HS. The MH will inevitably lose the updates it had made.

One potential problem is that the sleeping MH process may not return (e.g., MH dies or leaves the cell), in which case the proxy may wait around aimlessly. To remedy this problem, the MH state is sent to the HS after a system-specific amount of time.
and the proxy process is killed (the decision is made based on MH-sleep-time or time-MH-contacted-proxy). Should the MH return, the wakeup is treated as a startup process.

**Wakeup:** Wakeup is the powering up after a MH has been asleep. Although similar to startup, wakeup has slightly different semantics. The wakeup sequence is as follows:

Step 1: Upon wakeup, the MH waits to get MSS’s address (beacon).

Step 2: Upon receiving the beacon, the MH sends a wakeup notification to the MSS and requests missed messages. If the MH has saved its MH-sleep-time (possible in the case of voluntary sleep), it sends this information to the MSS. Otherwise, if the MH had gone to sleep involuntarily (e.g., power failure), the MSS proxy uses time-MH-contacted-proxy and time-invalidation-propagated for each page on its cache to calculate how many old messages should be resent.

**Move/Handoff:**

Step 1: The MH listens for a MSS beacon.

Step 2: When the MH notices that it is in a different region, it contacts the new MSS. The message sent includes the MH’s id, MH’s HS, MH’s old MSS, and last-time-MSS-contacted-MH (for the old MSS).

Step 3: The new MSS contacts the old MSS to get the state of the MH proxy.

Step 4: The old MSS flushes any dirty pages to the HS and sends the proxy state to the new MSS.

Step 5: The new MSS proxy contacts the HS to tell it where to contact the MH.

Step 6: The new MSS proxy broadcasts any invalidations whose timestamp is later than last-time-MSS-contacted-MH.

**Commit Transaction:**

Step 1: The HS decides if the transaction should be committed or aborted according to the two-phase commit protocol, then commits or aborts the transaction.

Step 2: The HS sends the transaction result to the MSS.

Step 3: The MSS broadcasts the transaction result to the MH.

Step 4: The MH updates the local knowledge base according to the transaction result.
4.3.2 Logic Programming Formalizations I: The Language £

The syntax of the language £ contains variables of three sorts: situation variables $s$, fluent variables $f$, and action variables $a$. Language £ has one predicate $\text{holds}(f, s)$, where $f$ is a fluent function, and $s$ is a situation. The predicate $\text{holds}(f, s)$ means that fluent $f$ is true at situation $s$. The term fluent [Aiello, 2001] represents predicates and functions whose value changes over time. £ has a resulting function $\text{res}(a, s)$, which denotes a situation resulting from situation $s$ by performing action $a$. £ also has one initial situation constant $S_0$. There are, also, some other predicates and function symbols. The types of their arguments and values will be clear from their use in the rest of this chapter. The language £ inherits situation calculus with extension to mobile environments. It has a few important features as follows: (1) It can deal with weak negation in addition to classical negation and therefore can be used for incomplete information reasoning in mobile environments; (2) It has declarative semantics; (3) This kind of Language has been implemented in conventional environments such as SMODEL [Niemela, 1997].

Example 1 illustrates how to represent knowledge in logic programs:

**Example 1:** Given a complete description of the initial state of the world and a complete description of the effects of actions, we want to determine what the world will look like after a series of actions are performed [Baral, 1994]. The most frequently cited example of such reasoning is probably the Yale Shooting Problem (YSP) from [Hanks, 1987]. In the Yale Shooting Problem [Baral, 1994], there are two fluents: alive and loaded, and three actions: wait, load, and shoot. It is known that the execution of loading leads to the gun being loaded, and that if the gun is shot while it is loaded, a turkey, Fred, dies. It is predicted that after the execution of actions load, wait, and shoot (in that order), Fred will be dead. It seems that the commonsense argument which leads to this conclusion is based on the so called axiom of inertia which says, “Things normally tend to stay the same”. As a typical normative statement, it can be represented by the rule $y1$: 

$$y1: \text{holds}(F, \text{res}(A,S)) \leftarrow \text{holds}(F, S), \text{not ab } (y1, A, F, S)$$

To represent the effect of the actions load, shoot, and wait, we need only the rule $y2$: 

$$y2: \text{holds}($load$ed, \text{res}(load, S)) \leftarrow$$
And the cancellation rule $y_3$:

$$ y_3: ab(y_1, shoot, alive, S) \leftarrow holds (loaded, S) $$

which represents the priority of specific knowledge about the results of actions over the general law of inertia. Let $S_0$ be the initial state, and suppose it is given that:

$$ y_4: holds (alive, S_0) $$

Even though the resulting program $\gamma$ consisting of $y_1$-$y_4$ is not stratified, it is possible to show that it has a unique stable model. It is easy to see that $\gamma$ entails:

$$ holds (alive, res(load, S_0)), \text{ and } \neg holds (alive, res(shoot, res(wait, res(load, S_0)))) $$

Example 2 is another knowledge representation example to discuss a very simple knowledge base update transaction:

**Example 2**: Given only one fluent, $updated$, and two actions, $commit$ and $abort$, it is known that $commit$ will result in the knowledge update. It is predicted that after the execution of action $commit$, the knowledge base will be updated.

$$ K1: holds (F, res(A, S)) \leftarrow holds (F, S), \text{ not } ab (K1, A, F, S) $$

To represent the effect of the action $commit$, only the rule $K2$ is needed:

$$ K2: holds(updated, res(commit, S)) \leftarrow $$

If the transaction is aborted, the knowledge base won’t be updated. So we have the cancellation rule $K3$:

$$ K3: ab(K1, abort, updated, S) \leftarrow $$

To formalize the rule-based knowledge transaction model, a transaction processing language $£$ is firstly defined to formalize transaction related actions and fluent functions at the MH, MSS and HS level. The different sorts of functions are introduced to characterize the basic components of this language. The actions and fluents are used to denote transaction processing activities, results and status. The language $£$ has the following specific action and fluent functions at different levels.

**Mobile Host (MH) level**:

The following action functions are defined in $£$:

$Move(y, x)$: denotes MH $x$ moves into MSS $y$ cell.

$Query(x)$: denotes MH $x$ has a query transaction request.
Chapter 4. A Rule Based Knowledge Transaction Model for Mobile Environments

Write(x): denotes MH x has a update transaction request.
Acquire-lock(x): denotes MH x requests a lock.
Flush(x): denotes MH x flushes its dirty pages.
Release-lock(x): denotes MH x releases its write lock.
Request-message(x): denotes MH x requests missed message from MSS y.
Fetch-message(x): denotes MH x fetches missed message from MSS y.
Update-knowledge(x): denotes MH x updates local knowledge base according to the transaction result.

The following fluent functions are defined in £:
Registered(y1, x): denotes MH x has registered in MSS1 y1.
Query-requested(x): denotes MH x has requested to start a query transaction.
Trans-start(x): denotes MH x has started a transaction.
Update-requested(x): denotes MH x has requested to start a update transaction.
Locked(x): denotes MH x has got a lock for an update transaction.
Vol-slept(x): denotes MH x has gone to voluntary sleep.
Sleep-sig(x): denotes there is a sleep signal on MH x.
Invol-slept(x): denotes MH x has gone to involuntary sleep.
Lock-cancelled(x): denotes write lock has been cancelled on MH x.
Update-lost(x): denotes updates have been lost on MH x.
Wakeup-sig(x): denotes there is a wakeup signal on MH x.
Message-received(x): denotes MH x has received missed messages.
Registered(y2, x): denotes MH x has registered in MSS2 y2.
Knowledge-updated(x): denotes MH x has updated the local knowledge base.
Commit-broadcasting(y, x): denotes MSS y has noticed transaction commit to MH x.
Abort-broadcasting(y, x): denotes MSS y has noticed transaction abort to MH x.
Timeout1(x): denotes MH x has expired after a certain time1.

Mobile Support Station (MSS) level:

The following action functions are defined in £:
Create-proxy(y1, x): denotes MSS1 y1 creates proxy process for MH x.
Create-proxy(y2, x): denotes MSS2 y2 creates proxy process for MH x.
Chapter 4. A Rule Based Knowledge Transaction Model for Mobile Environments 79

Retrieve(y): denotes MSS y proxy retrieves the MH x profile from the HS.
Cache(y): denotes MSS y proxy caches the MH profile.
Broadcast(y): denotes MSS y proxy broadcasts the sub-profile to the MH.
Mark(y): denotes MSS y proxy marks the in-MH-cache bit for those broadcasting pages.
Submit(y): denotes the transaction has been submitted to the HS by MSS y.
Update-sleeptime(y): denotes MSS y proxy updates MH-sleep-time.
Buffer(y): denotes MSS y proxy buffers the message and invalidations for the MH until the MH wakes up.
Page-broadcast(y1, x): denotes MSS1 y1 proxy does the page broadcasting to MH x.
Page-broadcast(y2, x): denotes MSS2 y2 proxy does the page broadcasting to MH x.
Cancel-lock(y): denotes MSS y proxy cancels the lock for transaction.
Flush(y1): denotes MSS1 y1 flushes any dirty pages to the HS.
Broadcast(y, x): denotes MSS y broadcasts the transaction result to MH x.

The following fluent functions are defined in £:
Proxy(y1, x): denotes MSS1 y1 has created proxy process for MH x.
Proxy(y2, x): denotes MSS2 y2 has created proxy process for MH x.
Cached(y): denotes MSS y proxy has cached the MH profile.
Broadcasting(y): denotes MSS y proxy has done the sub-profile broadcasting to the MH.
Marked(y): denotes MSS y proxy has marked in-MH-cache bit for pages.
Trans-submitted(y): denotes that the transaction has been submitted by MSS y.
Buffered(y): denotes MSS y proxy has buffered messages and invalidations for the MH.
Page-broadcasting(y): denotes MSS y proxy has done the page broadcasting.
Timeout(x): denotes involuntary sleep time has expired on MH x.
Move-sig(y2): denotes MH has moved to MSS2 y2.
Flushed(y1): denotes MSS1 y1 has flushed any dirty pages to the HS.
Commit-broadcasting(y, x): denotes MSS y has noticed transaction committing to MH x.
Abort-broadcasting(y, x): denotes MSS y has noticed transaction aborting to MH x.
Commit-noticed (z, y): denotes HS z has given transaction commit notice to MSS y.
Abort-noticed \((z, y)\): denotes HS \(z\) has given transaction abort notice to MSS \(y\).

**Home Server (HS) level:**

The following action functions are defined in \(\mathcal{E}\):

- **Send-page**(\(z, y\)): denotes HS \(z\) sends the pages to MSS \(y\) in the MH profile.
- **Mark-MH**(\(z\)): denotes HS \(z\) marks the MH as a valid reader of those pages.
- **Do-trans**(\(z\)): denotes HS \(z\) does the transaction.
- **Kill-proxy**(\(z, y\)): denotes HS \(z\) kills the MSS \(y\) proxy process.
- **Send-dirty-page**(\(z, y_2\)): denotes HS \(z\) sends any dirty pages to new MSS \(y_2\).
- **Commit**(\(z\)): denotes HS \(z\) commits the transaction.
- **Notice-commit**(\(z, y\)): denotes HS \(z\) gives transaction commit notice to MSS \(y\).
- **Notice-abort**(\(z, y\)): denotes HS \(z\) gives transaction abort notice to MSS \(y\).

The following fluent functions are defined in \(\mathcal{E}\):

- **Sent**(\(z, y\)): denotes HS \(z\) has sent pages to MSS \(y\).
- **MH-marked**(\(z\)): denotes HS \(z\) has marked the MH as a valid reader for those pages.
- **Trans-started**(\(z\)): denotes HS \(z\) has started transaction processing.
- **Proxy-killed**(\(z, y\)): denotes HS \(z\) has killed MSS \(y\) proxy process for the MH.
- **Trans-committed**(\(z\)): denotes transaction has been committed on HS \(z\).
- **Commit-agreed** (\(z\)): denotes HS \(z\) has got transaction commit agreement.
- **Abort-agreed** (\(z\)): denotes HS \(z\) has got transaction abort agreement.
- **Timeout2**(\(z\)): denotes HS \(z\) has expired a certain time \(2\).
- **Commit-noticed**(\(z, y\)): denotes HS \(z\) has given transaction commit notice to MSS \(y\).
- **Abort-noticed**(\(z, y\)): denotes HS \(z\) has given transaction abort notice to MSS \(y\).

**4.3.3 Logic Programming Formalizations II: Modeling**

Based on the knowledge transaction language \(\mathcal{E}\) defined above, a set of rules are specified and imposed to capture knowledge transaction features and formalize a knowledge transaction processing model in mobile environments, which models all transaction processing activities, requests, results and constraints on the MH, MSS, and HS levels.
Mobile Host (MH) level:

Register
When the MH moves into an MSS cell, it is registered. The rule for this is as follows:

\[ r_1: \text{holds}(\text{registered}(y_1, x), \text{res}(\text{move}(y_1, x), s)) \leftarrow \]
Rule \( r_1 \) denotes: after an action \( \text{move}(y_1, x) \) happens, \( \text{res}(\text{move}(y_1, x), s) \) becomes the current situation, and \( \text{registered}(y_1, x) \) is true.

Start a query or update transaction
For a query transaction, as long as the MH has a transaction request, the transaction should be started straight away. The rules are as follows:

\[ r_2: \text{holds}(\text{query-requested}(x), \text{res}(\text{query}(x), s)) \leftarrow \]
Rule \( r_2 \) denotes: after an action \( \text{query}(x) \), i.e., the MH submits a query request, \( \text{res}(\text{query}(x), s) \) becomes the current situation, and \( \text{query-requested}(x) \) becomes true.

\[ r_3: \text{holds}(\text{trans-start}(x), s) \leftarrow \text{holds}(\text{query-requested}(x), s) \]
Rule \( r_3 \) denotes: after \( \text{query-requested}(x) \) becomes true, \( \text{trans-start}(x) \) is true for a query transaction.

For an update transaction, after the MH has a write request, the lock needs to be acquired firstly to start this transaction. The transaction will start after the lock is available.

\[ r_4: \text{holds}(\text{update-requested}(x), \text{res}(\text{write}(x), s)) \leftarrow \]
Rule \( r_4 \) denotes: after an action \( \text{write}(x) \), i.e., the MH submits a write request, \( \text{res}(\text{write}(x), s) \) becomes the current situation, and \( \text{update-requested}(x) \) becomes true.

\[ r_5: \text{holds}(\text{trans-start}(x), s) \leftarrow \text{holds}(\text{locked}(x), \text{res}(\text{acquire-lock}(x), \text{res}(\text{write}(x), s))), \text{holds}(\text{update-requested}(x), s) \]
Rule \( r_5 \) denotes: after an action \( \text{write}(x) \), another action \( \text{acquire-lock}(x) \) takes place, \( \text{res}(\text{acquire-lock}(x), \text{res}(\text{write}(x), s)) \) becomes the current situation, \( \text{locked}(x) \) becomes
true, i.e., the MH gets a lock. If both locked(x) and update-requested(x) are true, trans-start(x) is true as well, i.e., an update transaction starts.

Sleep
For a voluntary sleep, the MH informs the proxy of its intention to sleep, flushes its dirty pages, and gives up any write-locks it holds. After this, the MH goes to voluntary sleep.

\[ r6: \text{holds}(vol-slept(x), \text{res}(release-lock(x), \text{res}(\text{flush}(x), s))) \leftarrow \text{holds}(sleep-sig(x), s) \]

Rule r6 denotes: after sleep-sig(x) becomes true, action flush(x) and release-lock(x) take place, and vol-slept(x) becomes true eventually, i.e., the MH is in voluntary sleep.

For an involuntary sleep, it is supposed that the MH is holding a write-lock when it goes to involuntary sleep and in the meantime, lock is requested by another writer. The HS forwards the request to the proxy, and the proxy forwards it to the MH. If the proxy does not receive the lock from the MH in a limited amount of time, it invalidates the lock and sends it back to the HS. Therefore, the MH will inevitably lose the updates it had made.

\[ r7: \text{holds}(invol-slept(x), s) \leftarrow \]

Rule r7 denotes: the invol-slept(x) is true, i.e., the MH is in involuntary sleep.

\[ r8: \text{holds}(lock-cancelled(x), s) \leftarrow \text{holds}(invol-slept(x), s), \text{holds}(timeout(x), s) \]

Rule r8 denotes: if both invol-slept(x) and timeout(x) are true, lock-cancelled(x) will become true as well, i.e., if the MH is in involuntary sleep and doesn’t respond to the proxy in a specified time period, then the lock is cancelled on the MH.

\[ r9: \text{holds}(update-lost(x), s) \leftarrow \text{holds}(lock-cancelled(x), s) \]

Rule r9 denotes: after lock-cancelled(x) becomes true, update-lost(x) becomes true as well, i.e., the MH loses its update.
Wake up
Upon the MH waking up, the MH sends wake-up notification to the MSS and requests missed messages. If the MH has saved its MH-sleep-time, it sends this information to the MSS. If the MH wakes up from involuntary sleep, the MSS proxy will use time-MH-contacted-proxy and time-invalidation-propagated for each page on its cache to calculate how many old messages should be resent.

\[ r_{10}: \text{holds}(\text{message-received}(x), \text{res}((\text{fetch-message}(x), \text{res}(\text{request-message}(x), s)))) \leftarrow \text{holds}(\text{wakeup-sig}(x), s) \]

Rule \( r_{10} \) denotes: after \( \text{wakeup-sig}(x) \) becomes true, the MH has action \( \text{request-message}(x) \) and \( \text{fetch-message}(x) \), then \( \text{res}(\text{fetch-message}(x), \text{res}(\text{request-message}(x), s)) \) becomes the current situation, then \( \text{message-received}(x) \) is true on the MH.

Move/handoff
When the MH notices that it is in a different region, it contacts the new MSS. The message sent includes the MH’s id, MH’s HS, MH’s old MSS and last-time-MSS-contacted-MH for old MSS. After the new MSS contacts the HS and the old MSS, the MSS proxy broadcasts any invalidations whose timestamp is later than last-time-MSS-contacted-MH.

\[ r_{11}: \text{holds}(\text{registered}(y_2, x), s) \leftarrow \text{holds}(\text{move-sig}(y_2, x), \text{res}(\text{move}(y_2, x), s)) \]

Rule \( r_{11} \) denotes: after action \( \text{move}(y_2, x) \) on MH, \( \text{move-sig}(y_2, x) \) becomes true, and then \( \text{registered}(y_2, x) \) becomes true as well, which means the MH moves and registers in a new MSS.

\[ r_{12}: \text{holds}(\text{message-received}(y_2, x), s) \leftarrow \text{holds}(\text{registered}(y_2, x), s) \]

Rule \( r_{12} \) denotes: after \( \text{registered}(y_2, x) \) is true, \( \text{message-received}(y_2, x) \) becomes true as well, the MH receives necessary messages from the new MSS.

Transaction commit
After the MH requests a transaction, the transaction will be committed or aborted on the HS according to the two-phase commit protocol [Blaybrook, 1992]. After that the HS sends the transaction result to the MSS, the MSS broadcasts the transaction result to the MH, and the MH updates the local knowledge base accordingly based on a
transaction commit or abort. If, after the transaction has started, during a period time (timeout1) the MH host still has not got any transaction commit or abort notification from the MSS for whatever reason, the closed world assumption is used. Here, assume not 
\( p \) as \( \neg p \) at this time, and unknown (not) becomes no (\( \neg \) ) for the transaction.

\[ r13: \text{holds}(\text{knowledge-updated}(x), \text{res}(\text{update-knowledge}(x), s)) \leftarrow \text{holds}(\text{commit-broadcasting}(y, x), s), \text{holds}(\text{trans-start}(x), s) \]

Rule \( r13 \) denotes: if both commit-broadcasting\( (y, x) \) and trans-start\( (x) \) are true, then action update-knowledge\( (x) \) happens, knowledge-updated\( (x) \) becomes true.

\[ r14: \neg \text{holds}(\text{knowledge-updated}(x), s) \leftarrow \text{holds}(\text{abort-broadcasting}(y, x), s), \text{holds}(\text{trans-start}(x), s) \]

Rule \( r14 \) denotes: if both abort-broadcasting\( (y, x) \) and trans-start\( (x) \) are true, then we know knowledge-updated\( (x) \) is not true.

\[ r15: \neg \text{holds}(\text{knowledge-updated}(x), s) \leftarrow \text{holds}(\text{trans-start}(x), s), \text{holds}(\text{timeout1}(x), s), \neg \text{holds}(\text{commit-broadcasting}(y, x), s), \neg \text{holds}(\text{abort-broadcasting}(y, x), s) \]

Rule \( r15 \) denotes: If neither commit-broadcasting\( (y, x) \) nor abort-broadcasting\( (y, x) \) holds (is true), and time is over the limit, i.e., timeout1\( (x) \) holds, then we know holds(knowledge-updated\( (x) \), s) is false.

**Mobile Support Station (MSS) level:**

**Register**

After the MH registers with the MSS, the MSS creates the proxy process which retrieves the MH profile from the HS, the proxy receives and caches the MH profile, and then broadcasts the sub-profile to the MH, and marks the in-MH-cache bit for those pages. The rules for these are as follows:

\[ r1: \text{holds}(\text{proxy}(y, x), \text{res}(\text{create-proxy}(y, x), s)) \leftarrow \text{holds}(\text{registered}(y, x), s) \]
Rule $r_1$ denotes: after $\text{registered}(y, x)$ becomes true, the action $\text{create-proxy}(y, x)$ happens, $\text{res}(\text{create-proxy}(y, x), s)$ becomes the current situation, then $\text{proxy}(y, x)$ holds, which means the MSS creates a proxy for the MH.

$r_2$: $\text{holds}(\text{cached}(y), \text{res}(\text{cache}(y), \text{res}(\text{retrieve}(y), s))) \leftarrow \text{holds}(\text{proxy}(y, x), s)$
Rule $r_2$ denotes: after $\text{proxy}(y, x)$ becomes true, actions $\text{retrieve}(y)$ and $\text{cache}(y)$ take place, then $\text{cached}(y)$ is true, i.e., the MSS has cached for the MH.

$r_3$: $\text{holds}(\text{broadcasting}(y, x), \text{res}(\text{broadcast}(y, x), s)) \leftarrow \text{holds}(\text{cached}(y, x), s)$
Rule $r_3$ denotes: after $\text{cached}(y, x)$ becomes true, action $\text{broadcast}(y)$ takes place, then $\text{broadcasting}(y)$ is true.

$r_4$: $\text{holds}(\text{marked}(y), \text{res}(\text{mark}(y), s)) \leftarrow \text{holds}(\text{broadcasting}(y, x), s)$
Rule $r_4$ denotes: after $\text{broadcasting}(y)$ becomes true, action $\text{mark}(y)$ happens, then $\text{marked}(y)$ becomes true, i.e., the MSS has marked broadcasting pages for the MH.

**Start a query or update transaction**

After the MH requests a query or update, the MSS submits this transaction request to the HS on behalf of the MH. If it is a write request, the lock needs to be acquired firstly to submit this transaction.

$r_5$: $\text{holds}(\text{trans-submitted}(y), \text{res}(\text{submit}(y), s)) \leftarrow \text{holds}(\text{update-requested}(x), s)$
Rule $r_5$ denotes: after $\text{update-requested}(x)$ becomes true, action $\text{submit}(y)$ happens, $\text{res}(\text{submit}(y), s)$ becomes the current situation, and $\text{trans-submitted}(y)$ then becomes true.

$r_6$: $\text{holds}(\text{trans-submitted}(y), \text{res}(\text{submit}(y), s)) \leftarrow \text{holds}(\text{locked}(x), s), \text{holds}(\text{update-requested}(x), s))$
Rule $r_6$ denotes: if both $\text{update-required}(x)$ and $\text{locked}(x)$ are true, action $\text{submit}(y)$ will happen and $\text{trans-submitted}(y)$ then becomes true.
Sleep
For the voluntary sleep, the proxy updates MH-sleep-time and buffers messages and invalidations for the MH until the MH wakes up and is ready to receive them.

\[ r7: \text{holds}(\text{buffered}(y), \text{res}(\text{buffer}(y), \text{res}(\text{update-sleeptime}(y), s))) \leftarrow \text{holds}(\text{vol-slept}(y, x), s) \]

Rule \( r7 \) denotes: after \( \text{vol-slept}(y, x) \) becomes true, action \( \text{update-sleeptime}(y) \) and \( \text{buffer}(y) \) will happen continuously and then \( \text{res}(\text{buffer}(y), \text{res}(\text{update-sleeptime}(y), s)) \) become the current situation, and \( \text{buffered}(y) \) becomes true.

In the involuntary sleep case, the proxy doesn’t know that the MH is not listening and continues to broadcast invalidations as normal. If the MH is holding a write-lock when it goes to involuntary sleep, and in the meantime the lock is asked by another writer, the proxy forwards this request to the MH. If the proxy does not receive the lock from the MH in a limited amount of time, it invalidates the lock and sends it back to the HS.

\[ r8: \text{holds}(\text{page-broadcasting}(y, x), \text{res}(\text{page-broadcast}(y, x), s)) \leftarrow \]

Rule \( r8 \) denotes: in the involuntary sleep case, action \( \text{page-broadcast}(y, x) \) still happens, \( \text{page-broadcasting}(y, x) \) is still true.

\[ r9: \text{holds}(\text{lock-cancelled}(y, x), \text{res}(\text{cancel-lock}(y, x), s)) \leftarrow \text{holds}(\text{invol-slept}(x), s), \text{holds}(\text{timeout}(x), s)) \]

Rule \( r9 \) denotes: if both \( \text{invol-slept}(x) \) and \( \text{timeout}(x) \) are true, action \( \text{cancel-lock}(y, x) \) will happen, i.e., proxy will cancel the lock for the MH and then \( \text{lock-cancelled}(y, x) \) becomes true.

Wake up
Upon the MH waking up, the MH sends wakeup notification to the MSS and requests missed messages. If the MH has saved its MH-sleep-time, it sends this information to MSS. If the MH wakes up from involuntary sleep, the MSS proxy will use time-MH-contacted-proxy and time-invalidated-propagated for each page on its cache to calculate how many old messages should be resent.
Chapter 4. A Rule Based Knowledge Transaction Model for Mobile Environments

\[ r10: \text{holds}(\text{page-broadcasting}(y, x), \text{res}(\text{page-broadcast}(y, x), s)) \leftarrow \text{holds}(\text{wakeup-sig}(x), s) \]

Rule \( r10 \) denotes: if \( \text{wakeup-sig}(x) \) is true, action \( \text{page-broadcast}(y, x) \) will take place, then \( \text{page-broadcasting}(y, x) \) becomes true.

**Move/handoff**

After the MH contacts and registers in the new MSS, the new MSS contacts the old MSS to get the MH proxy status. The old MSS flushes any dirty pages to the HS and sends the proxy status to the new MSS. The new MSS proxy contacts the HS to tell it where to contact the MH. The new MSS proxy broadcasts any invalidations whose timestamp is later than last-time-MSS-contacted-MH.

\[ r11: \text{holds}(\text{proxy}(y_2, x), \text{res}(\text{create-proxy}(y_2, x), s)) \leftarrow \text{holds}(\text{registered}(y_2, x), s) \]

Rule \( r11 \) denotes: after \( \text{registered}(y_2, x) \) becomes true, i.e., the MH has registered in new MSS, action \( \text{create-proxy}(y_2, x) \) happens, and then \( \text{proxy}(y_2, x) \) is true.

\[ r12: \text{holds}(\text{flushed}(y_1), \text{res}(\text{flush}(y_1), s)) \leftarrow \text{holds}(\text{move-sig}(y_2, x), s) \]

Rule \( r12 \) denotes: after \( \text{move-sig}(y_2, x) \) becomes true, action \( \text{flush}(y_1) \) happens and then \( \text{flushed}(y_1) \) becomes true, i.e., the old MSS flushes dirty pages for the MH.

\[ r13: \text{holds}(\text{page-broadcasting}(y_2, x), \text{res}(\text{page-broadcast}(y_2, x), s)) \leftarrow \text{holds}(\text{proxy}(y_2, x), s) \]

Rule \( r13 \) denotes: after \( \text{proxy}(y_2, x) \) becomes true, i.e., the new MSS has created a proxy for the MH, action \( \text{page-broadcast}(y_2, x) \) happens, \( \text{res}(\text{page-broadcast}(y_2, x), s) \) becomes the current situation and \( \text{page-broadcasting}(y_2, x) \) then becomes true.

**Transaction commit**

After the MSS gets the transaction commit or abort notice from the HS, the MSS will broadcast the transaction result to the MH accordingly.

\[ r14: \text{holds}(\text{commit-broadcasting}(y, x), \text{res}(\text{broadcast}(y, x), s)) \leftarrow \text{holds}(\text{commit-noticed}(z, y), s) \]
Rule $r_{14}$ denotes: if $commit-noticed(z, y)$ is true, i.e., the MSS has got a commit notice from the HS, action $broadcast(y, x)$ will happen and $commit-broadcasting(y, x)$ then becomes true, which means the MSS will send commit broadcasting to the MH.

$$r_{15}: \text{holds}(abort-broadcasting(y, x), res(broadcast(y, x), s)) \leftarrow \text{holds}(abort-noticed(z, y), s)$$

Rule $r_{15}$ denotes: if $abort-noticed(z, y)$ is true, i.e., the MSS has got abort notice from the HS, action $broadcast(y, x)$ will happen and $abort-broadcasting(y, x)$ then becomes true, which means the MSS will send abort broadcasting to the MH.

**Home Server (HS) level:**

**Register**

At registration stage, the HS sends the pages in the MH profile, marks the MH as a valid reader of those pages, and notes where to contact the MH.

$$r_{1}: \text{holds}(sent(z, y), res(send-page(z, y), s)) \leftarrow \text{holds}(proxy(y, x), s)$$

Rule $r_1$ denotes: if $proxy(y, x)$ is true, action $send-page(z, y)$ will take place, then $sent(z, y)$ becomes true.

$$r_{2}: \text{holds}(MH-marked(z), res(mark-MH(z), s)) \leftarrow \text{holds}(sent(z, y), s)$$

Rule $r_2$ denotes: after $sent(z, y)$ becomes true, action $mark-MH(z)$ will happen, $MH-marked(z)$, then, is true as well.

**Start transaction**

After the MH requests a query or update transaction and the MSS submits this transaction request to the HS, the HS starts the transaction. In update transaction situation, the MSS submits a transaction only when a lock is available.

$$r_{3}: \text{holds}(trans-start(z), res(do-trans(z), s)) \leftarrow \text{holds}(trans-submitted(y), s)$$

Rule $r_3$ denotes: After a transaction request is submitted by the MSS, i.e., $trans-submitted(y)$ is true, action $do-trans(z)$ happens, and $trans-start(z)$ then becomes true, i.e. the transaction starts.
Chapter 4. A Rule Based Knowledge Transaction Model for Mobile Environments

Sleep
The sleeping MH process may not return (e.g., the MH dies, leaves cell), in this case the proxy may wait around aimlessly. To remedy this problem, the MH status is sent to the HS after a system-specific amount of time and the proxy process is killed. The decision is made based on MH-sleep-time or time-MH-contacted-proxy.

\[ r4: \text{holds}(\text{proxy-killed}(z, y), \text{res}(\text{kill-proxy}(z, y), s)) \leftarrow \text{holds}(\text{timeout}(x), s), \text{holds}(\text{vol-slept}(x), s) \]

\[ r5: \text{holds}(\text{proxy-killed}(z, y), \text{res}(\text{kill-proxy}(z, y), s)) \leftarrow \text{holds}(\text{timeout}(x), s), \text{holds}(\text{invol-slept}(x), s) \]

Rules \( r4 \) and \( r5 \) denote: If the MH goes to sleep, i.e., \( \text{vol-slept}(x) \) or \( \text{invol-slept}(x) \) is true, and \( \text{timeout}(x) \) is true as well, action \( \text{kill-proxy}(z, y) \) will happen and \( \text{proxy-killed}(z, y) \) then becomes true.

Move/handoff
After the MH moves to a new MSS, the old MSS will flush any dirty pages to the HS and the new MSS will contact the HS to get these dirty pages regarding the MH.

\[ r6: \text{holds}(\text{sent}(z, y2), \text{res}(\text{send-dirtypage}(z, y2), s)) \leftarrow \text{holds}(\text{proxy}(y2, x), s) \]

Rule \( r6 \) denotes: After the new MSS creates a proxy for the MH, i.e., \( \text{proxy}(y2, x) \) is true, action \( \text{send-dirtypage}(z, y2) \) will happen, \( \text{sent}(z, y2) \) then becomes true and the pages are sent to the MSS from the HS.

Transaction commit
According to the two-phase commit protocol, if all involved MHs agree to commit the transaction, then the transaction will be committed. If any of them does not agree to commit and wants to abort the transaction, then the transaction will be aborted. If after a time bound (timeout2) the transaction is still not agreed to be committed, then closed world assumption is used to assume the transaction won’t be committed any more and unknown (not) becomes no (\( \neg \)) in this scenario. For example in the case where one of the involved MHs has gone to voluntary or involuntary sleep, no commit agreement can be available from that MH during this time duration. After the
transaction has been committed or aborted, the HS will send a transaction commit or abort notice to the MSS.

\[
\textbf{r7: } \text{holds}(\text{trans-committed}(z), \text{res}(\text{commit}(z), s)) \leftarrow \\
\quad \text{hold}(\text{commit-agreed}(z), s), \text{holds}(\text{trans-start}(z), s), \text{not holds}(\text{abort-agreed}(z), s)
\]

Rule \( r7 \) denotes: if commit-agreed(\( z \)) and trans-start(\( z \)) are true, and abort-agreed(\( z \)) is not true, i.e. all the parts agree to commit and none of them agrees to abort, then commit action commit(\( z \)) will take place, and the transaction is committed, i.e., trans-committed(\( z \)) becomes true.

\[
\textbf{r8: } \neg \text{holds}(\text{trans-committed}(z), s) \leftarrow \\
\quad \text{holds}(\text{abort-agreed}(z), s), \text{holds}(\text{trans-start}(z), s)
\]

Rule \( r8 \) denotes: if any part agrees to abort, i.e. abort-agreed(\( z \)) is true, trans-committed(\( z \)) will not be true any more.

\[
\textbf{r9: } \neg \text{holds}(\text{trans-committed}(z), s) \leftarrow \\
\quad \text{holds}(\text{timeout2}(z), s), \text{holds}(\text{trans-start}(z), s), \text{not holds}(\text{trans-committed}(z), s)
\]

Rule \( r9 \) denotes: if transaction commit is not confirmed, i.e. trans-committed(\( z \)) does not hold, and time is over the limit, i.e. holds(timeout2(\( z \), s)), the transaction commit will be assumed to be false, i.e. \( \neg \text{holds}(\text{trans-committed}(z), s) \).

\[
\textbf{r10: } \text{holds}(\text{commit-noticed}(z, y), \text{res}(\text{notice-commit}(z, y), s)) \leftarrow \\
\quad \text{hold}(\text{trans-committed}(z), s)
\]

Rule \( r10 \) denotes: after transaction commit, i.e. trans-committed(\( z \)) becomes true, action notice-commit(\( z, y \)) will take place to send the MSS a commit notice.

\[
\textbf{r11: } \text{holds}(\text{abort-noticed}(z, y), \text{res}(\text{notice-abort}(z, y), s)) \leftarrow \\
\quad \neg \text{hold}(\text{trans-committed}(z), s)
\]

Rule \( r11 \) denotes: after transaction abort, i.e. trans-committed(\( z \)) is false, action notice-commit(\( z, y \)) will take place to send the MSS an abort notice.

Now, based on the discussion above, the transaction model can be formally defined as follows.
Definition: Given language $\mathcal{L}$, a knowledge transaction model on domain $D$ is a pair $\Sigma = (R, I(D))$, where $R$ is the set of knowledge transaction rules imposed above, and $I(D)$ is a finite set of initial facts and rules with respect to domain $D$. We say that a fact $\varphi$ is entailed by $\Sigma$, iff $R \cup I(D) \models \varphi$.

In the knowledge transaction model above, $I(D)$ includes initial facts and special rules with respect to domain $D$ only. For example, if a knowledge transaction is studied in an investment domain, special rules are the rules related to investment only. For instance, they could be rules for investment risk, profit and funds. Initial facts could be the initial investment status. $\varphi$ can be a formula like $[\neg \text{holds}(F, \text{res}(A_1, \text{res}(A_2, S_0)))$, which should be true in every answer set of $\Sigma$.

4.4 A Study Case for Transaction Processing in Mobile Environments

Here, a study case is given to demonstrate that the proposed logic programming based knowledge transaction processing model is applicable in practical domains in mobile environments. In this study case, an initial fact and a finite set of rules with respect to a share investment problem domain are given and specified. The given study case discusses a yes scenario for an update transaction: The MH requests an update transaction, the transaction is committed on the HS using a two-phase commit protocol. The HS sends a commit notice to the MSS, the MSS then broadcasts the commit result to the MH and the MH updates the local knowledge base accordingly. This is the yes scenario for an update transaction.

The formalized logical programming based knowledge transaction language $\mathcal{L}$ and model in this chapter are used as the restriction language and model here by replacing $x, y, z$ with MH, MSS, and HS. It is assumed there is a local knowledge base on the MH. Let $S_0$ be the initial state, and suppose two initial facts are given for MH1:

$t1: \text{holds}($registered$(\text{MSS1}, \text{MH1}), S_0) \leftarrow$
$t2: \text{holds}($profit$(\text{share1,MH1}), S_0) \leftarrow$

$t1$ means MH1 has registered in MSS1, $t2$ means MH1 has profit to invest in share1.
The following rules are introduced here for the investment problem domain:

If share1 has a high profit, low risk, then share1 can be invested from the involved MH’s point of view. We have

\[ i1: \text{holds} (\text{invest}(\text{share1}, \text{MHi}), s) \leftarrow \text{holds} (\text{profit}(\text{share1}, \text{MHi}), s), \]
\[ \text{not hold} (\text{risk}(\text{share1}, \text{MHi}), s) \]

The share1 will not be invested if the risk is high. The cancellation rule is

\[ i2: \neg \text{holds} (\text{invest}(\text{share1}, \text{MHi}), s) \leftarrow \text{hold} (\text{risk}(\text{share1}, \text{MHi}), s) \]

MH1 wants to invest share1 based on the rule \(i1\), \(i2\) and initial fact \(t2\). MH1 requests an update transaction to do the investment. According to the MH level rule \(r4\) and \(r5\), formalized in section 4.3, we have

\[ t3: \text{holds} (\text{update-requested}(\text{MH1}), \text{res}(\text{write}(\text{MH1}), s)) \leftarrow \]
\[ t4: \text{holds} (\text{trans-start}(\text{MH1}), s) \leftarrow \text{holds} (\text{locked}(\text{MH1}), \text{res}(\text{acquire-lock}(\text{MH1}), \]
\[ \text{res}(\text{write}(\text{MH1}, s))), \text{holds} (\text{update-requested}(\text{MH1}), s) \]

After MH1 requests an update transaction and the lock has been acquired, the transaction will start.

MSS1 will submit this transaction request to the HS. According to the MSS level rule \(r6\) in section 4.3, we have

\[ t5: \text{holds} (\text{trans-submitted}(\text{MSS1}), \text{res}(\text{submit}(\text{MSS1}), s)) \leftarrow \]
\[ \text{holds} (\text{locked}(\text{MH1}), s), \text{holds} (\text{update-required}(\text{MH1}), s) \]

As long as the MSS submits the transaction to the HS, the HS will start the transaction. According to the HS level rule \(r3\), formalized in section 4.3, we have

\[ t6: \text{holds} (\text{trans-start}(\text{HS}), \text{res}(\text{do-trans}(\text{HS}), s)) \leftarrow \]
\[ \text{holds} (\text{trans-submitted}(\text{MSS1}), s) \]

Then the HS will send a commit request to all involved MHs for this investment. If all the involved MHs respond with yes, based on the investment rules, the transaction is committed according to the two-phase commit protocol. The HS sends a commit
notice to the MSS. According to the HS level rules r7 and r10, formalized in section 4.3, we have

\[ t7: \text{holds(\text{trans-committed(HS), res(commit(HS), s))}} \leftarrow \text{hold(commit-agreed(HS), s), not holds(abort-agreed(HS), s), holds(trans-start(HS), s)} \]

\[ t8: \text{holds(\text{commit-noticed(HS, MSS1), res(notice-commit(HS, MSS1), s))}} \leftarrow \text{hold(trans-committed(HS), s)} \]

MSS1 then broadcasts the commit result to MH1. According to the MSS level rule r14 in section 4.3, we have

\[ t9: \text{holds(\text{commit-broadcasting(MSS1, MH1), res(broadcast(MSS1, MH1), s))}} \leftarrow \text{holds(commit-noticed(HS, MSS1), s)} \]

All involved MHs will update the local knowledge base accordingly after the MSS broadcasts the commit result to the MHs. According to the MH level rule r13 in section 4.3, we have

\[ t10: \text{holds(\text{knowledge-updated(MH), res(update-knowledge(MH), s))}} \leftarrow \text{holds(commit-broadcasting(MSS1, MH), s), holds(trans-start(MH), s)} \]

Based on the formal definition presented in section 4.3, we have \( I(D) \) to denote initial facts and rules with respect to the investment domain and we have \( R \) to denote the set of general knowledge transaction rules. In this example, \( I(D) = \{t1,t2,i1,i2\} \), \( R = \{t3,t4,t5,t6,t7,t8,t9,t10\} \), therefore we know the fact \( \varphi \) is entailed here from \( R \cup I(D) \), i.e. \( R \cup I(D) \models \varphi \), and we have \( \varphi \) as follows:

\[ \varphi = \{\text{holds(trans-committed(HS), res(commit(HS), S_0))}, \text{holds(knowledge-updated(MH), res(update-knowledge(MH), S_0))}, \text{holds(invest(share1, MH1), S_0)}\} \]

i.e. the facts that the transaction is committed on the HS, the knowledge base is updated on the MHs, and share1 is invested on MH1, are entailed in this example.
4.5 Summary

This paper proposed and formalized a rule-based knowledge transaction model for mobile environments which integrates the features of both mobile environments and intelligent agents. The formalization started with defining a knowledge transaction processing language $\mathcal{E}$, which contains necessary components for specifying knowledge transactions associated with the MH, MSS and HS. Then, a set of rules to capture features of knowledge transactions in mobile environments were imposed and specified. Lastly, the model was formally defined. By illustrating a case study, it was demonstrated that the formalized knowledge transaction model is applicable in practical domains to process knowledge transaction in mobile environments.

In comparison with previous works, the formalized knowledge transaction model has the following major advantages: (1) It can be used for knowledge transaction representation, formalization and knowledge reasoning in mobile environments. This extends the languages and models that are used for knowledge representation, reasoning and problem solving only in conventional environments, such as logic programming, extended logic programming, stable model, SMODEL, DLV and XSB [Gelfond, 1991; Baral, 1994; Gelfond, 1991; Baral 2003; Eiter, 1997; Nemela, 1996; Rao, 1997]. (2) It is knowledge-oriented and has declarative semantics inherited from logic programming so it can be used to study knowledge transaction at a high level. This is different to all the works that only deal with data transaction [Ahmad, 1995; Barbara, 1994; Imielinski, 1996; Mirghafori, 1995; Bettini, 2002; Deugo 2001]. (3) It is a formalization that can be applied to general problem domains, which is different from all approaches that suffer from a lack of formal specification and, thus, only can be ad hoc for specific systems and environments [Ahmad, 1995; Barbara, 1994; Imielinski, 1996; Mirghafori, 1995].
Conclusions

The main goal of the thesis is to study data and knowledge transaction in mobile environments. In the introduction to this thesis, several inadequacies in this field were identified. Firstly, many concepts and protocols of transaction processing need to be explored in mobile environments. Furthermore, serializability criteria and concurrency control algorithms need to be reconsidered and, perhaps, modified or completely recreated to be applied to mobile applications. Moreover, knowledge representation and knowledge transaction are novel topics for the research of mobile computing.

This thesis has rechecked and redefined a number of important notions and protocols in mobile environments including the concepts of transaction, transaction history, equivalent history, history serial, serialization graph, notion of serializability, and locking protocols. This investigation is different from works in [Kistler, 1993; Mummert, 1996; Demers, 1994; Joseph, 1997; Reiher, 1994] in that they concentrate on mobile computing system modeling and implementation. A conclusion reached from this study is that classical notion of serializability still holds in mobile environments in a modified form. This differs with Ahamad and Smith’s work [Ahamad, 1995] wherein they claimed that strong consistency such as serializability can not be provided in systems where clients may become temporarily disconnected. The structure of mobile transaction was examined and shown that transactions executed by the mobile hosts always take on a form consisting of several blocks. Furthermore, a criterion for a serial history was given based on serialization graph, which is the analogue of the acyclic serial graph criterion in conventional environments. It was pointed out that self-intersection wormhole always indicates a violation of local causality so it is not allowed in a serialization graph of a serial history in mobile environments. The wormhole theorem and locking theorem were produced and proven in mobile environments as the upshot of discussion. These
explored notions and protocols are fundamental to the study of transaction processing in mobile environments and little previous work has been contributed in this area.

This thesis formally defined a relaxed correctness criterion for serializability – epsilon serializability in mobile environments. A relaxed concurrency control method for mobility support was proposed so that rapidly changing data objects can be processed with limited inconsistency for mobile applications, yielding improved transaction throughput. This proposed concurrency control method is applicable to both discrete mobility and continuous mobility, and was implemented by both the two-phase locking method and the optimistic divergence control method. A few global algorithms were adopted to maintain global consistency for distributed transactions. In comparison with previous concurrency control approaches, our method can provide concurrency control for mobile transactions, which is different from all the concurrency control algorithms that are suitable only for the conventional environments [Bober, 1992; Breitbart, 1991; Elmagarmid, 1990; Georgakopoulos, 1991; Herlihy, 1990; Ho, 1997; Garcia-Molina, 1982]. This method is application independent, which is different from the work in [Joseph, 1999; Katz, 1994; Davies, 1994; Kaashoek, 1994; Terry, 1995; Kistler, 1992; Kummar, 1994; Mummert, 1995; Reiher, 1994; Huston, 1995] that is limited to specific applications or problem domains. The proposed method uses epsilon serializability as a weaker correctness criterion and tolerates the bounded inconsistency, and thus gains improved transaction performance. It is particularly desirable for one class of application that involves querying for rapidly changing data, and therefore has a lot of real applications.

As a significant contribution, this thesis addressed the accounts of knowledge transaction in the field of mobile computing in addition to data transaction. Firstly, A rule based knowledge transaction model was presented and formalized in mobile environments, which integrates the features of both mobile environments and intelligent agents. This model can be used to study knowledge representation and knowledge transaction in mobile environments. With this formalization, a knowledge transaction processing language was presented to specify components of knowledge transactions associated with MH, MSS and HS three levels and a set of rules
were imposed to capture features of knowledge transaction in mobile environments. Additionally, a Mobile Logic Programming Multi-Agent System (MLPMAS) was modeled and formalized for mobile environments. Such a system consists of a number of agents connected via wire or wireless communication channels. Agents communicate with each other by passing answer sets and interactions between these agents were also modeled. Based on this model, knowledge transaction can be studied in this kind of mobile multi-agent systems. In the formalization above, extended logic programming was employed and justified to be a well-suited method to investigate knowledge representation and knowledge transaction in mobile environments. Mobile Semantics was also given to classical negation \( \neg \) and negation-as-failure \textit{not}. A few study cases were illustrated to demonstrate that the formalized knowledge transaction model is applicable in practical domains and knowledge transactions can be processed in MLPMAS systems.

With respect to previous works, the formalized knowledge transaction model and mobile logic programming multi-agent system in this thesis can be used to study knowledge representation, reasoning and knowledge transaction in mobile environments. This extends classical languages and models developed for knowledge representation, reasoning and problem solving in conventional environments [Baral 2003]. Our knowledge transaction model is knowledge oriented which is different from the systems and proposals that only can deal with data transaction [Ahamad, 1995; Barbara, 1994; Imielinski, 1996; Mirghafori, 1995; Bettini, 2002; Deugo 2001; Bhalla, 2003; Dang, 2003]. This model is also a formalization that can be applied in general problem domains, which differs from all approaches suffering from a lack of formal specification and thus only can be ad hoc for specific systems and environments [Ahamad, 1995; Barbara, 1994; Imielinski, 1996; Mirghafori, 1995; Krishnakumar, 1996]. Our MLPMAS system can specify details of knowledge transaction, input/output information and knowledge base using knowledge rules. This is different from most works on mobile agent systems in that they are not knowledge oriented in their processing, for example, Telescript [White, 1996], Aglets [Lange, 1998], Voyager [Glass, 1998], Mole [Baumann, 1998] and KLAVA [Bettini, 2002; Deugo 2001, Lange, 1998b]. Our very early investigation into formalizing knowledge transaction models and multi-agent systems in mobile
environments will provide a foundation for studying knowledge bases and intelligent agents in mobile systems and will help us to improve intelligent systems and mobile systems greatly.

As suggestions for future research, further study can be conducted in the following areas:

- Based on concepts of transaction and serializability discussed in Chapter 2 for mobile environments, further research can be carried out on the study of commit and abort protocols, rollback protocols and log protocols in mobile environments. They are very important notions and protocols for transaction implementation in the field of mobile computing.

- As pointed in Chapter 3, our proposed relaxed concurrency control method for mobile transaction is only limited in homogeneous distributed system. Investigation on the relaxed concurrency control methods suitable for heterogeneous distributed system in mobile environments can be an interesting research subject.

- Formalization of knowledge transaction model and multi-agent system framework in mobile environments in this thesis is a good starting point from which to investigate knowledge base and intelligent agents in mobile systems, and can lead to more research. Implementing and bringing this formalization to a real-world system will be a very meaningful future work. Current best known implementations of logic programming system in conventional environments, such as Prolog [Sterling, 1986], XSB [Chen, 1995], SMODEL [Niemela, 1997], dlv [Eiter, 2000], DeRes [Truszczynski, 1999], Ccalc [Lifschitz, 1995] and LDL++ [Wang, 2000], can form the basis for implementing our knowledge transaction and multi-agent models in mobile environments. In addition, knowledge reasoning and declarative problem solving in mobile environments are very challenging research topics that have significant applications in practice.
Bibliography


<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Bibilography</th>
<th></th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
</table>


Publications

Journal Papers:


2. Jianwen Chen and Yan Zhang, “Formalizing a Multi-Agent System for Mobile Environments”, to be submitted to IEEE Transactions on Knowledge and Data Engineering, 2004.

Conference Papers:


4. Jianwen Chen and Yan Zhang, “A Rule Based Knowledge Transaction Model in Mobile Environments”, Proceeding of 2nd IASTED International


Data and Knowledge Transaction in Mobile Environments

Jianwen Chen

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy (Science) – Computing and Information Technology

November, 2004

© Jianwen Chen 2004
This is to certify that the thesis is my own original work except where otherwise indicated. No part of this thesis has been submitted as part of any other degree.

Jianwen Chen
November 2004
Acknowledgements

I would like to thank the individuals who have helped me in my Ph.D. research program at both the University of Western Sydney and IBM Australia. In particular, I take great pleasure in thanking my supervisors Associate Professor Yan Zhang and Dr. Zhuhan Jiang at the University of Western Sydney for their enthusiastic supervision. I am greatly indebted to Associate Professor Yan Zhang for his invaluable guidance, encouragement, and detailed suggestions on all phases of my research, as well as his tremendous understanding and backup. I am also very grateful to Dr. Wai K. Chan for his enthusiastic guidance and support before his leaving the University of Western Sydney. I extend my special thanks to my manager Raquel Lesaca at IBM Australia for her great understanding and help.

I thank the University of Western Sydney for offering me an Overseas Postgraduate Research Reward which has made my Ph.D. research a reality. I also thank IBM Australia for the backing provided to me during my education.

I feel everlasting gratitude toward my husband, Tao Wei and my daughter, Jessica, for their tremendous support of my pursuit of the Ph.D. My research would not have been possible without their understanding. I am also grateful to my parents and parents-in-law for their continual encouragement and assistance.
Abstract

Advances in wireless networking technology have engendered a new paradigm of computing, called mobile computing, in which users carrying portable devices have access to a shared infrastructure independent of their physical location. Mobile computing has matured rapidly as a field of computer science. In environments of mobile computing, the mobility and disconnection of portable computing devices introduce many new and challenging problems that have never been encountered in conventional computer networks. New research issues combine different areas of computer science: networking, operating systems, data and knowledge management, and databases.

This thesis studies data and knowledge transaction in mobile environments. Transaction processing in mobile environments is a very meaningful and challenging research area, and most of the recent research works have been concentrated on mobile computing system modeling and implementation. Many important fundamental concepts and protocols of transaction processing have yet been thoroughly explored, thus they remain as open questions in the mobile computing field. To study transaction processing at the fundamental and theoretical level in mobile environments, a range of classical notions and protocols of transaction processing are rechecked and redefined in this thesis, and form the foundation for studying transaction processing in mobile environments. A criterion for a mobile serial history is given, and as the result of discussion, two new concurrency theorems are proved in mobile environments.

Serializability is too strong as a correctness criterion for mobile transaction processing due to the unique characteristics of mobile environments. The relaxed serializability criterion and concurrency control method are more desirable, and required, in mobile environments to improve performance or availability. This thesis
investigates a relaxed serializability - epsilon serializability for mobile transactions and proposes a relaxed concurrency control method to process rapidly changing data objects for mobile applications. The approach tolerates bounded inconsistency and therefore improved transaction throughput is gained.

In addition to data transaction, this thesis explores knowledge transaction in mobile environments. Currently, there is a separation between the intelligent agent community on one side, and the mobile system community on the other side. Most current transaction processing research concentrates on data rather than knowledge transaction in the mobile computing area. A lot of well-known mobile agent systems are not knowledge oriented. The languages and models developed for knowledge representation, reasoning and problem solving, and for intelligent agent systems, are discussed only in stationary environments. In mobile environments, no formal investigation has been conducted so far on the issue of knowledge transaction, no knowledge transaction model has been formalized, and no framework of mobile multi-agent systems has been modeled. They are all open questions in the field of mobile computing.

To study knowledge transaction in mobile environments, this thesis presents and formalizes a knowledge transaction language and model for use in mobile computing environments. Logic programming is used as a mathematical tool and specification method. The formalized knowledge transaction model can be used for knowledge transaction representation, formalization and knowledge reasoning in mobile environments.

Finally, the thesis further formalizes a framework/model for a mobile logic programming multi-agent system which can be used to study knowledge transaction in multi-agent systems in mobile environments. The formalization is knowledge-oriented and has declarative semantics. The details of knowledge transaction, input/output information, and the knowledge base itself are specified through various logic program rules.
It is believed that the investigation into the knowledge transactions and intelligent agents in mobile environments is critical because it will help us to significantly improve the current development of mobile systems and intelligent agent systems. Formalization of the knowledge transaction model and the multi-agent system framework in this thesis is a very early effort towards a formal study of knowledge base and intelligent agents in mobile environments. This work will provide a foundation for the formal specification and development of real-world mobile software systems, in the same way as traditional software systems have developed.
## Contents

Acknowledgements .......................................................... III
Abstract ........................................................................... IV
Abbreviations .................................................................. X

1 Introduction ..................................................................... 1
1.1 Literature Review ............................................................... 1
1.2 Thesis Contributions ......................................................... 7
1.3 Background Knowledge ..................................................... 8
1.3.1 Mobile Computing ......................................................... 8
1.3.2 The Basics of Transaction Processing ......................... 10
1.3.3 Multi-Agent System and Knowledge Representation .... 13
1.4 Thesis Structure ............................................................... 16

2 Transaction and Serializability in Mobile Environments .... 18
2.1 The Mobile Environment Model ......................................... 18
2.2 Transaction Notions in Mobile Environments ..................... 20
2.3 Serializability in Mobile Environments .............................. 24
2.3.1 Serializability in Conventional Environments ............... 25
2.3.2 Serializability Notions in Mobile Environments .......... 29
2.4 Serializability Theorems in Mobile Environments ............ 32
2.4.1 Wormhole Theorem .................................................... 32
2.4.2 Locking Theorem ....................................................... 33
2.5 Summary ...................................................................... 35

3 Concurrency Control for Mobility Support ....................... 36
3.1 Concurrency Control Overview ......................................... 36
3.1.1 Relaxed Serializability Criteria and Concurrency Control .......... 36
3.1.2 Epsilon Serializability in Conventional Environments .......... 38
3.1.3 ESR Divergence Control Algorithms ............................ 40
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MH</td>
<td>Mobile Host</td>
</tr>
<tr>
<td>MSS</td>
<td>Mobile Support Station</td>
</tr>
<tr>
<td>HS</td>
<td>Home Server</td>
</tr>
<tr>
<td>TP</td>
<td>Transaction Processing</td>
</tr>
<tr>
<td>DAI</td>
<td>Distributed Artificial Intelligence</td>
</tr>
<tr>
<td>AI</td>
<td>Artificial Intelligent</td>
</tr>
<tr>
<td>SG</td>
<td>Serialization Graph</td>
</tr>
<tr>
<td>SR</td>
<td>Serializability</td>
</tr>
<tr>
<td>ESR</td>
<td>Epsilon Serializability</td>
</tr>
<tr>
<td>DC</td>
<td>Divergence Control</td>
</tr>
<tr>
<td>ET</td>
<td>Epsilon Transaction</td>
</tr>
<tr>
<td>CC</td>
<td>Concurrency Control</td>
</tr>
<tr>
<td>S2PLDDC</td>
<td>Strict 2-Phase Locking Distributed Divergence Control</td>
</tr>
<tr>
<td>ODDC</td>
<td>Optimistic Distributed Divergence Control</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Process Call</td>
</tr>
<tr>
<td>CM</td>
<td>Commit Manager</td>
</tr>
<tr>
<td>API</td>
<td>Application Program Interfaces</td>
</tr>
<tr>
<td>RPM</td>
<td>Radio Packet Modem</td>
</tr>
<tr>
<td>RNG</td>
<td>Radio Network Gateways</td>
</tr>
<tr>
<td>SCR</td>
<td>Standard Context Router</td>
</tr>
<tr>
<td>RNC</td>
<td>Radio Network Controller</td>
</tr>
<tr>
<td>KB</td>
<td>Knowledge Base</td>
</tr>
<tr>
<td>MLPMAS</td>
<td>Mobile Logic Programming Multi-Agent System</td>
</tr>
</tbody>
</table>