TRANSACTIONAL ACTORS
IN
COOPERATIVE INFORMATION SYSTEMS

Statement

I hereby state that this thesis contains only my own original work except where explicit reference has been made to the work of others.

Mostafa S. Haghjoo

A THESIS
SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
OF THE AUSTRALIAN NATIONAL UNIVERSITY

© Mostafa S. Haghjoo
March 1995
First of all, I would like to thank my primary research supervisor, Professor Mike P. Papazoglou, for so many reasons it is impossible to list them all here. However, I must thank him for his constant consideration and guidance, continuous support, constructive discussions and valuable comments, particularly in the last year. I have learned lessons from him that will serve me throughout my life.

I would like to thank my supervisor Professor Heinz W. Schmidt for his close supervision and timely advice during the design and implementation of the scripting language. Constructive ideas were very helpful in presenting the formal semantics of the proposed model.

I would like to thank my supervisor Dr. Victoria Peterson for her constant support and encouragement, and Dr. Athman Bouguettaya of Queensland University of Technology, and Dr. Anne Ngo of University of New South Wales for their time and valuable comments. Their suggestions caused the thesis to be improved in several respects, including presentation and clarity.

Mostafa S. Haghjoo
First of all I would like to thank my primary research supervisor, Professor Mike P. Papazoglou, for so many reasons it is impossible to list them all here. However I must thank him for his constant consideration and guidance, continuous support, constructive discussions and valuable comments, particularly in the last year. I have learned lessons from him that will serve me throughout my life.

I would like to thank my supervisor Professor Heinz W. Schmidt for his close supervision and timely advice during the design and implementation of the scripting language. Also his constructive ideas were very helpful in presenting the formal semantics of the proposed model.

I would like to thank my supervisor Dr. Victoria Peterson for her constant support and valuable comments throughout the writing of this thesis.

Finally, I would like to thank the reviewers of my thesis, Professor Bala Srinivasan of Monash University, Dr Athman Bouguettaya of Queensland University of Technology, and Dr Anne Ngu of University of New South Wales for their time and valuable comments. Their suggestions caused the thesis to be improved in several ways, including organization, presentation, and clarity.
Abstract

Transaction management in advanced distributed information systems is a very important issue under research scrutiny with many technical and open problems. Most of the research and development activities use conventional database technology to address this important issue. The transaction model presented in this thesis combines attractive properties of the actor model of computation with advanced database transaction concepts in an object-oriented environment to address transactional necessities of cooperative information systems. The novel notion of *transaction tree* in our model includes subtransactions as well as a rich collection of decision making, chronological ordering, and communication and synchronization constructs for them. Advanced concepts such as blocking/non-blocking synchronization, vital and non-vital subtransactions, contingency transactions, temporal and value dependencies, and delegation are supported. Compensatable subtransactions are distinguished and *early commit* is accomplished in order to release resources and facilitate cooperative as well as long-duration transactions. Automatic *cancel* procedures are provided to logically undo the effects of such commits if the global transaction fails.

The complexity and semantics-orientation of advanced database applications is our main motivation to design and implement a high-level *scripting language* for the proposed transaction model. Database programming can gain in performance and problem-orientation if the semantic dependencies between transactions can be expressed directly. Simple and flexible mechanisms are provided for advanced users to query the databases, program their transactions accordingly, and accept weak forms of semantic coherence that allows for more concurrency. The transaction model is grafted onto the concurrent object-oriented programming language *Sather* developed at UC Berkeley which has a nice high-level syntax, supports advanced object-oriented concepts, and aims toward performance and reusability. We have augmented the language with distributed programming facilities and various types of message passing routines as well as advanced transactions management constructs.

The thesis is organized in three parts. The first part introduces the problem,
reviews state of the art, and presents the transaction model. The second part describes the scripting language and talks about implementation details. The third part presents the formal semantics of the transaction model using mathematical notations and concludes the thesis.

Contents

I. Transaction Management in Cooperative Information Systems

1. The Proposed Model and Language
   1.1 Introduction and Motivation
   1.2 Contribution of This Work
      1.2.1 A Transaction Model
      1.2.2 A Scripting Language

2. Related Work
   2.1 Database Transactions
      2.1.1 Concurrency Control
      2.1.2 Recovery
   2.2 Advanced Transaction Models
      2.2.1 Sagas
      2.2.2 Multilevel Transactions
      2.2.3 Split-Transactions
      2.2.4 S Transactions
      2.2.5 Flex Transactions
   2.2.6 DOM Transactions
   2.3 Transaction Languages and Systems
      2.3.1 Angus and Tine
      2.3.2 Avalon/C++
      2.3.3 Interactions and TaSL
      2.3.4 STDL
      2.3.5 VPL
      2.3.6 Arjuna
Contents

1 Transaction Management in Cooperative Information Systems

1 The Proposed Model and Language

1.1 Introduction and Motivation

1.2 Contribution of This Work

1.2.1 A Transaction Model

1.2.2 A Scripting Language

2 Related Work

2.1 Database Transactions

2.1.1 Concurrency Control

2.1.2 Recovery

2.2 Advanced Transaction Models

2.2.1 Sagas

2.2.2 Multilevel Transactions

2.2.3 Split-Transactions

2.2.4 S Transactions

2.2.5 Flex Transactions

2.2.6 DOM Transactions

2.3 Transaction Languages and Systems

2.3.1 Argus and Thor

2.3.2 Avalon/C++

2.3.3 Interactions and TaSL

2.3.4 STDL

2.3.5 VPL

2.3.6 Arjuna

viii
2.4 The Actor Model of Computation ........................................... 29
2.5 Conclusion ............................................................................ 31

3 TractorS: A Semantic-based Transaction Model 33
3.1 An Architecture for Cooperative Information Systems ............ 33
3.2 A Transaction Model for Cooperative Information Systems .... 36
3.2.1 Distributed and Multidatabase Transactions ..................... 36
3.2.2 Actors in Transaction Management ................................. 40
3.2.3 Scheduling and Synchronizing Transactions .................... 42
3.2.4 Case Study: Home-loan Request Transaction .................... 45
3.2.5 Transaction Termination ................................................. 49
3.2.6 Delegation ..................................................................... 51
3.2.7 Fully Decomposed Transaction Trees ............................. 53
3.2.8 Global Decomposed Transaction Trees ......................... 55

II The Scripting Language for TractorS 60
4 Linguistic Facilities ................................................................. 61
4.1 Sather Classes ..................................................................... 62
4.2 TractorS Library ................................................................. 66
4.2.1 Class Hierarchy ............................................................... 66
4.2.2 Unsafe_commit and Cancel ............................................ 70
4.2.3 Delegation ..................................................................... 71
4.3 End-user Interaction ............................................................. 73
4.4 Programmer Interaction ...................................................... 74
4.4.1 Conditional Subtransactions .......................................... 78
4.4.2 Site Specification ............................................................ 80
4.4.3 Replicated Subtransactions ............................................ 81
4.4.4 Serial Argument Dependency ....................................... 82
4.4.5 Commit Dependency ....................................................... 83
4.4.6 Value Dependency .......................................................... 84
4.4.7 Concurrent Subtransactions .......................................... 86
4.4.8 Non-vital Subtransactions ............................................. 87
4.4.9 Compensation ............................................................... 88
### 4.4.10 Setting Timeout Periods

### 4.5 Programming the Case Study

#### 4.5.1 Schedulers

#### 4.5.2 Atomic Transactions

### 5 Implementation

#### 5.1 Implementation Rationale

##### 5.1.1 The Host Language

##### 5.1.2 Interprocess Communication

#### 5.2 Communication and Synchronization in TractorS

##### 5.2.1 The Sather-PVM Interface

##### 5.2.2 Example: a Distributed Program

#### 5.3 Details of TractorS Library

##### 5.3.1 Atomic Transactions

##### 5.3.2 Schedulers

##### 5.3.3 Delegation

##### 5.3.4 Transactional Events

#### 5.4 Linking TractorS to the External World

##### 5.4.1 OBST: An Object Oriented Persistent Storage System

##### 5.4.2 ORACLE: A Relational Database Management System

#### 5.5 Achievements Gained Through Building a Prototype

### III TractorS Formalization

### 6 Formal Semantics

#### 6.1 A Short Summary of ACTA

##### 6.1.1 Characterization of Atomic Transactions

#### 6.2 Schedulers

#### 6.3 Scheduler Dependencies

#### 6.4 Transaction Completion and Failure

#### 6.5 Correctness of Schedulers

#### 6.6 Compensation

#### 6.7 Delegation

#### 6.8 Concurrent Transactions
# List of Tables

4.1 A sample of TractorS library classes ............................................. 68
5.1 Most important PVM interface routines .......................................... 109

3.1 Global transaction decomposition in an information agent network ........ 35
3.2 Transaction tree for the home-loan request transaction ...................... 46
3.3 A multi-level scheduling tree .......................................................... 54
4.1 Class hierarchy of TractorS library ................................................ 67
5.1 A pictorial view of the Sather-PVM interface .................................. 107
# List of Figures

1.1 A transaction tree for a hypothetical transaction in TractorS . . . 10  
2.1 Conceptual representation of an actor . . . . . . . . . . . . . . . . . . . 30  
3.1 Global transaction decomposition in an information agent network 35  
3.2 Transaction tree for the home-loan request transaction . . . . . . . . 46  
3.3 A multi-level scheduling tree . . . . . . . . . . . . . . . . . . . . . . . 54  
4.1 Class hierarchy of TractorS library . . . . . . . . . . . . . . . . . . . . . 67  
5.1 A pictorial view of the Sather-PVM interface . . . . . . . . . . . . . . 107
Part I

Transaction Management in Cooperative Information Systems

Transaction management in distributed information systems is addressed mostly by conventional database approaches. Our goal is to present a novel transaction information system architecture that combines advanced database concepts, as well as its linguistic support. It is our view that an appropriate combination of the traditional closed nested transaction model suggested by Moss [1], and open nested transaction mechanisms such as sagas [2], split transactions [3] and flexible transactions [4] would provide the advanced transaction concepts required for cooperative information systems. Our proposed model named TAC-actor System combines attractive properties of the actor model of computation [5, 6] with advanced database transaction management concepts in an object-oriented environment to address the processes of reasoning about the need for cooperation between the disparate information sources in a distributed information network.

With the current advances in communication and networking technology many organizations have shown considerable interest in integrating and consolidating their physically dispersed data/information resources. Accordingly, a level of replacing information management software is deemed necessary for access to both information residing at disparate information processing sites in a common communication network. Such a configuration is known as a Cooperative Information System (CIS)
Chapter 1

The Proposed Model and Language

1.1 Introduction and Motivation

Transaction management in distributed information systems is addressed mostly by conventional database concepts such as atomic transactions. Our goal is to present a novel transaction management model for cooperative information systems which is based on a cross-fertilization of knowledge base and database concepts, as well as its linguistic support. It is our view that an appropriate combination of the traditional closed nested transaction model suggested by Moss [1], and open nested transaction mechanisms such as sagas [2], split transactions [3] and flexible transactions [4] would provide the advanced transaction concepts required for cooperative information systems. Our proposed model named TractorS (for Transactional-actor System) combines attractive properties of the actor model of computation [5], [6] with advanced database transaction management concepts in an object-oriented environment to address the process of reasoning about the need for cooperation between the disparate information sources in a distributed information network.

With the current advances in communication and networking technology many organizations have shown considerable interest in integrating and consolidating their physically dispersed data/information resources. Accordingly a level of mediating information management software is deemed necessary for access to data/information residing at disparate information processing sites in a common communication network. Such a configurations is known as a Cooperative Information System (CIS).
The purpose of a CIS is to provide efficient and robust facilities for distributed transaction management, i.e. processing of what is known as *global requests*. The essential requirement of distributed transaction processing has been atomicity of global transactions so that the corporate information system is always kept at a consistent state. Recently, the object-oriented paradigm has been suggested as a major contributor for materializing the mediating software and thus integrating and coordinating the disparate information sources in a CIS [7].

In [8] knowledge based processing attachment to the object-oriented paradigm is proposed for transforming the relatively passive component information systems in the CIS-network into *information agents*. In such a network, information agents interact cooperatively to solve a multitude of complicated information-intensive problems which can only be solved by selectively fusing and combining data/information from diverse problem solving sites in the network. Each information agent must be able to reason about the need for cooperation and to understand local and global knowledge to locate the other information agents involved in the processing of a global request.

A typical CIS environment involves large numbers of information systems distributed over complex networks. Examples of such systems range from electronic highway to conventional banking systems. Such systems will have access to large amounts of information and computing services and will involve human interactions. Information and services are made available in heterogeneous forms. Demand for more efficient use of resources increases and concurrent computation becomes more and more necessary. The above issues pose serious technological challenges with regard to the type of global transaction processing which is viewed as some form of message passing activity between communicating/cooperating information agents. Our project proposes a model for transaction management in CIS [9].

Transaction tree in conventional nested transactions comprises of subtransactions with a fixed execution order. The novel notion of transaction tree in TractorS includes subtransactions as well as a rich collection of decision making and chronological ordering constructs for them. In the conventional nested transaction models it is impossible to indicate temporal dependencies between activities within the context of a top-level transaction. This forces users to split a
single task into several top-level transactions and then submit them either serially or concurrently. In a CIS environment, autonomy of sites, long-duration of transactions, and different types of failures make it difficult to control and synchronize such semantically dependent activities appearing as independent top-level transactions. Imagine a task with a number of subtasks which need to cooperate in order to proceed. The isolation property of conventional transactions make such a task impossible. TractorS provides flexible constructs to easily combine semantically dependent activities into a single top-level transaction. It facilitates concurrency by supporting parallel tasks and flow of information between them. Furthermore, it provides automatic control and synchronization of subtasks.

Several currently disjoint technologies are involved in this research. The concepts draw on many technology areas - database systems, the actor model of computation, distributed computing, object-oriented programming languages and databases, interprocess communication systems, and software and knowledge engineering. Our proposed architecture for CIS results from the appropriate integration of such technologies. The proposed transaction management model results mostly from advanced database transaction models and the actor model of computation. The linguistic support results from concurrent object-oriented programming languages and transaction management in object-oriented database systems.

The thesis is organized in three parts:

1. part one consists of three chapters. The first chapter introduces the project and the second one reviews state of the art. The third chapter introduces the transaction model and the underlying environment for cooperative information systems. It also reasons about using the actor model of computation in transaction management.

2. part two describes the scripting language for TractorS in two chapters. The first one introduces the facilities and the way to use them. A comprehensive case study introduced in part one is programmed in this chapter. The second chapter of this part rationalizes the implementation choices and talks about implementation details. It describes details of TractorS library and the distributed programming facilities being developed as part of the
scripting language. It also shows how TractorS is linked to the information systems in the external world.

3. Part three presents the transaction model in a mathematical form in order to facilitate understanding its exact semantics. Readers may refer to this chapter at any point for exact definitions and proofs. It also concludes the thesis and talks about further research.

1.2 Contribution of This Work

The research relating to TractorS has its origin on research activities and projects conducted both on long-lived transactions and on closed and open nested transaction models (in terms of their termination properties) for multidatabase systems (MDBSs) [10], [11]. The main contribution of the thesis is providing an advanced transaction model together with its underlying scripting language which adopts the attractive properties of the actor model of computation in transaction management paradigm to address the requirements of cooperative information systems. The model uses several of the notions suggested by the MDBS transaction models and enriches them with concepts of its own. It supports advanced transaction concepts such as open and closed nesting, blocking/non-blocking synchronization, vital and non-vital subtransactions, early commit and compensation, mixed transactions (compensatable and non-compensatable), contingency transactions, temporal and value dependencies between subtransactions, and delegation as clarified later in the thesis.

In contrast to other models, TractorS is not only an extension of the nested transaction concept, it also suggests the use of a highly structured linguistic approach to facilitate transaction programming. The complexity and semantics-orientation of advanced database applications is our main motivation to provide a scripting language for TractorS with features directly oriented to concurrent database programming. Such applications can also gain in performance if database programmers and advanced users can express semantic dependencies between transactions directly or accept weaker forms of semantic coherence allowing more concurrency.

The transaction model is grafted onto the concurrent object-oriented pro-
gramming language Sather [12], [13] developed at University of California at Berkeley. The language has an attractive high-level syntax. It is strongly typed and allows specification of class invariants and other semantic integrity conditions which makes it more suitable for database programming than many other object-oriented languages. It also supports parallel programming by providing advanced monitor and locking facilities [14]. The main goals in Sather are performance and reusability. Sather has a performance comparable to C++ [13]. Reusability has become increasingly important. There are different systems with similar transaction management needs, so well designed classes can be reused over and over to serve similar purposes. Distributed programming and message passing facilities are added to the language by making an interface to the well known distributed interprocess communication package PVM [15]. Also global transactions management functionality is added as a set of classes to support TractorS.

1.2.1 A Transaction Model

TractorS provides flexible constructs to easily combine semantically dependent activities into a single top-level transaction. It facilitates concurrency by supporting parallel tasks and flow of information between them. Dependencies such as chronological, argument, value and commit dependency between subtransactions are supported. Furthermore, it provides automatic control and synchronization of subtasks. The model is a modular actor system which manage advanced and complex transactions in CIS environments.

Transaction decomposition in TractorS is a dynamic process which partially takes place in parallel with other activities of the transaction. After an initial decomposition attempt, the resulted transaction tree may contain two logical levels of activities: (i) Ones that the home site knows how to handle or has recognized some close acquaintances which are definitely able to do so. This type of activity may still involve several subtransactions which need to communicate in different ways to solve the problem. (ii) Ones that neither the home site nor any of its close acquaintances know how to handle, but some foreign acquaintances are recognized which model that problem domain and may be able to handle the task. Such activities are delegated to one or more such sites selected by the user.
Chapter 1. The Proposed Model and Language

The delegated subtransaction may result into any type of activities at the target sites.

TractorS focuses on *how to build concurrent histories*. *Actors* are utilized to access data items, make decisions, enforce orders, control concurrency, etc. There are two types of actors in TractorS *transaction tree*:

1. **base actors**, appearing as *leaves* of the tree, are the *only ones* in direct contact with components of CIS (databases, file systems, knowledge bases). Other types of actors are forbidden from such direct contacts. Base actors send and receive messages, change state, and possibly create other actors in order to manage *atomic transactions* and communicate the results. The script part of base actors carries out *all* primitive transaction management activities. In TractorS these activities are:

   - **Retrieve** which retrieves state of an object at a site.
   - **Update** which updates state of an object at a site.
   - **S-lock** which acquires a shared lock on an object at a site.
   - **X-lock** which acquires an exclusive lock on an object at a site.
   - **Unlock** which releases the locks on an object at a site.
   - **Safe-commit** which commits operations on an object at a site and Unlocks it.
   - **Undo** which aborts operations on an object at a site, restores its state to the one that existed prior to the start of the related transaction, and Unlocks it.
   - **Unsafe-commit** which commits operations on an object at a site and Unlocks it, but also updates the *cancel-log* for a probable *Cancel* procedure. The operation is restricted to certain *compensatable* subtransactions characterized by the application.
   - **Cancel** which logically removes effects of operations on an object at a site, without necessarily restoring its state to the one that existed prior to the start of the related transaction, and Unlocks it.

---

1Upgrading and downgrading locks are extensions of X-lock and S-lock. We avoid defining them as main activities. The S-lock and X-lock actors may create sub-actors *Upgrade* and *Downgrade* to handle the activities.
prior to the start of the related transaction. The operation is restricted to the cases where a subtransaction is committed unsafely and its responsible top-level transaction is undone.

Except for the last two items which are novel in TractorS, other items are similar to the ones defined in [16] (*Undo* and *Safe-commit* are defined as *Abort* and *Commit* respectively there to preserve the general meaning of the terms). Unsafe-commit offers two types of flexibility: (i) information sharing between concurrent subtransactions which cannot proceed alone or execute sequentially in nature; for instance programs in CASE environments need partial results of each other to complete a software design. (ii) early release of data items, locked by long transactions, to other transactions which have to wait unnecessarily otherwise.

Keeping in mind that real world objects are not necessarily actors, TractorS provides the flexibility to support other types of objects as leaves of its transaction trees, i.e., any type of objects with atomic transaction properties may substitute base actors. This is important due to the autonomy of components of the system, i.e., the fact that the transaction management services provided by component information systems cannot be changed. Any sort of transaction manager which communicates for global synchronization serves our purpose.

2. **meta actors**, the more intelligent types of actors which create and govern base actors in a flexible open-nested transaction framework and combine results of atomic transactions to solve global problems. Currently they are of the following two types but the list is open-ended for possible future evolutions.

(a) **schedulers** which support all types of dependencies such as chronological, argument, value and commit dependency between their subtransactions. They are of four types. In *serial* and *serial-alternative* schedulers subtransactions are activated one at a time. In the former all subtransactions should commit in order to make the parent commit. In the latter however, commitment of one is sufficient and later ones will not get activated. In *parallel* and *parallel-alternative* sched-
ulers subtransactions are submitted in parallel. Here also the first type needs all subtransactions commit in order to make the parent commit, but in the second one commitment of one is sufficient. We refer to serial-alternative and parallel alternative schedulers as choice schedulers too. They follow the "one is enough" semantics.

(b) delegators which delegate subtransactions as well as their responsibility to other sites. A subtransaction may go through such process several times. In this way several sites my get involved until the one which is actually able to process the transaction is found. For this reason the delegation process also specifies the receiver of the result whereby the intermediate sites get free of such (irrelevant) interactions.

Figure 1.1 depicts a transaction tree for a hypothetical transaction. The shape of the tree nodes indicates their type as shown in the legend (for instance a rectangle represents a serial scheduler). There are three types of arcs in the transaction tree: (i) solid arcs for decomposition/nesting relationship similar to the arcs in conventional nested transaction trees. (ii) dotted arcs for serial argument dependency. The source of such an arc is the producer and the destination needs the produced values before being able to start executing. This type of arc is always labeled with the name(s) of the argument(s) being produced and can only be drawn between subtransactions of the same parent. It is only applicable to serial schedulers because they are the only type of scheduler having subtransactions running one after the other. (iii) dashed arcs for value dependency or commit dependency. Only dashed arcs are allowed to cut across inside the tree. The source of such an arc "signals" its completion. The destination "waits" for the completion of the source. Without a label, this type of arc represents a commit dependency that can only be explained semantically. A label on a dashed arc specifies that in addition to the temporal dependency there is information which needs to be passed between the subtransactions. The destination of the arc is the consumer of the information and will block until the required information is produced.

We distinguish the top-most actor in a transaction tree, i.e, the one with no parent, as the top-actor. A top-actor is responsible for all activities going on in the transaction tree and may pass parts of the responsibility to delegators.
Figure 1.1: A transaction tree for a hypothetical transaction in TractorS

top-actor is a scheduler due to the following reasons:

- It cannot be a base actor because the whole transaction would then be a single local atomic transaction with no distribution involved (a trivial case).

- delegation is not a stand-alone action. A node (parent) is needed to decide on necessity of such an activity and to initiate it.

1.2.2 A Scripting Language

Transaction models and their linguistic needs are well understood in traditional database management systems. Due to the richness of object oriented
model which presents many alternatives, there is no commonly accepted data model formalization. Making specific decisions about what features to support in a model should be done very carefully. Here we summarize what our scripting language provides to support advanced transactions in CIS:

- Objects are considered as instances of a abstract data types. The visible components of objects are described in two basic abstraction levels: (i) at each site a number of classes are defined which provide methods (routines) as their interface. Such methods have facilities for execution as well as failure atomicity. (ii) a site provides remote services in the context of specific classes which encapsulates features of other local classes for such a purpose and makes remote calls robust and easy. Our model requires behaviors of objects but not their internal structures.

- strong typing is supported in the sense that each object is related to one and only one type. In this way, the database schema is well defined and transaction management is more robust and clear. Multiple typing and sharing among objects is simulated by subtype/supertype and inheritance when necessary. For example if tutors belong to both employee and student types, a specific tutor type is defined to inherit from both. In this way, the strong typing property is not violated and the desired functionality is provided.

- static type checking method is used to ensure type safety.

- our host language is an object-preserving (vs object-creating) language. It only returns objects of the existing types to preserve database integrity. The output of an operation can be the input of the next if the type compatibility is not violated.

- mechanisms are provided for the caller as well as the callee of transactions to be able to distinguish between those activities that are essential for the completion of a transaction and those that are not, thereby necessitating the use of compensating and contingency activities.

- value dependencies among transactions are supported for all types of parallel tasks including the ones getting activated at different times without
prior run-time knowledge of each other, so that they can proceed up to the extent where a value is needed from another thread. Similarly, commit dependencies are supported so that a task waits for a signal from another before it is allowed to commit. Also temporal, i.e., chronological dependencies - not related to value dependencies - among transactions are supported, thereby allowing more flexibility for database programmers in transaction scheduling.

- delegation is supported whereby subtransactions and their responsibilities are delegated to foreign sites. Communication cost is minimized by eliminating intermediate sites involved in finding the appropriate one to do the job. Possible cycles are detected by carrying and checking transaction and site ids. In this regard a set of classes is designed and implemented which provides the run-time information about site choice, services provided by close and foreign acquaintances, etc.

- compensatable subtransactions are distinguished and committed unsafely, and locks are released in order to facilitate cooperation as well as long-duration transactions in advanced applications. Automatic cancel procedures are also provided to logically undo the effects of unsafe-committed subtransactions if the global transaction fails. Rules defining compensating transactions are attached to objects and facilities are provided for programs to distinguish such tasks and proceed accordingly.

- exception handling mechanisms are provided to attempt to execute pre-defined handlers according to the type of exception, in cases where failure occur during execution of subtransactions. It allows to catch and distinguish different kinds of aborts.

- Two types of messages are supported:

  1. ordinary messages as in the conventional actor model which go to the rear of the target actor’s mail queue and are buffered. Such messages carry special tags with them to specify their contents and sender. Operations are proposed to query the message queue for particular types of messages from particular actors. However, despite the actor model,
although all ordinary messages are queued, they are not necessarily processed sequentially. The receiver may decide to wait and query for arrival of a particular type of message from a source without processing previous ones.

2. express messages (signals) which get processed right away irrespective whether the target actor is active or has other messages in its queue. Such a message which is restricted to “abort” and “timeout” in TractorS has different impacts on different types of actors: (i) base actors are interrupted, their effects on databases are undone, and they will never resume. (ii) meta actors pass the message to all their acquaintances and terminate.

Readers may refer to the second part of this thesis for detailed description of the scripting language. There are two chapters: the first one demonstrates the language from the interface point of view and the second one talks about implementation details.
Chapter 2

Related Work

2.1 Database Transactions

A transaction [17], [18], [19] is a sequence of operations which transforms a database from one consistent state to another. To satisfy this goal, a transaction should have the following four (ACID) properties[20]:

1. **Atomicity**: either all of the transaction actions are performed or none of its effects is reflected in the database. The transaction is said to **commit** in the first case and **abort** in the second.

2. **Consistency**: a transaction executing alone on an initially consistent database will leave the database in a consistent state upon termination.

3. **Isolation**: the effects of an in-progress transaction is hidden from other concurrent transactions.

4. **Durability**: effects of a committed transactions on the database cannot be abrogated.

Atomic transactions which preserve ACID properties are widely used in database and information systems. One of the major goals in such systems is to allow several users to access the information simultaneously while the integrity of the system is preserved. Assuming that each transaction executing alone transfers the system from a consistent state to another, two main problems should be handled:
1. controlling the interactions of concurrent transactions in order to prevent them from destroying the consistency of the database. This control is achieved through a variety of mechanisms known as *concurrency control*.

2. making sure that the effects of completed transactions are made permanent and the effects of incomplete ones are removed. An integral part of a database system is a *recovery scheme* which is responsible for the detection of failures and restoration of the database to a consistent state.

In this section we address these two concepts. More detailed information can be found in [21].

### 2.1.1 Concurrency Control

The main approaches to concurrency control are locking mechanisms, timestamp ordering, and optimistic non-locking methods. The other two are basically proposed to avoid well-known disadvantages of locking mechanisms. One important drawback of locking mechanisms, particularly in distributed environments is deadlock which happens when two processes wait for each other to release resources. Deadlocks should be either prevented, e.g. suspend transactions until they reserve all the resources they need, or detected and broken. It is too expensive to keep track of locks to prevent or detect and break deadlocks. One way to avoid the price is by a timeout mechanism. The site originating a transaction uses such a mechanism to eventually break the deadlock. It does *not* solve the problem completely however since timeout periods are different for different subtransactions. In cases like advanced database applications, a transaction may take days while another one takes seconds. If two long transactions are in a deadlock, resources are unnecessarily unavailable for a long time before a timeout breaks it. TractorS uses such a mechanism, but allows users to set the timeout period for each subtransaction, or set a default for a whole transaction as clarified later on in the thesis.

Optimistic non-locking mechanisms basically try to avoid unnecessary locking activities. They assume that transactions are non-conflicting almost always and suspend conflict checking to the last steps when a transaction is ending. If no conflicts arise the transaction is free to commit, otherwise it will be aborted.
These mechanisms are not appropriate for distributed environments, particularly in advanced applications. This is because deciding whether there have been any conflicts needs lots of information be recorded which is similar to keeping track of locks to prevent or detect and break deadlocks in locking mechanisms. Besides, if there has been any conflicts, transactions should be rolled back which may lose hours of work and the results, particularly for long transactions. Depending on the level of autonomy of sites, a series of cascading aborts may result.

In the locking protocols, the order between every pair of conflicting transactions is determined at execution time by the first conflicting lock that they both request. Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is called timestamp ordering. Timestamp ordering guarantees that a deadlock situation cannot arise. A data manager orders conflicting access to data items by multiple transactions according to their timestamps and the timestamps associated with database items. A protocol will decide which one can go, must wait, or must abort (and restart later). Timestamps provide a total ordering of transactions to be used for controlling concurrent access to the same data item. For multiversions of data, where different transactions may access copies of the same data item simultaneously, a multiversion timestamp ordering mechanism is proposed [22]. Although timestamp ordering solves the above problems by not using locks and not allowing conflicting access to data, its main drawback is restricting concurrency. A transaction may be done with a data item, but while still being busy somewhere else, it may cause others to unnecessarily wait or abort.

Concurrency of transactions can be improved by using transactions semantics. Semantic-based synchronization can be broadly divided into two groups. Models of the data approach define concurrency properties on abstract data types according to the semantics of the type and its methods. Semantic knowledge about individual types is used to develop synchronization strategies that allow more concurrency. An object offers a concurrent behavior regardless of the semantics of applications using it. The interleaving of concurrent transactions is implicitly constrained by operation conflicts defined on abstract data types. Models of the transaction approach define concurrency properties on transactions according to
Chapter 2. Related Work

their semantics and the data they manipulate. Here, interleaving is explicitly constrained by specifications on transactions. This approach requires centralized control with respect to concurrent transactions.

Semantic knowledge allows non-serialized but consistent transaction schedules. There are two well-known methods based on this principle [23]:

In the first method, the transactions are grouped into a collection of disjoint classes. Actions belonging to transactions of the same class are compatible and can interleave. Others should be serialized. In original models, a locking mechanism is used to ensure consistency [24][25].

In the second method, each transaction is divided into a set of steps. In the schedule, breakpoints are used to permit transfer from one transaction to another based on steps which do not violate consistency. Locking is used to produce such schedules in the original model [26].

2.1.2 Recovery

Recovery basically deals with failures which may prevent in-progress transactions from completion or remove effects of ones already completed. Both cause inconsistencies due to violation of atomicity and durability properties of transactions. A good system should be able to recover from most types of failures without human intervention.

Failures are of different types. The ones regarding data entry or transaction programming errors are beyond the scope of the thesis. Other types of failures irrelevant to this discussion are operators’ errors (e.g typing a wrong command) and hardware errors, for which mechanisms are actually designed to detect and recover. Failures directly relevant to centralized transaction management include transaction failure, system failure and media failure [21]. There are additional types of failures related to distributed transaction management. Site failure refers to partial or total failure of the sites in the network, i.e some or all of the sites being down. Communication failure refers to failure of communication links between sites. Some or all communication paths between two sites may fail. A partition situation happens when two or more site clusters can only communicate internally. A popular mechanism for detecting failures of other sites is by timeout. An extra requirement for distributed transaction management is
global commit synchronization. Popular methods are two-phase and three-phase commit protocols [27] [28].

Two types of operations are considered in data management: (i) cache management which provides operations to fetch and flush data between volatile and stable storage. (ii) recovery management which handles failures. It controls flush operations to ensure that the stable storage always has the data needed to restart a failed transaction. Updates may be done in-place which destroys the old copy each time a data item is overwritten (keeping one copy at a time), or by shadowing which keeps older versions as shadow copies. The recovery manager usually stores additional information in stable storage to log the history of execution. It enforces the cache manager to write enough information on the log which may be needed for undoing the effects of uncommitted transactions and redoing the effects of committed ones if failures occur. Efficiency of the recovery manager is very important since doing a restart prevents all users from accessing the database. Therefore the size of log is a crucial factor. The problem is solved by checkpointing which is an activity that writes information to stable storage during normal operation in order to reduce the amount of work the restart procedure has to do after a failure. The restart procedure fails if media failure destroys the needed information. The only recourse is to maintain redundant copies of every data item's last committed value. Keeping more copies (at different places, etc.) increases the probability of recovery.

A new recovery method called ARIES and reported in [29] fares well with respect to several metrics of transaction models including partial rollback, fine-granularity locking, novel lock modes based on commutativity and other properties, inter transaction activities, etc. It records in a log the progress of a transaction, and its actions which cause changes to recoverable data objects. ARIES is applicable not only to database management systems but also to persistent object-oriented languages, recoverable file systems, and transaction-based operating systems. It uses the widely accepted write ahead logging protocol and introduces flexible log record types (undo-redo, undo-only, redo-only). Transaction, system, and media failures are dealt with and logical undo is performed which allows early lock release by subtransactions. Some other goals of ARIES are simplicity, flexible storage and buffer management, partial rollback, minimal
overhead, parallelism and fast recovery. This method of recovery which covers
the compensation concepts is the closest to TractorS necessities.

2.2 Advanced Transaction Models

The ACID properties of conventional transaction models are not appreciated
in advanced applications, particularly ones which involve long-duration transac­
tions. Several mechanisms have been proposed to challenge the problem. We
summarize the most popular ones in this section and comment on the closest
ones to TractorS at the end. More information about them can be found in [30].

2.2.1 Sagas

Sagas [2] are long-duration transactions comprising a set of subtransactions
that can be interleaved in any order with subtransactions of other sagas. Each
subtransaction is associated with a compensating subtransaction. A saga re­
quires that either all component subtransactions complete execution or compen­sating
transactions are run to undo the effects of ones which have committed
before failure of the saga. This does not necessarily mean that the database is
restored to the state that existed when the transaction began. A saga is not
failure atomic, but it cannot execute partially. Subtransactions within a saga
execute in a predefined order. Both types of transactions (component and com­
pensating) have the ACID properties, but their behavior is constrained by certain
dependencies (e.g. a compensating transaction may only execute after failure of
its counterpart). Sagas may view the partial results of other sagas since com­
pONENT subtransactions commit independently. Therefore, consistency in sagas
is not based on serializability. Failure of a component forces the whole saga to
abort. In this respect, sagas do not have the flexibility that most of the advanced
transaction models have, i.e. they are not able to retry an aborted component or
ignore it or try an alternative instead. Database inconsistencies are handled by
compensating sagas.
Chapter 2. Related Work

2.2.2 Multilevel Transactions

The main features of the multilevel transactions model (also known as layered transactions) [31], [32] are exploiting the semantics of operations to relax isolation of concurrent transactions, using compensating transactions to achieve atomicity, and making partial results of subtransactions visible to other concurrent transactions. The model provides parallelism and supports long-duration transactions by utilizing the semantics of operations in an object oriented manner. It is a special case of open nested transactions in which nodes of the transaction tree correspond to execution of operations at particular levels of abstraction in a layered system. Hence all transaction trees have the same height which is equal to the number of levels in the underlying system architecture. It uses a multilevel concurrency control mechanism in which the semantics of level-specific operations is exploited to handle conflicts. Conflict relations are defined on operations as specified at particular level of abstraction rather than operation execution. The high-level operations are implemented by read and write accesses to the underlying records. For example if accounts a and b belong to the same branch and deposit operations update a branch total as well, their concurrent access to it is regarded as pseudo conflict and the schedule is regarded as serializable at the top level. If, in a two-level system, the conflict relation at the higher level is empty, the model is similar to sagas which is based on the assumption that all high level steps are conflict-free. A transaction is decomposed into a sequence of independent subtransactions. A concurrency control criterion called multilevel serializability is developed [32] which is based on the assumptions that all recovery-related steps are explicit actions in the transaction schedule and are subject to concurrency control, and the resulting complete schedule is multilevel serializable in the ordinary sense. The multilevel transaction model is relatively conservative compared to other models, but is fairly powerful and is shown to be applicable in a number of applications including extensible DBSs, federated DBSs, operating system transactions and object oriented DBSs [32].
2.2.3 Split-Transactions

Split-Transactions [3] are used for open-ended application, such as VLSI design, CAD/CAM and software development activities. The purpose is to split the objects of an on-going transaction among two or more serializable transactions. Certain concurrency properties are defined for operations of resulting subtransactions to ensure consistency. Split transactions are useful for:

- committing part of a transaction early, thereby releasing resources;
- making partial results of the original transaction available to others;
- delegating responsibility of incomplete parts to other ongoing transactions.

The model reduces isolation property and saves parts of work from subsequent failures. Furthermore, serializable access to resources is eased by defining the inverse operation of split, i.e join-transaction. On-going serializable transactions are joined as if they had been a single one. Hence transfer of resources can be achieved by a split where a resulting transaction joins another on-going one. The combination is useful in long-duration transactions. The real application is what the authors call user-controlled transactions where the operations are selected by the user as they go along. The transaction manager provides certain user commands for this purpose.

2.2.4 S Transactions

The S Transaction (Semantic Transaction) model [33] is a variation of nested transactions with some flexibilities. If a participant refuses to process a subtransaction or if a failure arises, an alternative source could be tried. Local autonomy of component databases are respected in this regards as they do not have to process a request. Furthermore, compensating transactions are used for recovery. Hence the isolation property is relaxed to subtransaction level. Traditional atomicity is replaced by a semantic one. A top level S Transaction either does all what must be done or cancels incomplete attempts in such a way that no logical inconsistencies are left in any database. If automatic recovery reaches its limits, i.e it cannot ensure the consistency of the databases or the timeout period is passed, a human intervention will rectify the situation. The probability of such
Chapter 2. Related Work

a semantical crash is decreased by trying to run local compensating programs several times. S Transactions are dynamically generated. No particular concurrency control or commitment protocol is recommended at the top-level. The local transactions are the concurrency units. Five types of autonomy, namely organizational, design, management, communication, and execution autonomies are addressed in the model. A component database is even free to break a communication process in the middle or may refuse to execute a commit or abort message in a global synchronization protocol. A timeout mechanism is used to control late subtransactions. The model was developed for an inter-organizational autonomous banking system. It is however more general and could be used for other application domains where local autonomy is the main goal.

2.2.5 Flex Transactions

The work on flexible and multidatabase transactions [4], [34], [35] is based on the observation that failures of individual transactions may be tolerated as a transaction may be accomplished by more than one local database system. They provide a framework where the designer may specify atomicity requirements of subtransactions as well as their precedence and data flow requirements. Unlike sagas, the model supports the concept of mixed transactions allowing compensatable and non-compensatable transactions to coexist within a single global transaction. It also incorporates the concept of time in scheduling of transactions and subtransactions. A global transaction in this model is syntactically a two-level nested transaction, but its semantics are expanded by allowing function replication, independent commitment of some subtransactions before the corresponding global transaction, and the specification of the value of completion time of (sub)transactions. Users are allowed to specify alternative subtransactions or sources of data for implementing the same task. The temporal dependency of the model supports transaction execution and completion orderings. The concepts of positive and negative dependencies are introduced for execution ordering and defining alternative subtransactions respectively. To facilitate the execution dependency specification, a transaction execution state is defined as a n-tuple which specifies if a subtransaction has been submitted, successfully completed, unsuccessfully completed, failed, or is being executed. Also for each subtransaction
Chapter 2. Related Work

an acceptable state set as well as a precedence predicate is defined. A global transaction is also defined as a tuple which specifies all its aspects. Scheduling of transactions is done by following some execution rules. The Predicate Petri Nets [36] are used to control the execution of global transactions.

The model is used in an InterBase prototype and also has been implemented in the Vienna Parallel Logic (VPL) language [37]. We address the language later in this chapter.

2.2.6 DOM Transactions

The Distributed Object Management (DOM) transaction model [38] is intended to facilitate the development of non-traditional applications in a distributed object-oriented environment which supports the co-existence of autonomous, heterogeneous systems, some of which may be non-database systems (e.g. file systems). The model allows a combination of closed nested transactions (called toptransactions) and open nested transactions which relaxes top-level atomicity constraint (called multitransactions) as well as compensating and contingency transactions and vital or non-vital subtransactions. Rules defining compensating transactions are attached to objects. Dependencies may be specified to force subtransactions to execute or commit in a specific order.

The main contribution of the model is to support disparate requirements such as active capability, heterogeneity, local autonomy, abstract operations, and long-duration activities within a single integrated model. It separates the transaction model from correctness criterion. The Transaction model determines capabilities and restrictions for users to write transactions while the correctness criterion determines acceptable concurrent transaction histories. Such a separation allows many transaction management schemes, some of which have been studied by the authors.

2.3 Transaction Languages and Systems

In this section we present an overview of some of the well-known transaction programming languages and systems which have similarities with scripting language for TractorS.
2.3.1 Argus and Thor

There exist some notable similarities between our approach and that taken first by Argus [39], [40], so we describe what the language is and does in some detail. Argus attempts to make conventional nested transaction processing available in the object-based language CLU. Although Argus is not intended for parallel programming, it presents constructs which could be used for the purpose [41]. Programmers should think about efficiency of data representations and the degree of concurrency. It is designed for distributed applications which require a high degree of fault tolerance. Argus programs are a collection of guardians. A guardian is a set of data objects, handlers (procedures), and processes encapsulated into a module. Multiple guardians may run on the same host, each one on a separate processor. There is a procedure called creator which could be called to create guardians dynamically. The creator specifies the node at which the new guardian is to reside. Guardians and handlers can be sent as arguments. Processes running in the same guardian communicate via shared data items. Mutual exclusion is guaranteed by a construct called mutex. Processes residing at different guardians however can only communicate via remote handler calls creating a new process in the receiving guardian and blocking the caller. Using the center construct, a process can call several handlers concurrently. Guardians could behave as either active or passive objects. A guardian may contain a background section running continually during the active life cycle of the guardian. When this section is exited (or not created at all) the guardian behaves as a passive object by accepting handler calls from other guardians only. Programmers can define a recovery section. If any guardian’s object is declared stable, a copy is kept on stable storage (called base versions) and is reset when the guardian is restarted (possibly on another machine) after a crash. Other objects of the guardian are initialized and the recovery section is executed. The background section is restarted as soon as the recovery is done. Modifications to a stable object are done on a copy version (in volatile memory). Upon commitment of an action, the copy version replaces the base version; if the action aborts the copy version is discarded.

The prime construct of Argus is the atomic action abstraction being a group of operations with execution and recovery atomicity properties. It provides a
number of built-in types of atomic objects (arrays, records, etc.) with the same kind of operations as their ordinary counterparts and additional support needed for atomicity. It also provides a mechanism for users to define new atomic data types [39]. The Argus nested transaction model supports topactions and nested subactions similar to the Moss model [1]. Every handler call is run as a subaction. It does not permit any concurrency within an action except by creating subactions. Each individual action runs at just one guardian to avoid anomalies such as an action that commits at one guardian and aborts at another. The run-time system does the locking automatically and uses conventional two-phase locking and two-phase commit protocols, with lazy lock propagation between nested transactions. The programmer must think about deadlocks, starvation, etc. and implement the code to avoid them when possible [42].

A new kind of data type called promise is presented in [43] which combines remote procedure call with futures. A future is a send/receive linguistic construct in which the sending process may proceed until the result is needed and is blocked when the result is not yet ready. Promise extends futures in several ways. It uses strongly typed objects, addresses node failures, and utilizes exception handling. In promise, the caller and callee can run in parallel. Promise is implemented in Argus. A new replication algorithm based on primary copy technique as well as a special kind of timestamp (viewstamp) to detect lost information are introduced in [44] to improve performance. Computations run at a primary copy which notifies its backups of what has been done. If the primary one crashes, the backups are recognized, and one of them becomes the new primary. Argus has been used for a collaborative editing system[45], a distributed mail repository [46], a long-running parallel application[47], and several small applications.

A new object-oriented database system called Thor has recently been announced by the author of Argus. Thor is intended to be used in heterogeneous distributed systems to allow programs written in different programming languages to share objects in a convenient manner. Thor objects are persistent in spite of failures, are highly likely to be accessible whenever they are needed, and can be structured to reflect the kinds of information of interest to users. Thor combines the advantages of the object-oriented approach with those of relational databases. Users can store and manipulate objects that capture the semantics of
Chapter 2. Related Work

their applications, and can also access objects via queries.

2.3.2 Avalon/C++

Avalon/C++ [48], [49] is a recent distributed transaction programming language developed at Carnegie Mellon University. Although Avalon/C++ and Argus provide much of the same functionality, the former permits more concurrency by providing the ability to query and possibly relax the transaction serialization ordering at runtime (but still remains compatible with the two-phase locking mechanism used in Argus). Avalon/C++ allows programmers to “customize” the synchronization and fault-tolerance properties of new data types by letting them inherit properties such as serializability and recovery from a library of basic types. A program in Avalon/C++ consists of a set of servers, each of which resides at a single node and encapsulates a set of objects and exports a set of operations and a set of constructors. Servers do not share data directly, but communicate by calling one another’s operations. Constructors help application programs to create servers at specified nodes. Operation calls only accept by-value parameters. As in Argus, objects within a server may be stable or volatile. Transactions being identified with a single process could be created in sequence or in parallel. Users can define atomic types by inheritance from existing atomic types. Atomic objects have ACID properties as in the conventional nested transactions model. Recoverable objects save results of subtransactions to guarantee persistence in the presence of crashes. The base hierarchy consists of three classes, recoverable, atomic, and subatomic. Recoverable, the most basic class provides primitives for ensuring persistence. Atomic and subatomic classes which are subclasses of the recoverable class provide primitives for ensuring atomicity. Subatomic class gives the programmer a finer-grained control over synchronization and crash recovery. Programmers can define non-atomic but recoverable objects.

2.3.3 Interactions and TaSL

A recent open-nested transaction model for defining long duration tasks is called Interactions and its underlying language is called TaSL [50]. Interactions operate on a heterogeneous multirelational database environment. Tasks are broken up into
smaller atomic units called subtasks. Alternative subtasks can be specified where possible, so that more than one set of subtasks can accomplish the overall task. With Interactions, flexibility is specified as alternative sets of global transactions. The component databases are accessed via procedures called steps. The steps encapsulate information in their corresponding local database and allow it to be accessed uniformly.

The language TaSL focuses on the database issues of transaction and recovery. It allows a multidatabase user to define an Interaction in terms of the steps provided by the local databases. Sequences of steps can be specified to execute atomically in the multidatabase. TaSL supports the ability to backtrack when some transaction aborts and causes its subtasks to fail. An alternative execution plan may be tried, but backtracking of the previous one should be done first. The user also can specify constraints that must be maintained to ensure consistency of the multidatabase. Compensation mechanisms are proposed to remove the effects of Interactions that violate such constraints. A mechanism called Agent coordinates the serialization and commitment of global transactions and executes their steps on the local database.

2.3.4 STDL

S Transaction Definition Language (STDL) aims at bringing the S Transaction model introduced in previous section into use. The language has data definition and data manipulation parts.

The STDL/DDL has a request data section in which data types and data structures for the data being exchanged between sites are defined. The input data section defines the input parameters of the S Transaction to be invoked. In the local data section internal variables and intermediate results are managed. It provides the primitive data types and operations corresponding to those of modern high-level programming languages. A special constructor Table-of is provided for relational database support. Some other operators and built-in functions are also provided which help in constructing more complex data types as well as manipulating data and transferring them between sites.

STDL/DML has constructs to support service providing between subtransactions. A S Transaction is divided into subtransactions that can be invoked
(possibly by remote sites) via continuation points. The root S Transaction is activated at the special continuation point \textit{init-cp}. The REQUEST and RESPONSE messages help to invoke continuation points and submit results back. Orders are defined by means of the operators indicating sequential, parallel, or conditional execution. Local data could be used for synchronization. Failure or success of subtransactions can be indicated between processes. For every global and local S Transaction, a semantical compensating transaction is assumed as their integral part.

### 2.3.5 VPL

The Flex Transaction model introduced in previous section is also implemented by the Vienna Parallel Logic (VPL) language [37] which has implicit parallelism and supports the idea of compensating transactions. The language is a superset of Prolog which gives the programmer the ability to specify sequential or parallel execution of subtransactions. Both compensatable and non-compensatable transactions as well as success and failure dependencies are supported in the language.

The VPL language consists of three layers: the kernel, the primitives, and programmer-defined procedures. VPL kernel is based on the principles of resolution and unification. As in other logic programming language, the execution of a VPL program is viewed as finding a proof for a given goal. Goals within a parallel conjunction may be executed in parallel. If a parallel procedure consists of more than one way of proving a goal, the VPL inference machine selects an arbitrary one; if the goal is not proven, then the inference machine must undo the effect and select another way (backtracking). The programmer may influence the selection of an alternative way. Special \textit{cut} and \textit{commit} operators are provided to prevent the interface machine from performing needless backtracking. The commit operation is based on the idea of compensating transactions. The primitives help in implementing various types of communication, synchronization, and error handling as presented in the Flex Transaction model. The VPL language is a superset of the Flex Transaction model and can even extend the ideas presented in the model.
2.3.6 Arjuna

Arjuna is a public domain persistence system. Arjuna supports nested atomic actions (transactions) for controlling operations on persistent objects (instances of C++ classes). Arjuna has been implemented in C++ to run on a variety of platforms (Unix on SUNs, HPs, etc). The software available includes a C++ stub generator which hides much of the details of client-server based programming, plus a system programmer’s manual containing details of how to install Arjuna and use it to build fault-tolerant distributed applications. Several enhancements and ports on various distributed computing platforms are in progress as announced by authors. Some new features of Arjuna are faster object store, support for replicated objects, and memory resident object store.

2.4 The Actor Model of Computation

The actor model was proposed in [5] and since then has evolved over time, the form mostly used now is the one proposed by in [6]. Actors are concurrent active objects that communicate via message passing. Each actor has a mail queue which can accept and buffer a finite number of messages, a set of acquaintances (known actors), and a script which defines its current behavior and is normally a set of methods (procedures). An actor can send messages to any other actors it knows.

Messages arrive in a linear order into the mail queue. In response to processing an incoming message, an actor may take a finite set of actions of the following three types, and may also process simple conditional statements. Figure 2.1 illustrates a conceptual representation of an actor, which may:

1. send messages to specific actors that it knows (including itself);

2. create new actors;

3. specify replacement behavior to process the next message. The current behavior will act as the replacement behavior if not explicitly specified.

Upon creation of an actor, a unique mail address is assigned to it as its identification. An actor knows all the actors created by itself. It also may know other
Several actor languages with more powerful specifications are proposed to the literature. Perhaps the most comprehensive is the family of languages which perform the three basic operations: send messages to an actor, replace an actor, and destroy an actor. Ordinary structures like conventional data structures, loop statements may also be used.

A family of actor languages exists [6]. Other languages, such as Act, Act2, Act3, Ada, and SAL already exist. [6]. Other languages, such as Act, Act2, Act3, and SAL already exist. [6]. [SAL, ABCU1] has most attractive features; they consist of objects to exist with other types of data, such as numeric constants, and objects communicate via passing messages. Other traditional statements are similar to being viewed as instances of message passing among objects. The instances of objects are fully encapsulated; the only data a message sees is that which passes their values. An object in ABCU1 has two operations: send and receive. Send sends a message to another object, and receive receives a message sent by another object.

2.5 Conclusion

The conceptual representation of an actor

![Conceptual representation of an actor](image_url)

Figure 2.1: Conceptual representation of an actor

actors whose mail addresses appear in the incoming messages. The mail address of an actor can be communicated as desired. Therefore the set of acquaintances of an actor can grow over time [6].

An actor system comprises two parts:

1. a collection of actors

2. an operating system which creates and destroys actors and passes messages among them.

It may happen that an actor is not active and may not subsequently be activated,
and thus not reachable by any other actors. Such an actor should be garbage-collected by the underlying system [51].

Several *actor languages* with diverse goals and specifications are proposed in the literature. Primitive constructs of an actor language are the ones which perform the three basic operations of an actor. The following is a typical syntax:

1. Send messages: SEND message TO actor;
2. Create an actor: NEW actor;

Ordinary structures like conditional and loop statements may also be used.

A family of actor languages including Act, Act2, Act3, and SAL already exists [6]. Other linguistic conventions in this paradigm which are closer to our point of view are ACT++, ABCL/1, and POOL-T which have proposed object oriented derivatives of the actor model tailored to fit diverse requirements [52], [53], [54]. ABCL/1 has more attractive features than others. It allows objects to coexist with other types of data such as numbers and lists. While objects communicate via passing messages, other traditional types are manipulated by operations as in conventional programming languages. Common constructs such as conditional statements are primitive concepts in the language rather than being viewed as instances of message passing among objects. The instance variables of objects are fully encapsulated, i.e. only the object itself can access their values. An object in ABCL/1 has two message queues for *ordinary* mode and *express* mode messages. The receipt of an express mode message will interrupt any ordinary computation underway to process the express one. The ordinary mode resumes when the express mode is finished.

### 2.5 Conclusion

TractorS combines attractive properties of the actor model of computation with the transaction management constructs in an object-oriented environment to address advanced necessities in transaction management. Actors of different types are utilized to access data items, make decisions, enforce orders, etc. They appear all over in TractorS transaction tree.
Our research activities center around a linguistic framework which utilizes the appealing properties of the actor model to support distributed transaction processing in an CIS environment. In section 3.2.2 we describe how these properties are used, reason about using them in this context, explain our deviations from the conventional actor model and argue about its shortcomings, and show how the actor constructs are combined with transaction management operations to manage distributed transactions as proposed in our model. More detailed information about this matter can be found in [55] and [56]). Our choice from the actor language paradigm was ABCL/1 which is more pragmatic than the conventional actor model and implements a novel and interesting collection of message passing constructs. Reasons for not choosing the language for TractorS implementation are given later on in the thesis.

Among the research work conducted on open-nested transaction properties and models, we distinguish the work on the Distributed Object Management (DOM) model [7] and on flexible MDBS transactions [4], [34], [35] as most relevant to TractorS. The DOM transaction model is intended to facilitate the development of non-traditional applications in a distributed object-oriented environment which supports the co-existence of autonomous, heterogeneous systems. As in the DOM model, rules defining compensating transactions in TractorS are attached to objects. Facilities are provided for programs to distinguish such tasks and proceed accordingly. The work on flexible and multidatabase transactions are based on the observation that failures of individual transactions may be tolerated as a transaction may be accomplished by more than one local database systems and provide a framework where the designer may specify atomicity requirements of subtransactions as well as their precedence and data flow requirements. All these ideas are of primary importance in TractorS.
Chapter 3

TractorS: A Semantic-based Transaction Model

3.1 An Architecture for Cooperative Information Systems

A typical CIS environment comprises sets of heterogeneous, autonomous, and distributed information systems, e.g., databases, knowledge-bases, and file systems interconnected via a common communication network. In [8] and [56] knowledge-based processing attachments to the object-oriented paradigm and the underlying transaction management architecture are proposed. It is suggested that CIS are built around the concept of an information agent which is the equivalent of a logical front-end in traditional federated database technology. Agent wrappers provide increased intelligence and virtual homogeneity – by adapting to passive heterogeneous information resources – as well as ease of inter-agent communication and increased modularity. Distributed objects are defined in a common object-oriented data model which incorporates knowledge-based facilities, such as rules and triggering mechanisms, and is referred to as an expert database system.

Each information agent focuses on solving problems in the domain of its expertise. Several information agents can interact cooperatively to solve a multitude of complex information-intensive problems which can only be solved by selectively fusing and combining problem solving expertise and data/information from diverse problem solving nodes in the network. Each information agent must be
able to reason about the need for cooperation and understand local and global knowledge to locate the other information agents involved in the processing of a global request. In order to minimize the amount of knowledge stored locally and facilitate information agent interaction, agents are organized into clusters as referred to in [8], whereby each cluster is organized around some common domain of expertise, such as customers, products, or revenues. Clusters can overlap in that agents can belong to more than one cluster. Normally clusters are static. We assume that changes at the schema level of existing databases will not lead to changes in clusters. However adding (deleting) a site or database to (from) a network must be reflected at the cluster level.

Figure 3.1 illustrates an architecture for CIS. Each site has its own global transaction manager as well as expert database system. An information agent at each site is seen to consist of the following components:

1. the intellect which is the faculty for knowledge, reasoning and decision making developed by an expert database system (EDS) which has access to two sets of specialized knowledge sources:

(a) the Close Acquaintances Knowledge Source which contains semantic and structural descriptions about the data items available in other sources that belong to the same cluster. It knows in detail all about a node’s subject area and includes information about the data items available by other sources that model subject areas overlapping with that of the home site. Information (or data) underlying a certain subject-area may be located in more than one site with redundancy. It is the purpose of intellect to alleviate such problems, as well as to resolve conflicts and in general guarantee virtual integration.

(b) the Foreign Acquaintances Knowledge Source which contains the meta-knowledge about “who can do what we can’t do” in the network. This source keeps descriptive information about other sites that model problem domains that are disjoint or complementary to that modeled by the home site and thus belong to different agent clusters.

2. a Global Transaction Manager (GTM) which accepts global transactions and handles their execution, comprising the following components:
Figure 3.1: Global transaction decomposition in an information agent network

(a) a Transaction Planner and Decomposer (TPD) which, in interaction with the intellect decomposes global transactions and provides necessary information for scheduling them. To minimize the communication cost, TPD tries to extract as much data and information as possible from the local site and its close acquaintances and in general optimize the number of participants (nodes needed to process a global request).

(b) a Transactional-actor System (TractorS) which accepts decomposed transactions together with sites to execute them, from the TPD or advanced users, and manages their execution and dependencies until they commit or abort. An important part of TractorS is a script-
Chapter 3. TractorS: A Semantic-based Transaction Model

TractorS: A Semantic-based Transaction Model

The TractorS environment is written in Prolog and acts as the interface of the TractorS environment. It combines the results of subtransactions and produces reports. Depending on the application, it may produce intermediate reports while a transaction is still in progress.

All sites in the network cooperate in solving global problems, have an equal status and retain a high degree of autonomy by maintaining their behavior and local control.

3.2 A Transaction Model for Cooperative Information Systems

3.2.1 Distributed and Multidatabase Transactions

In distributed systems, a transaction consists of some subtransactions executing at different sites. There exist several methods for distributed transaction management [57]. A popular one is to consider the site which issues a transaction as the root agent and let it have the responsibility of communicating with other sites and following the transaction until it commits or aborts.

A popular transaction model for distributed environments is the nested transactions model [1], [22] which considers a whole transaction as a tree. The root of the tree is the global transaction. Its children are subtransactions running at different sites. Each subtransaction can exhibit the same recursive structure. Subtransactions only inform their parent of their commits or aborts. Subtransactions in this model, possibly running at different sites, do not necessarily satisfy the four ACID properties. A subtransaction may not transform the database into a consistent state (violating the C property), but a number of them together will do so. For example a money-transfer transaction may have two subtransactions to withdraw some funds and deposit it in another account. The database is in a consistent state when both subtransactions commit. Similarly effects of a subtransaction on databases are not always persistent (violating the D property).
since it is undone if the parent aborts (failed subtransactions can be repeated or replaced). In the Moss model the number of locking activities in the concurrency control process is reduced compared to flat transactions since the parent inherits locks of its committed children [1] (Reed model does not use locks).

The main advantages of nested transactions are their ability to restart or replace a failed subtransaction without forcing the upper level transaction to abort, thereby improving performance by decomposing transactions into concurrent subtransactions. Moreover, they provide modularity as well as finer grained recovery. However, conventional closed nested transaction models present three major shortcomings with regard to advanced transaction requirements:

1. they enforce strict serializability in the transaction tree with a hierarchical isolation constraint. Other concurrent transactions should wait for completion of the top-level transaction to release locked data items no matter how long it would take.

2. they do not reveal partial results produced by subtransactions to other subtransactions outside the scope of the parent.

3. subtransaction commitment is subject to the commitment of its superior subtransactions. Its effects become permanent only when the enclosing top-level transaction commits.

As a result the model is not flexible enough for fairly complex global transactions.

Despite its limitations the nested transaction model provides several appealing properties for distributed processing and has been used extensively as a basis for the development of long-lived transaction models appropriate for multibase database systems (MDBSs). The limitations of the conventional transaction concept has been studied for quite some time in the context of long-lived transactions used for advanced application domains such as CAD/CAM, office automation, software engineering environments and design environments [10]. such transactions are usually very complex, need to lock data items for long periods of time, have higher probability of failure due to their long execution time and cannot be easily rolled back due to loss of substantial amount of work being done. To surmount the problem of long-lived transactions several mechanisms have been proposed [10].
Chapter 3. TractorS: A Semantic-based Transaction Model

The closed-nested transaction model of Moss has been generalized to a two-level (global vs. local mode of operation) open nested model such as that proposed by sagas, split-transaction and combinations of these types. Open-nesting relaxes the top-level atomicity restriction of closed nested transactions by allowing partial results of subtransactions to be exposed to other transactions. This introduces the concept of compensating transactions which are the logical equivalent of rollback since they undo the effects of a committed transaction in a logical manner. It also becomes necessary to define contingency transactions which are executed when a subtransaction fails, vital subtransactions which force their parent to abort once they have failed and non-vital subtransactions which may abort without forcing their parent transaction to do so. Various models are proposed which violate the ACID properties in different ways to address transactional necessities of various application domains. We have already addressed them in chapter 2.

Multidatabase systems comprise several pre-existing heterogeneous and autonomous databases, usually referred to as local or component databases. The purpose of a MDBS is to support global applications that rely on and access data items spread over more than one local database. To achieve this objective MDBSs attempt to integrate local databases into a seamless whole and provide for global transaction facilities for enabling the processing of disparate data items over a network of interconnected local database systems. Remote data items are accessed by global transactions which are eventually decomposed into a set of subtransactions that instigate local transactions in the component databases.

An important feature of MDBSs is the preservation of local autonomy: each local database system has the right to access and administer its own data, use its original applications and continue functioning unaffected by the presence of the MDBS software. This reflects the fact that local database management systems were developed independently in isolation and may belong to different organizations and should not be tampered with to allow for MDBS functionality.

Most of the work in the area of multidatabase systems has relied on the existence of conventional (short) transactions; assumes the existence of a two-level nested transaction model for the processing of remote data; and adheres strictly to the classical ACID paradigm for network-wide transaction management [58], [59], [60], [61], [62]. In the two-level transaction model each component database
system manages its own local transactions in addition to subtransactions generated by a given global transaction. Global transactions in their turn are managed by the MDBS software. However, this model of MDBS transaction processing introduces several acute limitations. Firstly, MDBS transactions result in long-lived transactions which may lock local database resources for unacceptably long periods of time delaying significantly the termination of conventional short transactions submitted at these sites (and which are outside the scope of the MDBS). Secondly, serializability theory is inappropriate for MDBS transactions since it does not differentiate between local and global transactions and restricts the set of acceptable histories. To remedy this situation the concept of multilevel serializability has been proposed [63]. From this discussion it is apparent that the traditional ACID properties are far too restrictive for MDBS applications. These deficiencies have been pointed out and solutions have been suggested in the research work conducted on the area of non-conventional database transaction management for MDBS and distributed object-oriented systems [4], [38], [64], [65], [35].

Transaction concepts in multidatabase environments have begun to be applied to support applications of activities that involve multiple tasks [50], [66], [67]. The designer of such applications may specify inter-task dependencies to define task coordination requirements. They may also define additional requirements for isolation, and failure atomicity of the application.

The purpose of this project is to present a transaction model based on the proposed architecture for CIS, and its underlying linguistic support. We have designed and implemented classes which provides information about the sites in a typical network, clusters, their close and foreign acquaintances, and the type of services each one provides. Other aspects of CIS, not related to its transactional necessities, is beyond the scope of this thesis. The proposed CIS transaction model includes synchronization primitives as part of its underlying transaction programming linguistic support and is in part designed to capture the requirements of open systems. It provides a variety of structures which facilitate the process of cooperative-problem solving by representing the CIS information agent space as a dynamic network of intercommunicating systems of active objects, i.e
Chapter 3. **TractorS: A Semantic-based Transaction Model**

objects having their own threads (lightweight processes)\(^1\), for sharing intermediate results and synchronizing computational activities. The linguistic framework provides convenient abstractions for data objects and invocations of open and closed nested transactions and uses message passing as the predominant paradigm for interprocess communication.

We envisage two possible scenarios for the use of the linguistic facilities:

1. a simple internal structure which can be readily used as an intermediate level onto which the higher-level language of the information agents (used for querying the MDBS and developing corporate system-wide applications) can be compiled. This is a system driven activity which implies that the information agent's intellect decides - after consulting its appropriate knowledge sources - which subtransactions should be submitted or committed and in which order. The information agent explicitly defines ordering properties and control flow of subtransactions at each level of nesting such that maximum possible concurrency is achieved, and passes this information to TractorS.

2. a high-level scripting language for advanced users to program distributed applications that need to address database and distributed system programming aspects. We have focussed on this type and developed the language. The third part of the thesis is entirely devoted to this matter.

### 3.2.2 Actors in Transaction Management

The term *TractorS* which stands for *Transactional-actor System* conveys the meaning that our transaction manager is an actor system. TractorS adopts attractive properties of the actor model of computation, but eliminates its drawbacks. The most important drawbacks of the actor model are lack of modularity and garbage collection [68]. For example, if an actor is part (script) of another actor, the outside world should not normally be able to communicate directly with the inner one. This can however happen in the conventional actor model because the mail addresses of actors are freely communicated. Our transaction management

\(^1\)We consider *passive objects* as objects equipped with two components: data, and methods (procedures). Active objects essentially extend passive objects with thread capabilities.
system is a highly modular actor system in which the acquaintances of actors are controlled via certain constructs in the model, the strong typing capability of the scripting language, and the controlled communication mechanisms. An actor communicates only with those actors which are permitted either by the type system or by the semantics of the transaction. This happens mostly in the context of parent-child communication or top-level synchronization. Other acquaintances of actors are ones which have value dependency, commit dependency, or argument dependency with them.

As an example, schedulers of different types communicate with each other and with the atomic transactions to find appropriate solution to a global problem. The communication path hierarchy is defined by the type system and authorized users. The communication is highly modular. The hierarchy is a nested transaction tree in which a scheduler is responsible for its subtransactions; it creates them in the order defined in its type and communicates with them until a solution is reached. This happens recursively until the top-actor is terminated (committed or aborted). A scheduler has a mail queue which accepts messages. Each message carries information about its originator, type, sites already gone through and so on to help the receiver make a decision. It may create another actor, possibly at another site, to process the message and specify a replacement behavior to receive the next message which could be related to the same subtransaction or another one. It may send the message to an existing actor, or may do the job itself. The subtransaction can be delegated to a foreign acquaintance or rejected (aborted) if the receiver is not clear what should be done. The upper level actor however tries other sources before giving up. In an interaction between scheduler actors and the user, the transaction may be tried at all sources to prevent it from aborting. This has been missing in conventional static database transactions which abort if something goes wrong.

The top-actor has a very important role in the final agreement and commitment process of a distributed transaction. When all schedulers in a top-level transaction are done, the following takes place:

- the responsibility of all base-actors is inherited by the top-actor.
- a direct communication path is established between the top-actor and base-actors.
• since the responsibility of delegated transactions is also delegated, the inter-
face actors delegatee behave as a top-actor at the target site and as a base-actor in synchronizing with the main top-actor.

• Intermediate schedulers do not take part in the global synchronization pro-
cess. This is because they do not access database items.

Garbage collection of actors, particularly in distributed environments, is a
complex and lazy process. It is performed in two phases [69]. In phase one,
the actors which are not active and can not become active in future are marked.
This can be a tedious process in a reasonable-sized network. In the second phase,
the marked actors are garbage-collected, i.e. are sent back to the memory pool.
Garbage collection is an automatic and optimized process in our scripting lan-
guage. The run time system of Sather decides when objects are garbage collected.

The commonalities between TractorS and the actor model of computation
and its well accepted extended models are summarized in the following points:

• asynchronous message passing.

• creating and garbage collecting active objects.

• direct communication between active objects.

• changing state to process the next message.

• delegation as a means of passing a computation from one information agent
to another to continue processing with self-responsibility, and communicate
results back.

• simple mode messages as well as express mode messages, which get pro-
cessed immediately.

3.2.3 Scheduling and Synchronizing Transactions

Transaction trees in TractorS not only contain subtransactions but also ex-
press their dependencies explicitly. These dependencies are enforced by interac-
tions between schedulers. There are four types of schedulers in TractorS:
Chapter 3. TractorS: A Semantic-based Transaction Model

1. a **serial scheduler** whose subtransactions are *submitted and committed from left to right*. They have begin-on-commit dependency on each other, i.e one cannot begin unless the previous one commits. If any of them fails either during its execution or after being done, the parent aborts all others which are already started. Success or failure of subtransactions is recursively based on success or failure of the leaves being base actors. All subtransactions should commit in order to make the parent commit.

   Subtransactions of a serial scheduler may have *argument dependency*. This is the case where a subtransaction gets some of its arguments from a sibling which is already done, before being able to start execution. The case is similar to executing a transaction and then passing its results to the next one as arguments.

2. a **parallel scheduler** allows all of its subtransactions to be *submitted and committed in parallel* as independent activities. Here also all subtransactions should commit in order to make the parent commit. The subtransactions may have dependencies of two types:

   - value dependency, i.e one process produces a value and another one consumes it. The producer “signals” its completion and the consumer continues processing and “waits” for the value to appear whenever needed. This is similar to “semaphores” in interprocess communications.

   A major point to notice is that the subtransactions with value dependency are in-progress while subtransactions of a serial scheduler with argument dependency should commit serially.

   - commit dependency, i.e two activities may go in parallel but one may not commit unless the other commits first. Again the synchronization is done via the “signal” and “wait” method.

   One should notice that the type of dependency called *commit-serial dependency* in the literature is equivalent to a parallel scheduler with commit dependency. Due to the richness of parallel scheduler in TractorS, the dependency type does not occupy an independent construct here.
3. a *serial-alternative scheduler* attempts subtransactions from left to right until one produces the desired outcome. This type of scheduling node corresponds to the negative dependency defined in [4] or the contingency transactions used in DOM [7]. The parent only aborts if the last subtransaction aborts.

4. a *parallel-alternative scheduler* is a parallel choice. In this case several alternatives are attempted in parallel. This is often useful in distributed searches and decision procedures in which several information sources contain enough information to resolve a question but a decision has to be made very quickly in order to proceed with transactions subject to deadlines. As soon as the answer is available from any one of the subtransactions, the scheduler commits and the effects of other parallel subtransactions are undone.

While all subtransactions of a parallel or serial scheduler must commit for the scheduler to commit, for choice nodes only one committed subtransaction is sufficient to commit the parent.

A scheduler may have non-vital subtransactions, i.e. ones which may abort without aborting the parent. This case makes sense just for *parallel* schedulers because in a serial scheduler later subtransactions depend on earlier ones, and, in choice schedulers all subtransactions are semantically non-vital (so there is no need for such a explicit declaration). There is however no need to have extra constructs for them. Non-vital subtransactions can be simulated by a *serial-alternative* scheduler with a pre-defined null subtransaction as the second (and last) child. The null subtransaction commits immediately after getting started without accessing any data items, so if the actual subtransaction fails the null one sends a commit result to the parent which will be able to see what has happened.

Schedulers may have conditional subtransactions, ones which get activated if certain conditions hold at run time. They may also have replicated subtransactions, the same one to be sent to different sites. The user may define such sites or leave it to TractorS run-time system to decide. In chapters 4 and 5 we address all the aspects of the schedulers from the implementation point of view.
3.2.4 Case Study: Home-loan Request Transaction

Before further developing other ideas such as early commit and delegation, we demonstrate basic features of the transaction model by means of a case study. We also compare the way conventional nested transactions would handle the fairly complex task with the way TractorS handles it.

The case study is a transaction which tries a home-loan request at some alternative banks as well as insurance companies to support it. A familiar example is chosen to help illustrate various features without introducing its own complexities. In the proposed environment an applicant does not need to refer to different banks and insurance companies to decide about his loan. Instead the system inputs his/her conditions and restrictions, tries all the sources in the network, and recommends the best possible plan. It does things in parallel and tries alternative sources in a flexible manner to succeed. Even if the applicant cannot provide the necessary loan deposit, it seeks a personal loan for him/her at alternative sources.

The top-level home-loan transaction tries to find a bank to offer the loan as well as an insurance company to support it, and then finalizes loan activities if this step is successful. Finding a bank and an insurance company could go in parallel up to some extent, but finalizing the loan comes after. Figure 3.2 depicts the transaction. The shape of a tree node indicates its type as introduced before. For instance the root is a rectangle representing a serial scheduler. Solid arcs, dotted arcs, and dashed arcs represent decomposition/nesting relationship, serial argument dependency, and value dependency or commit dependency respectively (a dashed arc without a label represents a commit dependency and with a label represents value dependency).

The transaction tree is composed of several actors of different type. The home-loan transaction is a serial scheduler with the following two subtransactions:

1. the first subtransaction (find-institutes) is a parallel scheduler which tries to find a bank and an insurance company concurrently. It does so by means of two subtransactions find-bank and find-insurance-co.

2. the second subtransaction (finalize) is activated if the previous one succeeds. It opens an account for the loan at the bank offering it, retrieves the balance of applicant’s account to possibly get the loan deposit from, starts
Figure 3.2: Transaction tree for the home-loan request transaction

the finance subtransaction to provide the loan deposit, and finally updates records and issues documents.

In the first branch there are the find-bank and the find-insurance-co subtransactions. We assume that the applicant has preference over banks but not insurance companies, i.e he/she likes banks to be tried in a certain order but any insurance company will do. Hence find-bank is a serial-alternative scheduler of atomic transactions offer-home-loan at pre-defined sites, while find-insurance-co is a parallel-alternative scheduler of atomic transactions support-home-loan at possible sites. This is a good example of different ways of specifying sites for subtransactions. Such sites may be specified (possibly with a preference order by the
user), or left to the system to do so. A combination is also possible, i.e specifying it for some of the schedulers in the transaction tree or even for any number of the children of a scheduler. The find-insurance-co branch continues up to the extent where its subtransactions need the bank name offering the loan, i.e they have a value dependency over the succeeded child of the find-bank subtransaction. The dashed line labeled bank-name in the figure shows this dependency. At the point where the value is needed they should be suspended until the value is provided. It makes sense to let them go in parallel however because a lot need to be done before the decision is made and the bank-name is needed.

The second branch, finalize, is a serial scheduler of subtransactions open-account, balance, finance and update. Here finance is a parallel scheduler with three subtransactions. The first two are withdraw and deposit. There is no need to assume that the deposit should be done only after the success of withdraw since, being related to the same top-level transaction, either both commit or abort. The third subtransaction of finance is the conditional subtransaction personal-loan. It is conditional because the applicant may not need a personal loan, i.e existence of the subtransaction depends on run-time values for different applicants. It has a different nature than the home loan one. The bank offering the home loan and/or its close acquaintances may or may not offer personal loans. If not the subtransaction is delegated to some foreign acquaintances to provide the service required. It is a parallel-alternative scheduler with subtransactions offer-personal-loan. The first response from a site is accepted and other requests are aborted.

The subtransaction finalize could be configured in different ways. For example one could put balance and finance under a separate parent (a new serial scheduler) because balance is not related to other children. It is important to configure transactions in the right way. The number of schedulers should be minimized since each one is a separate actor with its own process. Also the right type of scheduler should be selected for each task. Things should go in parallel unless the nature of the subtransactions is really serial. A good example is finance. Despite the conventional money-transfer transaction, the withdraw and deposit actions can go in parallel as mentioned above. Parallelism is particularly important in distributed environments because sites may not respond quickly due to various reasons and therefore should not wait for each other unnecessarily.
Serial argument dependency also exists between some subtransactions in this case study. The two children of the top-level transaction have this type of dependency because the second one needs the information about the actual bank and insurance company to be able to finalize the loan. Also the last two subtransactions of finalize have argument dependency on the first one; they need the account number to transfer money to and update the related records respectively.

The choice constructs above are based on commitment and express provision of alternatives in case of failure. For transaction scheduling this appears to be the most common case. However, sometimes an ordinary conditional is needed as seen in finance, for which we use the usual conditional construct. To avoid unnecessary cluttering and keep our graphic representation simple and concise we allow conditional constructs to appear on any decomposition arc between a node and its subtransactions.

The subtransaction offer-personal-loan can be considered as a compensatable transaction. If the home loan process is not finalized a compensating transaction will cancel its effect. The data items can be released by committing the subtransaction unsafely. Whether a subtransaction is actually compensatable or not is decided by its semantics. The system will take care of committing unsafely and then canceling such transactions if necessary, without involving human intervention. However, in particular cases where the timeout limit is passed, authorized programmers have the choice to stop automatic cancel procedure and do the job manually.

To observe the flexibility of TractorS, let us see how the case study could be managed in conventional environments with closed nested transactions. At least the following separate top-level transactions, similar to the subtransactions above, should be executed one after the other:

1. find_bank with its subtransactions is hard but semantically possible to be programmed as a nested transaction. In conventional environments each bank should actually be tried separately;

2. find_insurance_co is similar;

3. open_account, money_transfer and balance as one nested transaction;

4. personal_loan is actually similar to the whole home-loan transaction and
may need several top-level transactions since there is no notion of delegation in conventional environments.

5. the rest of finalize as one transaction.

The transactions should commit independently. Programmers need to pass information from one to another and control their execution. Keeping in mind that in an autonomous environment a site may process a request at a convenient time, controlling such a number of related transactions could be a tedious task. Programmers normally get involved in other transactions in between, which may cause confusion and inconsistency. Various types of failures add to the complexity. The question which naturally comes to mind is “why should a single task be splitted to cause problems ?”. Also a low performance rate is expected.

In TractorS these tasks are combined in a single parallel scheduler which makes matters faster and easier to program and control. One should not worry about incomplete tasks; it is controlled automatically, i.e either the whole transaction is committed or aborted. Furthermore the unsafe-commit and cancel processes help in releasing locks and facilitating other concurrent transactions. The case study is fully developed and programmed in chapter 4. We refer the readers to that chapter for more information.

3.2.5 Transaction Termination

Two types of recovery from failures are talked about in the literature, undo and logical undo or compensation. While undo means restoring databases- once a transaction has failed- to the state that existed prior to the start of the transaction, compensation means restoring databases to a consistent state, not necessarily the one that existed prior to the start of the transaction.

In the execution of nested transactions in TractorS we distinguish between two different kinds of action completion:

1. unsafe-commit that can be undone by compensating actions. Component databases retain their own logic of commitment, however, additional information is logged (in the sense of sagas) at the MDBS level to guarantee distributed functionality. We call these logs cancel-logs because they are used in probable cancel processes.
2. *safe-commit* allows TractorS to discard unsafe action logs when safe breakpoints are reached. Safe commits are used to reduce the log overhead. Currently they can only be requested at the end of top-level transaction.

In conventional nested transaction models where transactions and subtransactions must wait for completion of each other to release data items, the negative impact on performance could be considerable. Particularly in a multidatabase environment it appears crucial to omit blocking distributed resources when transactions can have long durations. The unsafe commit allows us to release resources but requires keeping unsafe logs to be able to roll back to certain safe breakpoints. The amount of overhead that this type of log and its management causes is aimed to be minimal, but depends on the application. Users should be aware of the fact before deciding to use the system.

These two types of commitment have as counterparts two different types of recovery: we say a transaction is *undone* if it fails and the databases are restored; if it is *canceled* its effects are logically removed after an unsafe commitment. As a rule, safely committed transactions cannot be canceled. Furthermore, we do not allow users to explicitly cancel unsafely committed transactions since this would give them the ability to corrupt the TractorS recovery mechanism. If a transaction is canceled, all its dependents should also be canceled. Similarly a transaction does not safely commit unless all its dependents have committed safely.

In a *cancel* procedure, a single compensating transaction is formed, executed, and committed for all compensatable subtransactions in the context of a failed top-level transaction. We have expressed assumptions in chapter 6 to make this process feasible. For instance for each compensatable transaction there must be a unique compensating counterpart with the same number and type of arguments in the same order. Whether the process is expensive or not depends on the application.

A version of the well-known two-phase commit (2PC) protocol together with a timeout mechanism is tailored to TractorS for safe global commitment. The main disadvantage of the conventional scheme is the amount of messages communicated between component databases. We can reduce this figure by proposing a *dynamic group* for global synchronization. The top-level scheduler and the committed
atomic transactions which do not commit unsafely join the group. At this stage all delegation activities are terminated. Due to the self-responsibility of delegated transactions a particular actor called *delegatee* exists at each site which joins the global synchronization group, acts as an atomic transaction in the group and as a top-actor in that site, to synchronize local commitments with the global top-actor. In the vote-request stage the top-level transaction broadcasts a message to members of the group. If all of them receive the message, they are still alive and the commit activity follows. Otherwise the active ones are aborted and the already unsafe-committed ones canceled. All schedulers other than the top-level one are actually terminated after reaching the commit stage because they are done with their job (scheduling and following their subtransactions) and have not accessed database items to need global synchronization. The logging mechanism we use for recovery from crashes is similar to the one described in [29] which supports semantic-based operations and logical undo needed in TractorS.

### 3.2.6 Delegation

The term "delegation" has been used with different meanings in the literature [30]. In TractorS, transactions are delegated to a foreign site rather than to another ongoing transaction. The target site decides about the transaction which becomes responsible for itself.

Delegation occurs if the transaction falls in the domain of expertise of the target site which is different from that of the delegator so that the delegator site is not able to decompose the transaction and handle it. It however has enough knowledge to identify at least one site that has more expertise in that domain. Users may provide such information and the sites to try. The subtransaction gets aborted if no site could be located to handle it. Delegated transactions may commit unsafely like ordinary subtransactions but cannot commit safely. Depending on the semantics of the transaction, the (final) target site decides about its compensatability. If the delegator decides to abort later, it sends abort messages to all delegated transactions which will cause them to be undone or canceled. The delegator cannot terminate unless the delegated transaction terminates.

The main point in delegation is that the delegator site is not able to handle the transaction and therefore will not be held responsible for it. An important rela-
tion between the delegator and the delegated transaction is commit dependency. This is needed since on one hand the top-level transaction needs the results of the delegated transaction to commit and should not abort without undoing the effects of it (compensation helps here to release resources), and on the other hand the delegated transaction is not an independent task to be allowed to terminate independently. It is part of a global problem to be solved as a whole. For example in our case study the purpose of a personal loan is to facilitate the home loan. If the latter fails there is no point to let the applicant keep the personal loan.

There are two types of transactions responsible for others in TractorS, namely top-level and delegated transactions. Since atomic transactions are the only ones accessing objects, responsibility means following up such subtransactions until they end properly.

Delegation can be simple or cascading. Even if a delegated transaction as a whole is passed between several sites, it is considered simple. The result is communicated directly to the original delegator without involving the intermediate sites any more. As an example the target site for the personal loan subtransaction in the case study may not provide the service and prefer to delegate it to another site. There is no point to involve the former in the process. The case is effectively equivalent to delegating the subtransaction to the latter. Notice however that the foreign acquaintances knowledge source includes only meta-knowledge and cannot predict what will happen at foreign sites. Users also have the choice to specify target sites for a delegated transaction.

Cascading delegation happens when the delegated transaction gets decomposed and portions of it are further delegated to other sites. This case needs special treatment because results of different types, from ordinary as well as delegated subtransactions, should be combined and sent to the original delegator. The two cases are also distinguished due to the fact that only simple delegation can be implemented in the absence of transaction planner and decomposer, where advanced users program their decomposed transactions.

In summary, important characteristics of delegated transactions are:

- a delegated transaction becomes responsible for itself, i.e. it produces correct results at the target site, while the top-level transaction is responsible for ordinary subtransactions. A delegated transaction however should not
commit safely due to the fact that it is part of a global problem and its result is considered useless if that problem cannot be solved.

- cascading delegation occurs whenever a delegated transaction gets decomposed on the way and possibly portions of it are further delegated. It cannot be supported in the case where users directly program their decomposed transactions, due to lack of automatic decomposition power.

- to minimize communication costs, a multiply delegated transaction (as a whole) communicates the result directly to the original requester without involving the intermediate sites any more.

- a particular actor called delegatee exists at each site which joins the global synchronization group, acts as an atomic transaction in the group and as a top-actor in that site to synchronize local commitments with the top-level transaction.

- mechanisms are provided in the high level scripting language for TractorS to allow users specify sites for delegated transactions.

In some applications it may occasionally be helpful to let certain delegated transactions commit safely in an independent manner in order to use their results in subsequent transactions. In TractorS terms, in cases where a top-level transaction aborts it may be attractive not to compensate certain delegated transactions committed unsafely. This issue is not considered in the thesis and constitutes part of future work.

3.2.7 Fully Decomposed Transaction Trees

In TractorS each of the sites may decompose transactions and further delegate portions thereof. Therefore transaction processing at one site and further decomposition at another site can indeed proceed in parallel. During execution, each node of the tree has a status recording whether it has been scheduled, aborted, unsafely committed and so on, or whether it is blocking and awaiting a signal via a dependency arc. Besides subtransactions, other messages are also passed between sites. For instance, safe commitment messages, cancel messages or timeout
safely committed
unsafely committed
executing
to be scheduled

Figure 3.3: A multi-level scheduling tree

messages, and global synchronization messages are part of this communication between different sites.

In any event, after all decomposition is completed, we can view the entire decomposition of one top-level transaction as one multi-level tree (figure 3.3) mapped to the sites of the network. This tree distinguishes between those nodes that are leaves, such as $l$ in the transaction tree at site $s_1$, and roots of nested transactions at another site$^2$ such as $r$ in the transaction tree at site $s_2$.

For this tree-like decomposition, we guarantee that the delegation process is acyclic as we associate the originator path, i.e., a site identifier sequence, with

$^2$Additional information may be needed to convey the fact that delegated calls may time-out.
each delegated subtransaction. This information is then used for dynamic cycle
detection to prevent cyclic submissions. This is the responsibility of the par-
ticular actor delegatee at each site. It checks the path and looks for its home

cite identifier. If found, it aborts the transaction right away to stop the cycle.
Depending on its type, the parent may try other sources.

From the set of meta actors, only schedulers remain in a fully decomposed
transaction tree, mapped to the sites of the network. Delegators ultimately re-
sult into decomposed portions which are a combination of atomic transactions
and schedulers. Autonomy of components of the system implies that atomic
transactions may not be actors. Actually the leaves of the tree can be any sort
of atomic transaction as long as the corresponding local transaction manager
properly communicates for global synchronization.

3.2.8 Global Concurrency Control and Recovery

A major goal in a CIS is to allow maximum possible concurrency. A con-
currency control mechanisms is needed to ensure integrity of the system. Also a
transaction in CIS, like in any other system, may suffer from failures. A recovery
mechanism is necessary to preserve integrity of the system against incomplete
transactions and to make sure that the results of committed ones are not lost.
Here we discuss the concurrency control and recovery methods which best fit
TractorS's necessities.

Serializability has been the most common correctness criterion used for con-
currency control. It does not work well in complex distributed environments
where autonomy of each site is a primary concern since it is originally introduced
for centralized databases and is centralized in nature. The fact that two global
transactions may conflict even if they do not access any common data item makes
the case complex. Such conflicts are introduced by effects of local transaction on
global ones. On the other hand full autonomy of sites means that the correspond-
ing DBMSs need not be aware of the existence of each other. Hence their local
concurrency control mechanisms could hardly be used for global purposes with-
out violating local autonomy. The proposed concurrency control algorithms for
such environments either violate local autonomy or allow low degree of concur-
are proposed which guarantee global serializability, take into account the effects of local transactions on global ones (ignored in a number of other works), and respects local autonomy of sites.

TractorS supports non-serializable schedules. CIS is a generalization of multidatabase systems towards supporting more diverse and heterogeneous applications. In CIS the component systems to be synthesized may be not only database systems, supporting the ACID properties, but also related expert systems and file systems. This suggests the use of a more flexible transaction model for CIS. Transactions in this environment are more complicated than multidatabase transactions but the concurrency control problem is similar in both due to autonomy of components. Here we summarize the well-known non-serializable concurrency control methods for multidatabase systems and comment about their compatibility with TractorS and CIS environment.

A popular non-serializable correctness criterion proposed for multidatabase environments is quasi-serializability [63]. The restrictions imposed by quasi-serializability are:

1. there can be at most one subtransaction at a time per site for each global (top-level) transaction. This is natural in TractorS due to the fact that all sites have decomposition power. Hence multiple concurrent subtransactions are packed together and submitted or delegated to other site as a whole. They have more knowledge about transactions in their expertise domain and can decompose them better.

2. there should be no relationship between data items at different component databases. As a result, replication of data is restricted, i.e only those data items that are not updatable by component databases are allowed to be globally replicated in quasi-serializability.

3. quasi-serializability also restricts value dependency [63], [70]. This restriction is observed in TractorS due to the following:
   (a) different top-level transactions cannot have value dependency.
   (b) the consumer cannot have conflicting access to the source of the value for that purpose.
(c) if the producer aborts, the consumer is always either undone or canceled. It is important to notice that value dependency is only allowed in parallel schedulers which commit only if all their children commit.

Another well-known non-serializable correctness criterion proposed for multi-database environments is *multidatabase serializability* [71]. It is similar to quasi-serializability in that both require all local histories to be serializable and the global history be conflict-equivalent to a serial one. Also both assume that a global transaction may not submit more than one subtransaction to any single component database. A disadvantage of the two is that they ignore local transactions. Besides the commonalities with quasi-serializability, the basic assumption in multidatabase serializability is that the component databases are totally autonomous, and can hardly communicate and cooperate in executing global transactions. Furthermore, it is assumed that only syntactic information is available about the global transactions. The class of schedules that are multidatabase serializable is identical to the class of schedules that are quasi-serializable [71].

Another concurrency control method which is claimed to be better adapted to interoperable database systems is the *Multidatabase Value Dates* scheme [72]. It requires the component transaction managers not to utilize two-phase locking. This assumption is not appreciated in the flexible environment of CIS and is a threat to local autonomy.

Other extensions of serializability are also proposed. The SRC ( Serializable with Respect to Compensation) [73] defines a serializable schedule in a multi-database environment in which subtransactions of a global transaction are compensatable, retrievable, or pivot (neither of the other two). If a schedule is SRC then each transaction sees a consistent database state. A concurrency control protocol is proposed that irrespective of the one followed by the local DBMSs, ensures that schedules in which every global transaction has terminated are SRC. The protocol involves insertion and deletion of edges from a transaction-site graph.

None of the above methods has enough flexibility for the CIS environment. The closest one is quasi-serializability but the fact that it ignores local histories and restricts data replication makes it unrealistic with regard to some applications. Further research is needed to develop an appropriate concurrency control mechanism for the complex CIS environment. We provide some linguistic support
and leave the choice to the application. In the formal semantics we use quas seri alizability to prove correctness of the system. One should notice that quasi serializability is proposed for multidatabase environments where the autonomous component databases may have any local concurrency control mechanisms. The probability of global deadlock in such an environment depends on the application and is not predictable. TractorS uses a flexible timeout mechanism whereby programmers can attach timeout periods to each process.

Recovery in TractorS is similar to the scheme proposed in [29]. Our extra construct is delegation which does not affect recovery because it finally results in ordinary subtransactions executed at some sites. One should notice that local autonomy is most respected in our model, and therefore local mechanisms work as usual. The extension to conventional nested-transactions which affects recovery mechanism is making it open-nested, i.e adding the notion of compensation. This concepts adds two operations: unsafe-commit and cancel. A new type of log, called cancel-log, is introduced to serve the two operations. It is kept at the TractorS layer at the site running a top-level transaction and is handled by particular routine when unsafe-committing or canceling subtransactions. The unsafe-commit process does not impose extra recovery actions other than updating the cancel-log due to the fact that it is the same as safe-committing a particular (compensatable) type of top-level transaction. The cancel procedure is similar because it is a top-level transaction by itself. The compensating transaction should be properly formed, executed, and its commitment ensured.

An important point to clarify is failure of a cancel process. As a top-level transaction it may face any type of failures while in progress. The main question is what to do with some of its subtransactions at some sites which are actually done. Should they be unreasonably rolled back and restarted due to the failure of others? For instance assume that the final synchronization of a cancel process fails. It would not be feasible to restart all subtransactions which mostly needed to commit independently. There are possible solutions to this problem. One is to have a special synchronization process for cancel transactions to finalizes certain subtasks upon their completion. Another solution is to commit the subtransactions which are totally done in an unsafe manner (it cannot be called safe-commit because the top-level task is still in progress) without further cancellation (do not
allow repeated cancellations). The second choice fits better with TractorS environment and gets executed automatically. It however needs the compensating transactions to be compensatable as well. It is reasonable to assume that compensation mostly involves pairs of subtransactions which are counterparts of each other. Notice that this is not a necessity but is for improving performance by avoiding unnecessary actions. Results are correct in any case.
Chapter 4

Linguistic Facilities

Part II

The Scripting Language for Tractors

Two interaction scenarios are considered in the proposed architecture for Tractors: automatic query decomposition done by the transaction planner and decomposer, and the programming of transactions and subtransactions done by advanced users in the Tractors IDE. We do not expect the language to further develop in future and provide facilities for the automatic transaction decomposition as well. The high-level scripting language for Tractors is described in this and the next chapter. This chapter introduces the language from the interface point of view. We describe some of the features and show what is offered and how it can be used.

Advanced users, called programmers hereafter, are the ones who know how to put together distributed applications. Applications are programmed as Sather classes which inherit from Tractors library. Programming in this environment is different from conventional programming, although the full power of Sather can be used. Programmers need to know things from a very high-level point of view. A minimal set of constructs need to be utilized in a clearly defined format to put together and run a transaction as a hierarchy of classes. Tractors hides certain details of the execution of the script program from programmers. As an example, a programmer may define a subtransaction as compensatable but details of committing one unsafely and committing it if the top-level transaction fails is hidden. Also one cannot change the type of the existing transactions provided by others or the system. Delegation is, however, different. Although the choice of the site depends on the network configuration and the semantics
Chapter 4

Linguistic Facilities

Two interaction scenarios are considered in the proposed architecture for CIS: automatic query decomposition done by the transaction planner and decomposer, and the programming of transactions and subtransactions done by advanced users in a high-level scripting language supported by TractorS (see figure 3.1). Only the second choice is developed in the thesis. We expect the language to further develop in future and provide facilities for the automatic transaction decomposition as well. The high-level scripting language for TractorS is described in this and the next chapter. This chapter introduces the language from the interface point of view. We describe some of the features and show what is offered and how it can be used.

Advanced users, called programmers hereafter, are the ones who know how to put together distributed applications. Applications are programmed as Sather classes which inherit from TractorS library. Programming in this environment is different from conventional programming, although the full power of Sather can be used. Programmers need to know things from a very high-level point of view. A minimal set of constructs need to be utilized in a clearly defined format to put together and run a transaction as a hierarchy of classes. TractorS hides certain details of the execution of the script program from programmers. As an example, a programmer may define a subtransaction as compensatable but details of committing one unsafely and canceling it if the top-level transaction fails is hidden. Also one cannot change the type of the existing subtransactions provided by others or the system. Delegation is, however, different. Although the choice of the site depends on the network configuration and the semantics
of the application, mechanisms are provided for programmers to check the meta-
knowledge sources, decide which sites are relevant to carry out a delegated task,
and define the hierarchical structure of the transaction tree accordingly.

4.1 Sather Classes

The two main components involved in programming and running a typical
transaction in the scripting language are:

- Sather classes which is the main framework for putting tasks together. This
  part is briefly described here.
- TractorS library which supports the necessary types of constructs such as
  schedulers, value dependency, etc, to be inherited by the classes defined by
  the programmers. This part is described in the next subsection.

Sather is a class-oriented language. Programs appear as a collection of classes
put together in a hierarchy. It is especially aimed at complex, performance-critical
applications. The language has parameterized classes, object-oriented dispatch,
statically-checked strong typing, separate implementation and type inheritance,
multiple inheritance, garbage collection, iteration abstraction, higher-order rou-
tines, exception handling, constructors for arbitrary data structures, precondi-
tions, postconditions, and class invariants. Most of these ideas are used in the
implementation of the current prototype. Programmers however do not need to
know them in detail. Instead they need to be able to program simple classes
which inherit the existing ones, and put together distributed transactions as a
hierarchy of classes.

The “Hello World” program in Sather looks like the following:

class HELLO is
    main is
        -- Print "Hello World" on stdout.
        OUT:s("Hello World").nl;
    end; -- main
end; -- class HELLO
One should notice the use of the keyword “is” for class and method definition. A class is either aimed to be inherited, or executed, or both. Executable classes should have a routine called “main” which is a common wrapper which calls other routines of the same or other classes. All inherited routines become part of the class. Routines of other classes can be called by using the “::” notation. For example, the statement OUT::s(“Hello World”).nl is actually calling the routines s (for string) and nl (for new line) from the class OUT. Multiple routines are called in the same statement by the dot notation. As another example one may replace the OUT statement above by the following: OUT::s(“Sather Says ”).s(“Hello World”).nl;

Besides methods, Classes have other parts including constants and attributes. We illustrate these by adopting a portion of a simple example from the Sather manual, namely the class STACK, with some changes.

class STACK{T} is
    -- General purpose stack of objects of type T.
    attr arr:ARRAY{T}; -- Array holding stack elements.
    attr ssize:INT; -- Number of elements in stack=insertion loc.
    const initial_size:INT:=5; -- Start size of the stack array.
    create:SAME 1s 'SAME' used so it works when inherited.
        -- A new stack.
        res := new; -- 'res' is returned at routine completion.
        res.arr := ARRAY{T}::new(initial_size);
    end,

    push(e:T) is
        -- Insert the element 'e'.
        if ssize >= arr.asize then -- Resize if stack area is full.
        end;
        arr[ssize] := e; -- Put new element at the top.
        ssize := ssize + 1; -- Increase the stack size.
    end;
Chapter 4. Linguistic Facilities

pop:T is
   -- Pops off the first element.
   ssize := ssize - 1
   res := arr[ssize];
   arr[ssize] := void;
end; -- class STACK

This class has a parameter T which is the type of stack members. It could be any type. Type in Sather is either elementary (INT, REAL, BOOL) or any class. The keywords attr and const stand for defining attributes and constants respectively. The attribute arr in class STACK is an array which holds members of type T. Notice that ARRAY is a class in Sather which provides facilities for array handling. Similarly LIST is a class which provides facilities for handling lists. Two sizes are defined in the class STACK: one is ssize which is the size of a stack and changes at run time. This is why it is defined as an attribute. The second one, initial.size, is a constant which cannot be changed.

A class needs a create() routine to allocate instances of it (objects). The keyword new is used in this routine to get a new object which is of type SAME for inheritance reasons (e.g stacks with different element types). The new object is returned using the keyword res. If any other classes are used for defining attributes or constants, they should be explicitly created and returned as part of the keyword res in the routine create.

Statements in Sather have a Pascal-like syntax. For example the if statement in the routines push() and pop() are different from Pascal by not having a begin statement. A general syntax for it is if ... then ... else ... end:. The assignment statement also uses the “:=” notation.

The routine push() compares the attribute ssize with the attribute asize, defined in the class ARRAY, attached to the attribute arr to check for the size of stack. If not big enough, it calls the routine extend from the class ARRAY, attached to the attribute arr, to double the size of the stack. It then adds the element to the stack, increments the size of stack, and returns. The routine pop() is obvious.
Inheritance is simply done by just using the name of a class. All parts are inherited unless they are redefined. To illustrate we define a new class MY_STACK which redefines the initial size and the routine pop and defines the parameter of the class STACK as string (defined as the class STR in Sather). It also adds the routines is_empty and main. The rest remains the same. Notice that STACK does not have the routine main and hence is only useful for inheritance or variable definition in other classes (e.g s: STACK{INT}).

```sather
class MY_STACK is
  T: STR;
  STACK{T};
  const initial_size:INT:=100; -- Start size of the stack array.

  is_empty:BOOL is
    if ssize = 0 then
      res := true;
    else
      res := false;
    end;
  end; --is_empty

  pop:T is
    -- Pops off the first element or 'void' if empty.
    if is_empty then
      res := void;
    else
      ssize := ssize - 1
      res := arr[ssize];
      arr[ssize] := void;
    end;
  end;

  main(arg: ARRAY{STR}) is
    -- assume there are three arguments.
Chapter 4. Linguistic Facilities

s: STACK{T} := create;
s.push(arg[1]).push(arg[2]).pop.push(arg[1]);
OUT::s('Number of elements of stack s is ').n(s.ssize).nl;
end; --main
end; -- class MY_STACK

We briefly explain the code. The class STACK is inherited, the constant initial.size is redefined to 100, and the routine is_empty is added to check whether the stack is empty or not. Notice that the variable ssize is used in this routine without being defined in the class (is inherited). The routine pop is slightly changed to check if the stack is empty and return void (the original class from the manual does these things, but in a slightly different way).

The routine main creates a stack, pushes two elements, pops one of them, and then pushes a third one. The elements pushed and popped are the arguments to main. As a result the stack would have two elements. Notice how routines s (for string) and n (for integer) are called from the class OUT to output the size. The output would be: Number of elements of stack s is 2.

4.2 TractorS Library

4.2.1 Class Hierarchy

A number of classes are designed to assist programming in the scripting language. Here we describe the class hierarchy from the interface point of view. Detailed information about the implementation of the classes is left for the next chapter where major design and implementation decisions are described and some sample Sather code is provided.

Figure 4.1 gives a pictorial view of TractorS library class hierarchy. Inheritance is upward in the figure, e.g the class HOME_LOAN inherits the classes SERIAL, etc. while they all inherit the class SCHEDULER. Some low-level classes are not shown in the figure.

A sample of TractorS library routines are also summarized in table 4.1. The purpose of this table is to give a picture of the library rather than its description.
Chapter 4. Linguistic Facilities

Table 4.1: A sample of Tractors library classes

<table>
<thead>
<tr>
<th>layer</th>
<th>inheritance hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>user-defined</td>
<td>HOME_LOAN</td>
</tr>
<tr>
<td>scheduling</td>
<td>SERIAL, PARALLEL, SERIAL_ALTERNATIVE, PARALLEL_ALTERNATIVE, SCHEDULER</td>
</tr>
<tr>
<td>delegation</td>
<td>DELEGATE</td>
</tr>
<tr>
<td>atomic transactions</td>
<td>ACTION, COMPENSATABLE_ACTION</td>
</tr>
<tr>
<td>transactional events</td>
<td>CANCEL, UNSAFE_COMMIT</td>
</tr>
<tr>
<td>communication &amp; synchronization</td>
<td>SATHER_PVM</td>
</tr>
<tr>
<td></td>
<td>MESSAGE_PASSING</td>
</tr>
<tr>
<td></td>
<td>INITIALIZE, TRACE</td>
</tr>
</tbody>
</table>

Figure 4.1: Class hierarchy of Tractors library

It shows some of the high-level routines in the classes hierarchy.

In the following we describe briefly the functionality of the Tractors library classes in a top-down order:

1. schedulers are realized by five classes. The class SCHEDULER provides the common aspects of them all, and is inherited by other classes. Each one of the other four is dedicated to one of the four types of schedulers (e.g. the class PARALLEL supports parallel schedulers). Since a top-level transaction in Tractors is always an scheduler (except for a trivial non-nested single task transaction), classes at this level inherit from ones at lower levels. Also global synchronization is done as part of this layer.
Table 4.1: A sample of TractorS library classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Routine Name</th>
<th>Brief Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEDULER (general)</td>
<td>set_hosts()</td>
<td>set user-defined hosts for subtransactions.</td>
</tr>
<tr>
<td></td>
<td>find_hosts()</td>
<td>find suitable hosts for subtransactions.</td>
</tr>
<tr>
<td></td>
<td>spawn_sub()</td>
<td>spawn a subtransaction at a host.</td>
</tr>
<tr>
<td></td>
<td>name()</td>
<td>specify name of a subtransaction.</td>
</tr>
<tr>
<td></td>
<td>arg()</td>
<td>specify an argument of a subtransaction.</td>
</tr>
<tr>
<td></td>
<td>is_a_sub()</td>
<td>if the argument is a subtransaction.</td>
</tr>
<tr>
<td></td>
<td>decision()</td>
<td>make decision when scheduler is done.</td>
</tr>
<tr>
<td></td>
<td>commit_protocol()</td>
<td>global commit protocol.</td>
</tr>
<tr>
<td></td>
<td>get_group_size()</td>
<td>find out the size of a given group.</td>
</tr>
<tr>
<td>SCHEDULER (specific)</td>
<td>exec()</td>
<td>execute subtransactions (one for each type).</td>
</tr>
<tr>
<td>e.g.</td>
<td>get_next_result()</td>
<td>receive result of the next subtransaction.</td>
</tr>
<tr>
<td></td>
<td>get_all_results</td>
<td>receive result of all subtransactions (parallel).</td>
</tr>
<tr>
<td>SERIAL</td>
<td>set_produced_args()</td>
<td>set the arguments for the next (serial) child.</td>
</tr>
<tr>
<td></td>
<td>get_probe()</td>
<td>check for a particular message and receive it.</td>
</tr>
<tr>
<td></td>
<td>abort_others()</td>
<td>abort the rest (choice schedulers).</td>
</tr>
<tr>
<td>DELEGATE</td>
<td>delegatee()</td>
<td>accept and handle delegated transactions.</td>
</tr>
<tr>
<td></td>
<td>service_assign()</td>
<td>assign a service to a host.</td>
</tr>
<tr>
<td></td>
<td>is_host()</td>
<td>if the argument is an actual host.</td>
</tr>
<tr>
<td></td>
<td>add_to_cluster()</td>
<td>add a host to a cluster.</td>
</tr>
<tr>
<td></td>
<td>which_cluster()</td>
<td>to which cluster a host belongs.</td>
</tr>
<tr>
<td></td>
<td>cluster_members()</td>
<td>return a list of cluster members.</td>
</tr>
<tr>
<td></td>
<td>which_hosts()</td>
<td>hosts which provide a particular service.</td>
</tr>
<tr>
<td></td>
<td>is_close_acquaint()</td>
<td>whether a host is a close acquaintance.</td>
</tr>
<tr>
<td>ACTION</td>
<td>pass_values()</td>
<td>pass values to the consumers.</td>
</tr>
<tr>
<td></td>
<td>get_values()</td>
<td>get values from the producer.</td>
</tr>
<tr>
<td></td>
<td>pass_signal()</td>
<td>pass commit signal to the dependent tasks.</td>
</tr>
<tr>
<td></td>
<td>get_signal()</td>
<td>get signal from the producer.</td>
</tr>
<tr>
<td></td>
<td>decision()</td>
<td>make decision when the task is done.</td>
</tr>
<tr>
<td>CANCEL</td>
<td>main()</td>
<td>govern top-level task of the cancel process.</td>
</tr>
<tr>
<td></td>
<td>extract_log()</td>
<td>extract info about top-level from cancel-log.</td>
</tr>
<tr>
<td></td>
<td>form_comp_trans()</td>
<td>form the compensating transaction.</td>
</tr>
<tr>
<td>SATHER_PVM</td>
<td>pvm_routines</td>
<td>interface to PVM (similar routines as in C).</td>
</tr>
</tbody>
</table>
Two of the routines appearing in the class SCHEDULER are particularly important. They are \texttt{name()} and \texttt{argument()} which are used frequently in the user-defined classes to specify name and arguments of subtransactions.

2. the next level is delegation. The class \texttt{DELEGATE} supports the task. It also provides meta-knowledge about sites and services and how to direct subtransactions to expertise sites. The information is basically about sites, clusters, domains of expertise and their relationship. The class provides routines for programmers to check the meta-knowledge, decide which sites are relevant to a particular delegated transaction, and specify one or more sites for each one (see below).

3. atomic transactions are supported by the two classes \texttt{ACTION} and \texttt{COMPENSATABLE\_ACTION}. They provide all types of functionalities required for creating and communicating with atomic transactions. The script language for TractorS provides flexibility for the programmers to utilize any kind of atomic transaction supported by local transaction managers in CIS, as long as they properly communicate for global synchronization. This issue is clarified in the next chapter.

4. classes handling unsafe-commit and cancel appear at the next level because they support primitive transactional events. They are \texttt{UNSAFE\_COMMIT} and \texttt{CANCEL}. One should notice that \texttt{safe-commit} and \texttt{undo} are taken care of by components of CIS. The global synchronization is done in the class \texttt{SCHEDULER}.

5. the \texttt{SATHER\_PVM} and related classes appear at the lowest level of TractorS library hierarchy. The classes are the interface between Sather and PVM, a well-known public domain interprocess communication package, and add distributed programming facility to Sather. The interface is described in the next chapter.

At the top most level appears the application layer. Programmers write their classes here and inherit from the library. As an example the classes related to the case study developed in section 3.2.4 are programmed later in this chapter.
4.2.2 Unsafe_commit and Cancel

As examples of the classes in TractorS library, we briefly introduce the classes UNSAFE_COMMIT and CANCEL here and the class DELEGATE in the next subsection. These classes are chosen due to their exceptional semantics.

Committing a subtransaction unsafely has two basic parts:
(i) logging enough information for the possible cancel process.
(ii) committing the subtransaction and releasing locks.

The second part is the same as safe_commit which is done at the components of CIS. Hence, the class UNSAFE_COMMIT is basically concerned with managing the cancel-log. The cancel-log contains three types of records:

1. a begin record for each top-level transaction which includes its unique transaction identification. Besides the transaction id, the begin record includes the hierarchical structure of the top-level transaction. This part shows the complete parent-child structure. The type of schedulers and the name of the leaf nodes are recorded. This record is particularly important for ordering the compensating subtransactions. One should notice that although two such compensatable subtransactions may belong to some parallel schedulers, they might have a common serial ancestor in the hierarchy and therefore their order of commitment is important. This is why the type of the parent is not good enough and the types of ancestors of subtransactions (appearing in the hierarchy) are also required.

2. one record for each unsafely committed subtransaction which includes the following information:
   - the name and the identification of the subtransaction. This item is used to look up its compensating counterpart.
   - the type and the identification of the parent of the subtransaction. This is necessary because multiple copies of the same subtransaction may possibly appear in the same top-level transaction.
   - the transaction identification of the corresponding top-level transaction. This is necessary because multiple copies of the same subtransaction (e.g. withdraw), belonging to different top-level tasks could be committed and logged in a period of time.
• the host that committed the subtransaction unsafely.

3. an end record for each top-level transaction which includes its unique transaction identification.

All the records belonging to a particular top-level task appear between its begin and end records on the log. One should notice that the records in a typical log belong to multiple concurrent top-level transactions in no particular order. However, the fact that all the records belonging to a particular top-level transaction appear between its begin and end records helps extracting related ones.

The class CANCEL provides mechanisms for cancel processes. A single compensating transaction is formed, executed, and committed for all compensatable subtransactions of a failed top-level transaction. High-level routines of this class appear in table 4.1.

As an example the subtransaction offer-personal-loan in the case study is a compensatable transaction. If the home loan process is not finalized, the compensating transaction cancel-personal-loan will cancel its effects. To do so, the “main” routine in the class CANCEL calls the extract_log() routine and passes the id of the failed top-level transaction to it to get all compensatable subtransactions belonging to that task. The result is then passed to the routine form_comp_trans() which returns a transaction tree. The transaction is then executed. If anything goes wrong, the programmer is informed and enough information is passed to him to execute the transaction again.

4.2.3 Delegation

Delegation is different in the two interaction scenarios of the proposed architecture for CIS. We have only implemented the programmable case, i.e the case where there is no automatic decomposition and programmers should do the task themselves by programming their fully decomposed transactions in the high level scripting language. In such a case cascading delegation is impossible to implement due to the lack of decomposition power. Simple delegation is however relevant, i.e a site may delegate a task to a foreign acquaintance and the target site may forward it to a third one if it cannot be handled there.
Chapter 4. Linguistic Facilities

The class DELEGATE is designed and implemented to support simple delegation. Delegation has two sides: one of the client and the other at the server (target) site. The main job at the client is to interact with the programmers to find appropriate foreign hosts, forward the transaction to them, get the result back, and decide how to proceed. At the target site an interface routine called delegatee exists as part of the scripting language which accepts delegated transactions, handles the task, and sends the outcome back to the delegator. If the target site forwards the transaction to another one, the latter sends the result directly to the original delegator. Forwarding a delegated transaction may occur multiple times. Cycles are prevented by checking the ids of sites and ancestors of subtransaction which appear as a parameter to delegatee. A user-defined timeout mechanism aborts the ones which introduce excessive delays.

The direct communication between the original delegator and the final target site is established by defining a dynamic group for each delegation process. Such a group is a variable size set of tasks that may join or leave the group at run time. The system keeps track of the members and provides facilities to communicate between them. Tasks communicate by joining the group, broadcasting messages, receiving relevant ones, and waiting for others to join. To eliminate intermediate sites in a delegation process, a special type of group with only two members is defined. The first member is the original delegator of a particular task, which cannot be removed. If the task is further delegated by the target site, the new site asks for joining the group. The system removes the intermediate site and adds the new member. At the end, the result is communicated between the two remaining members.

Various routines of the class DELEGATE support delegation (see table 4.1). Programmers may request the system to looks up the sites to perform a task via the which_hosts() routine. The hosts are passed to is_close_acquaint() to ascertain whether it is a close acquaintance of the current site. If the answer is false then the routines which_cluster() and cluster_members() are called to find out about possible sites to handle the process. All these information is passed to programmers to make the decision and define the proper sites for each delegated transaction. As a simple example, we show how a programmer can write a script program to find out which hosts may be able to perform the task “my_task”
Chapter 4. Linguistic Facilities

below.

class MY_CLASS is

  DELEGATE;

    main is
      which_hosts('my_task');
    end; --main

end; -- class MY_CLASS

A class is defined, DELEGATE is inherited, and a main routine is defined which calls the routine which_hosts. This routine in turn calls other routines and outputs the appropriate hosts. For instance one of the routines called is is_close_acquaint () which specifies the type of the host. A typical output of the above class is:

Foreign_acquaintances providing the service "my_task" are barnard, dali.

Delegated subtransactions are passed to the interface routine delegatee at foreign sites which will follow up and communicate the result. An example of delegation is the personal-loan subtransaction in the case study which has a different nature from the home loan. The bank offering the home loan and/or its close acquaintances may not offer personal loans. If not the subtransaction is delegated to some other banks chosen by the applicant.

4.3 End-user Interaction

End-users may execute transactions already programmed and compiled by programmers. In this type of interaction, the structure of the transactions, their subtransactions, and the corresponding sites cannot be changed. End-users may type the name and parameters of subtransactions which already exist in TractorS bin directory. For instance one may type withdraw acnt.123 1000.00 to mean “withdraw $1000.00 from account number acnt.123”. The type of output that end-users get is less descriptive than the one programmers get. The former is only informed of the major actions being taken place, while the latter is provided
with descriptive information about progress of subtransactions in order to make him able to follow up.

As a simple example, we show how a money transfer transaction which withdraws from an account and deposits the funds into another account, possibly located at another site, is executed and what it provides as output. The end user types the name and parameters of the transaction as:

```
transfer from_acnt to_acnt 1000.00
```

If successful, the result is:

```
1 transfer#260: BEGIN TOP_LEVEL TRANSACTION.
2 transfer#260: withdrawal of $1000.00 from from_acnt successful.
3 transfer#260: deposit of $1000.00 into to_acnt successful.
4 transfer#260: safe_commit successful.
5 transfer#260: END TOP_LEVEL TRANSACTION.
```

Each line of the output starts with the name of the transaction together with its unique transaction id (e.g. #260). This guarantees that different concurrent transactions with the same name are distinguished. Line 1 states the start of the transaction. Lines 2,3 indicate that the subtransactions `withdraw` and `deposit` have been successful. Line 4 shows that the global safe-commit stage is reached and Line 5 states the end of the transaction.

A transaction may fail for various reasons. Let's try to withdraw an amount more than the account balance.

```
transfer from_acnt to_acnt 100000000.00
```

The result is:

```
1 transfer#262: BEGIN TOP_LEVEL TRANSACTION.
2 transfer#262: withdrawal failed (no enough balance).
3 transfer#262: undo successful.
4 transfer#262: END TOP_LEVEL TRANSACTION.
```

4.4 Programmer Interaction

Main users of TractorS are advanced users (programmers). They need more descriptive output messages to follow the transactions more closely. There are
three levels of *trace* switches that they can choose (explained how in the next chapter). We make the output one level more descriptive and execute the same *transfer* transaction. The transaction is actually programmed with the following assumptions:

- it is a serial scheduler, i.e the *withdraw* subtransaction is executed first, and the *deposit* subtransaction is executed only if the first one is successful.

- it is a distributed transaction. Accounts are located at different sites, *withdraw* is done by the host “arp” and *deposit* is done by the host “earth”.

The output of the transaction *transfer* for programmers is:

1. `transfer#267: BEGIN TOP_LEVEL TRANSACTION.`
2. `transfer#267: SERIAL scheduler.`
3. `transfer#267: host ‘arp’ is doing withdraw.`
4. `transfer#267: withdrawal of $1000.00 from from_acnt successful.`
5. `transfer#267: host ‘earth’ is doing deposit.`
6. `transfer#267: deposit of $1000.00 into to_acnt successful.`
7. `transfer#267: start global commit protocol.`
8. `transfer#267: global commit protocol successful.`
9. `transfer#267: safe_commit successful.`
10. `transfer#267: END TOP_LEVEL TRANSACTION.`

Line 2 states that *transfer* is a serial scheduler. Lines 3 shows that ‘arp’, an actual host, is doing withdraw, and the next line shows that it has been successful. Similar output is produced for deposit in lines 5,6. Lines 7,8 report the start and successful end of the global commit protocol, and the rest shows the successful end of the transaction.

The output lines appear gradually as the transaction proceeds. The purpose is to inform the programmers of the progress of subtransactions and help him to follow up and make possible changes and retry the task if it does not succeed. To illustrate, we show what happens if something goes wrong. Instead of the actual host ‘earth’, we choose ‘moon’ (a made up name) to do *deposit*. The result is:
Chapter 4. Linguistic Facilities

1 transfer#275: BEGIN TOP_LEVEL TRANSACTION.
2 transfer#275: SERIAL scheduler.
3 transfer#275: host 'arp' is doing withdraw.
4 transfer#267: withdrawal of $1000.00 from from_acnt successful.
5 transfer#275: 'moon' is an unknown host. Cannot activate deposit.
6 transfer#275: incomplete task; abort transaction.
7 transfer#275: undo successful.
8 transfer#275: END TOP_LEVEL TRANSACTION.

Now we show how such transactions and the flow of information between subtransactions are programmed. Programmers should be familiar with the component databases related to their applications. Also they need a elementary knowledge of Sather to be able to customize classes. As a convention we always use capital letters for names of classes and lower case letters for their instances (subtransactions which are actual objects). For example WITHDRAW means the class and withdraw means the subtransaction which is an instance of that class.

The simplest type of user-defined classes are ones for atomic transactions. We show how such a class for the withdraw transactions looks like and explain it here:

class WITHDRAW is
  ACTION;
  const me: STR := "withdraw";
  work: STR is
    THE APPLICATION DEPENDENT TASK IS PROGRAMMED HERE.
  end; --work
end; --withdraw

Programmers do three things here:

- inherit the class ACTION which supports atomic transactions and provides all related instance variables and routines (methods) including the main routine.

- customized the constant me which is used for name of the subtransaction in the output as already seen. The unique transaction id is also printed, but it is provided by the system.
provide the “work” routine in which the application dependent part of the
transaction is programmed. This is the only routine left for programmers
to provide.

The “work” routine cannot be pre-programmed because of its application de­
pendent nature. The system is however flexible enough to link with atomic
transaction written in other languages, as long as they properly communicate
for global synchronization. Programmers may call appropriate functions in the
“work” routines or use those type of transactions instead of their user-defined
classes for atomic transactions. Sather allows C functions to be called in any
class.

Atomic transactions are a simple case. The main interaction of users is via
user-defined classes for schedulers. There are three main differences with atomic
transactions:

• depending on the type of scheduler, one of the classes SERIAL, PARAL­
LEL, SERIAL_ALTERNATIVE, or PARALLEL_ALTERNATIVE
are inherited.

• the “work” routine here defines subtransactions and their arguments. To do
so, one should use the following syntax as used in the class TRANSFER.

    subtransaction.name(name).arg(argument)...

The keyword subtransaction is for clarity purpose. The routines name() and arg(), as already seen in table 4.1, are used to define the name and
arguments of a subtransaction. The former is used once and the latter as
many times as the number of arguments of the subtransaction.

• all sorts of things like site specification, value dependency, delegation, etc.
are also done in the “work” routine. For some of them, programmers should
define certain things. Others are done automatically. For instance the
programmer should specify the source and destination of values for value
dependency, but cancel is an automatic process.

As a simple example, the class TRANSFER is shown and described below.
Chapter 4. Linguistic Facilities

class TRANSFER is
    SERIAL;
    const me: STR := "serial";
    work is
        subtransaction.name("withdraw").arg(argv[1]).arg(argv[3]);
        subtransaction.name("deposit").arg(argv[2]).arg(argv[3]);
    end; --work
end;

Programmers must make sure to inherit the right class for the type of scheduler (SERIAL here), and defines the subtransactions (two in this case) in the "work" procedure. Only the name and arguments of each subtransaction should be provided as shown. For instance the first subtransaction of transfer has the name "withdraw" and its arguments are the first and the third arguments of transfer itself. The order is obviously important in serial and serial-alternative schedulers only.

Several mechanisms are provided in the scripting language to program the subtransactions and control flow of information between them. Below we discuss how these things are done in detail. Concepts are clarified by examples drawn from the case study. Readers may refer to section 3.2.4 for a detailed presentation of the case study which is a home-loan request transaction running in a distributed banking environment.

4.4.1 Conditional Subtransactions

At this point we revisit part of the case study to be able to give better examples for different tasks. The subtransaction finance (see section 3.2.4) does the equivalent of the above-mentioned transaction transfer plus trying to borrow money for the applicant if he/she cannot provide enough deposit for the home loan. It therefore has three subtransactions: withdraw, deposit, and personal-loan. Despite the conventional money transfer transaction, there is no need to assume that the deposit should be done only after the success of withdraw since, being related to the same top-level transaction, either both commit or abort. Hence finance is a parallel scheduler and runs faster than transfer. The third subtransaction of finance is the conditional subtransaction personal-loan. It is conditional because
the applicant may not need a personal loan, i.e. existence of the subtransaction depends on run-time values for different applicants.

As a more descriptive example, the user-defined class FINANCE appears here and its description follows:

class FINANCE is
  -- arguments are: name, loan_deposit, applicant_acnt,
  -- applicant_balance, loan_account.
  PARALLEL;
  const me: STR := "finance";
  work: STR is
    -- decide amount to transfer and whether to borrow or not.
    amount: STR;
    borrow: REAL := argv[1].to_r - argv[3].to_r;
    if borrow > 0 then amount := argv[3];
    -- transfer the whole balance and borrow the rest.
    else
      amount := argv[1]; -- only transfer loan_deposit.
    end;
    subtransaction.name("withdraw").arg(argv[2]).arg(amount);
    subtransaction.name("deposit").arg(argv[3]).arg(amount);
    -- conditional transaction.
    if borrow > 0 then
      subtransaction.name("personal_loan").arg(borrow.to_s);
    end;
    ...
end; --work
end;

The first two lines are comments, which start with "--" as in Sather (not necessarily a full line). Then the class PARALLEL is inherited and the constant "me" is
customized to the name of the subtransaction as usual. The work routine uses the
main arguments to decide how much money to transfer and whether a borrow
subtransaction is needed or not. The variables *amount* and *borrow* show the
amount to be transferred and borrowed respectively. Notice the order of the
main arguments from the comment lines. Also notice the use of *to_r()* and *to_s()*
functions which are Sather functions provided for conversion from a string to a
real numbers and vice-versa.

The last lines of the class define the three subtransactions. The first two are
as in *transfer*. The third subtransaction is a conditional one. If there is a need to
borrow, the subtransaction *personal_loan* is added. In this way the transaction
may have either two or three subtransactions depending on the run-time values.

### 4.4.2 Site Specification

Programmers may specify the sites to process subtransactions or may leave
the choice to the system, provided that the necessary information is available to
it. In the current prototype we have developed a class which keeps information
about sites and the services they provide. The system looks up the site if it is not
specified by the programmer, and continues the process if an appropriate site is
found for a service. A combination is also possible, i.e specifying the site for only
some of the schedulers in the transaction tree or even for a subset of the children
of a scheduler.

The sites are specified by customizing the attribute *hosts* in the “work” rou-
tine. The host name concatenated with the name of the corresponding subtrans-
action (separated by a “;”) appears as a string separated with a “;”. For example
in the class *FINANCE*, at the beginning of the routine “work”, one may specify
the hosts for *withdraw* and *deposit* as:

```
hosts := “arp:withdraw; earth:deposit”; 
```

It will cause *withdraw* and *deposit* to be submitted to hosts *arp* and *earth* respec-
tively. In this case the appropriate host for the *personal_loan* subtransactions,
which is a scheduler, is the current host. Hosts are defined for atomic and dele-
gated transactions only (see the following subsection), and the current host always
takes care of the rest.
4.4.3 Replicated Subtransactions

Schedulers of any type generally have multiple children. There could be one of the following two cases:

- each child is a different subtransaction at the same or a different site. An example is the finance transaction.
- a single child explicitly defined, but to be tried at a number of sites. An example is a subtransaction to reserve a ticket for the same trip, but to try it at different airlines (e.g. to get the less-expensive one).

Although the second case effectively results into multiple subtransactions, programmers just define one of them together with their choice of sites. If no sites are specified the system tries all possible ones recommended by the information source which is implemented as a class in the current prototype. The personal-loan subtransaction of finance, which is a schedule by itself, is an example from the case study (see figure 3.2). The transaction tries a personal-loan at some sites in parallel to get a result as soon as possible. It is a parallel-alternative scheduler of the atomic transaction offer-personal-loan and looks like:

```plaintext
class PERSONAL_LOAN is
  -- arguments are: name, to_acnt, amount.
    PARALLEL_ALTERNATIVE;
  const me: STR:= "personal_loan";
  work is
    hosts := ... -- customized if wanted.
    subtransaction.name("offer_personal_loan").arg...
  end; --work
end;
```

Notice the inheritance of the scheduler class PARALLEL_ALTERNATIVE and the customization of the constant "me". The "work" routine defines the hosts if necessary and the subtransaction which will be submitted to all hosts. Due to the fact that a single-child schedule is not allowed in any other circumstances, this case is semantically unique and therefore the system understands what is meant and takes care of it automatically. Programmers do not need to explicitly specify any thing else.
4.4.4 Serial Argument Dependency

Subtransactions may get two sorts of arguments (parameters):

- the first type of arguments are ones provided directly by the programmer. They may type them in when submitting the top-level transaction. An example already seen is \texttt{transfer from\_acnt to\_acnt amount}. They may also define the arguments in the "work" procedures as seen for the subtransactions of \texttt{transfer} and \texttt{finance}.

- the second type of arguments are ones provided not directly by programmers, but indirectly by other subtransactions of the same parent. This is applicable only to serial schedulers, the only type of scheduler having subtransactions running one after the other. For example a subtransaction may open an account and then pass the account number to another subtransaction to deposit some funds into it. We refer to such siblings as \texttt{arg\_producer} and \texttt{arg\_consumer}.

The script language for TractorS provides a simple mechanism to support the second case. The "work" routine of the \texttt{arg\_producer} should return the arguments packed into a string. The system will automatically add them to the end of the parameter list of \texttt{arg\_consumers}.

The producer may produce any number of arguments for any number of its siblings. No restrictive assumptions are made. We propose a simple convention to cover all cases:

Each set of arguments must be attached to the name of the consumer by a ":". Multiple sets are separated by a semicolon.

An example is the serial scheduler \texttt{finalize} in the case study (see section 3.2.4). Its subtransactions are \texttt{open\_account}, \texttt{balance}, \texttt{finance} and \texttt{update}. The last two need the account number of the newly opened account. Therefore the subtransaction \texttt{open\_account} should pass the account number to \texttt{finance} and \texttt{update}, so the class looks like:

```pascal
class OPEN\_ACCOUNT is
  -- arguments are name, acnt\_info.
  ACTION;
```
4.4.5 Commit Dependency

Two subtransactions have commit dependency (also called commit serial dependency in the literature) if they may proceed in parallel, but one cannot commit before receiving a message about commitment of the other. We call the two tasks signal_sender and signal_receiver. To support this, programmers should specify such tasks and make the dependent one wait for a commit signal from the other in the following way.

1. in the "work" procedure of the sender customize the attribute is_commit_dep to "true", e.g
   is_commit_dep := true;

2. in the "work" procedure of the receiver suspend the final commitment by calling the get_signal routine and providing the name of the sender as its parameter, e.g:
   get_signal("sender");

In the case study programmed at the end of this chapter, the subtransaction offer_personal_loan has commit dependency on deposit. This is because the applicant is not eligible to a personal loan unless he/she provides some of the home loan deposit, i.e., some funds are successfully transferred from his/her account. Parts of the two classes appear below.

class DEPOSIT is
   ACTION;
   const me: STR := "deposit";
work: STR is
  ...
  is_commit_dep := true;
  ...
end;
end; --deposit

class OFFER_PERSONAL_LOAN is

  const me: STR := "offer_personal_loan";

  work: STR is
    ...
    get_signal("deposit");
    -- get signal from deposit to commit.
  end; --work
end; (a) customizes the attribute is_producer to "true"

The two concurrent processes may run at different sites without knowing each other (the signal_receiver may even be created later). Communicating between such processes with no run-time knowledge of each other is in fact difficult. The situation is however common in TractorS since subtransactions are dynamically configured via schedulers at run-time. This type of communication is supported in TractorS by defining dynamic groups. Processes communicate by joining the same group, broadcasting messages, waiting for others to join (called the barrier mechanism), etc. The only thing that they all need to know is the name of the group. We have decided to use the name of the signal_sender. That is why the receiver should provide this name which is actually used to join the group.

4.4.6 Value Dependency

Concurrent subtransactions belonging to the same top-level transaction may have value dependency, i.e. one may produce a value to be consumed by the other. The latter blocks when the value is needed until it arrives. Here we explain how value dependency is supported in the scripting language for TractorS. Assume subtransactions value_producer and value_consumer are to produce and
consume a value respectively. Any number of values may be produced, so they are packed in a string for which Sather provides functions to convert to/from other basic types. One value_producer may produce values for any number of value_consumers and vice-versa, but we reasonably assume that the values are the same for all value_consumers, i.e we do not allow a value_producer to produce different values for each value_consumer. This is what the programmers should do:

1. if there is more than one value_consumer for a value_producer (the default is one with no special action necessary) then in the “work” procedure of the value_producer and all value_consumers customize the constant no_of_consumers to the actual number of consumers, i.e if there are three value_consumers then:

   no_of_consumers := 3;

2. in the “work” procedure of the value_producer:

   (a) customize the attribute is_producer to “true”, i.e,

       is_producer := true;

   (b) store the values (packed into an string) in the attribute produced_values, i.e,

       produced_values := "value1 value2 ...";

3. in the “work” procedure of value_consumers’, take the values by calling the get_values routine and providing the name of the value_producer as its parameter, i.e:

   values := get_values("value_producer");

   The list should then be popped and the values utilized.

As an example, the subtransaction support_home_loan in the case study has value dependency on offer_home_loan. The latter should provide the name of the bank for the former, which blocks at a point to receive it. The two classes appear below and a brief description follows.

class OFFER_HOME_LOAN is

   ACTION;
const me: STR := "offer_home_loan";
work: STR is
  is_producer := true;
  ... -- do the actual work.
  produced_values := bank_name; --whatever it is.
end;
end; --offer_home_loan

class SUPPORT_HOME_LOAN is
  ACTION;
  const me: STR := "support_home_loan";
  work: STR is
    ...
    values := get_values("offer_home_loan");
    ...
    -- consume the value.
  end;
end; --support_home_loan

Only one value is produced, so the class OFFER_HOME_LOAN sets the
attribute is_producer to true and also puts the value bank_name into the attribute
produced_values. TractorS run-time system defines the group and broadcasts the
value. The value_consumer (SUPPORT_HOME_LOAN) calls the get_values rou-
tine with the name of the producer subtransaction (get_values("offer_home_loan"))
which is the name of the dynamic group as well. The system joins the task to
the group and receives the value for it. Recall that as a convention we always use
capital letters for names of classes (enforced by Sather syntax) and lower case
letters for their instances (subtransactions).

The get_values() function can be used more than once with different arguments
to receive values from different value_producers.

4.4.7 Concurrent Subtransactions

Parallel and parallel-alternative schedulers may activate concurrent subtrans-
actions. Notice that multiple atomic or delegated transactions belonging to the
same top-level transaction may not run at the same site concurrently due to a condition imposed by the concurrency control mechanisms most suitable for TractorS (see section 3.2.8). The run-time system prevents such an error in the following way:

1. a list of all sites having in-progress atomic or delegated transactions is kept at run-time for each top-level transaction. The list is updated whenever a subtransaction is started or ended.

2. if the site to execute an atomic or delegated transaction is already in the list, it is tried at other sites in the same cluster or is suspended until one is able to do the job correctly.

4.4.8 Non-vital Subtransactions

A scheduler may have non-vital subtransactions, i.e. ones which preferably commit but do not cause abortion of the parent if they do not commit. This case makes sense only for parallel schedulers because:

- in a serial scheduler a subtransaction has begin-on-commit dependency on the previous one, i.e. it cannot begin unless the previous one commits.
- in choice schedulers all subtransactions are semantically non-vital so there is no need for such an explicit declaration.

Non-vital subtransactions can be simulated by a serial-alternative scheduler with the null subtransaction being its last child. The subtransaction null commits after getting started without accessing any data items.

There is no reason to prevent a non-vital subtransaction from having value or commit dependency. However, such a dependency causes a small problem: the producer which waits for the consumer to get the value (signal) may get blocked due to the failure of the latter. To rectify the situation, the null transaction is of two types:

1. null_task which commits immediately after getting started.

2. null_value which should be used if value or commit dependency is involved.

All it does is receiving the value (signal) without actually using it. It commits immediately then.
In short, to make a subtransaction non-vital, programmers should replace it by the following serial-alternative scheduler:

1. define the actual subtransaction as the first child.

2. define either null_task or null_value as the second and the last child.

It is obvious that the first subtransaction is preferred by the parent, but it gets a success message back in any case.

As an example, consider the subtransaction finalize in the case study with four subtransactions, the last one being update. The programmer may decide to commit the transaction even if the system cannot update all its records right away (let it be done at a comfortable time for the system). He/she should replace the subtransaction update with a non-vital one so that the long-duration home-loan request transaction is not aborted. Keeping in mind that update does not have value dependency, here is how its replacement would look like:

```plaintext
class N_UPDATE is
  SERIAL_ALTERNATIVE;
  const me: STR:= "n_update";
  work: STR is
    subtransaction.name("update").arg(...);
    subtransaction.name("null_task");
  end;  --work
end;
```

### 4.4.9 Compensation

Whether a subtransaction is compensatable or not is decided by its semantics stored in the subtransaction itself. As a rule, only atomic transaction can be declared compensatable. This is to ensure the integrity of the database systems when failures occur. To declare a subtransaction compensatable, programmers should inherit the class COMPENSATABLE_ACTION. For example, to make the subtransaction withdraw compensatable, one should define the class as:

```plaintext
class WITHDRAW is
```
-- arguments are: name, from_acnt, amount.
COMPENSATABLE_ACTION;
const me: STR := "withdraw";
work: STR is
    --THE APPLICATION DEPENDENT TASK IS PROGRAMMED HERE.
end; --work
end; --withdraw

The system automatically commits the subtransactions and cancels its effect later on if the top-level transaction fails.

The class COMPENSATABLE_ACTION provides all functionality of the class ACTION plus some extra facilities for distinguishing and committing compensatable subtransactions. There is a particular constant, called is_compensatable which is set to true. The system checks this constant. If it is true the subtransaction is committed and the cancel-log is updated for a possible cancel procedure. As far as other transactions are concerned this action is the same as safe_commit, i.e. the data items are released with no restrictions.

TractorS has a mechanism to stop programmers from changing the compensatability property of existing subtransactions, by redefining them either intentionally or accidently. This is possible in Sather because classes can define their own variables and methods, and inherit from others freely. A programmer may attempt to do so by redefining the is_compensatable constant in the corresponding object classes or inheriting the COMPENSATABLE_ACTION class in a wrong way. In Sather, the last defined or inherited copy is always used and there is no way to stop redefinition of parts of an object.

To illustrate, we assume that the subtransaction deposit exists and is not compensatable. A programmer may try to redefine the class DEPOSIT in the wrong way to make it compensatable as follows (notice the comments):

class DEPOSIT is
    -- arguments are: name, to_acnt, amount.
    COMPENSATABLE_ACTION;                        -- either this:
    const is_compensatable: BOOL := true;        -- or this.
    const me: STR := "deposit";
    work: STR is
--THE APPLICATION DEPENDENT TASK IS PROGRAMMED HERE.
end; --work
end; --deposit

To run it, one may type deposit acnt_123 1000.00
TractorS produces the following result:

1 deposit#341: BEGIN ATOMIC TRANSACTION.
2 deposit#341: wrong type; deposit is not compensatable. ABORT.
3 deposit#341: undo successful.
4 deposit#341: END ATOMIC TRANSACTION.

4.4.10 Setting Timeout Periods

A timeout period is attached to each atomic subtransaction. TractorS runtime system raises an exception if the timeout period is exceeded and enters a particular timeout_handler routine which aborts the subtransaction and informs its responsible transaction. Programmers have three choices: (i) accept the default which is the same for all atomic subtransactions. (ii) change the default by customizing the attribute timeout_period in the class INITIALIZE. This class is designed for users to customize their environment without accessing other parts. (iii) set their own timeout period (in micro seconds) for any number of the atomic transactions by updating the attribute timeout_period in their "work" routines. As an example of the third choice we update the time_out period for the sub-transaction WITHDRAW to 500.

class WITHDRAW is
...
work: STR is
  timeout_period := 500;
...
end; --work
end; --withdraw

At this point we summarize the way TractorS handles deadlocks.
• local transaction managers at component databases and knowledge bases handle deadlocks autonomously. They may have any deadlock-free or deadlock detection and correction scheme;

• the file systems for which the Unix distributed locking facilities is proposed have a deadlock-free scheme (see section 5.3.4).

• possible global deadlocks are broken by timeout.

### 4.5 Programming the Case Study

In section 3.2.4 we introduced the case study. Here we revisit it briefly and program it using the TractorS library.

The top-level `home_loan` transaction tries to find a bank to offer the loan as well as an insurance company to support it, and then finalizes loan activities if this step is successful. Finding a bank and an insurance company could go in parallel up to some extent, but finalizing the loan comes after. Hence the top-level transaction is a serial scheduler with two subtransactions:

(i) The first subtransaction (`find_institutes`) is a parallel scheduler which tries to find a bank and an insurance company in parallel. It does so by means of two subtransactions `find_bank` and `find_insurance_co` which are of type `serial_alternative` and `parallel_alternative` respectively. Their atomic transactions `offer_home_loan` and `support_home_loan` have value_dependency (name of the bank).

(ii) The second subtransaction (`finalize`) is activated if the previous one succeeds. It opens an account for the loan at the bank offering it, retrieves the balance of applicant’s account to get the loan deposit from, starts the `finance` subtransaction to provide the loan deposit, and finally updates records and issues documents. The subtransaction `finance` is a parallel scheduler. Its last subtransaction is conditional and has a commit-dependency on its second, i.e `personal_loan` is aborted if transferring the money has not been successful.

The transaction is programmed in the following two subsections. Parts of the case study are already used as examples in this chapter, so the explanations below would be brief.
4.5.1 Schedulers

The following list shows the classes for the schedulers. The top-level transaction (class HOME_LOAN) has four arguments: its name, the loan_amnt needed by both subtransactions, the loan_deposit and the applicant_acnt needed by the second subtransaction (finalize).

The subtransaction find_institutes and its two children find_insurance_co and find_bank are straightforward. Each one of the children has the same atomic subtransaction to be repeated at the set of sites defined by the programmer or decided by the system (atomic transactions are described in the following subsection).

The subtransaction finalize has four children. It is a serial scheduler and its subtransactions have argument dependency. For example the first one should provide the account number to the third and fourth.

The next subtransaction is finance. It calculates the amount to be transferred and then defines the two subtransactions withdraw and deposit to do so. At the end it defines a conditional subtransaction to try a personal loan if necessary.

The task is already described in section 4.4.1.

class HOME_LOAN is
  -- arguments are: name, loan_amnt, loan_deposit,
  -- and applicant_acnt.
  SERIAL;
  const me: STR:= "home_loan";
  work: STR is
    subtransaction.name("find_institutes").arg(argv[1]);
    subtransaction.name("finalize").arg(argv[2]).arg(argv[3]);
  end; --work
end;

class FIND_INSTITUTES is
  -- arguments are: name, loan_amnt.
  PARALLEL;
  const me: STR:= "find_institutes";
  work: STR is
    subtransaction.name("find_bank").arg(argv[1]);
class FIND_BANK is
    -- arguments are: name, loan_amnt.
    SERIAL_ALTERNATIVE;
    const me: STR := "find_bank";
    work: STR is
        hosts := "defined if desired";
        subtransaction.name("offer_home_loan").arg(argv[1]);
    end; --work
end;

class FIND_INSURANCE_CO is
    -- arguments are: name, loan_amnt.
    PARALLEL_ALTERNATIVE;
    const me: STR := "find_insurance_co";
    work: STR is
        hosts := "defined if desired";
        subtransaction.name("support_home_loan").arg(argv[1]);
    end; --work
end;

class FINALIZE is
    -- arguments are: name, loan_amnt, loan_deposit, and applicant_acnt.
    SERIAL;
    const me: STR := "finalize";
    work: STR is
        subtransaction.name("open_account");
        subtransaction.name("balance").arg(argv[2]);
        subtransaction.name("finance").arg(argv[2]).argv[3]);
        subtransaction.name("update").arg(argv[1]).argv[2]);
    end; --work
end;
Chapter 4. Linguistic Facilities

class FINANCE is
   -- see section 5.4.1 for the complete list.
end;

class PERSONAL_LOAN is
   -- arguments are: name, loan_amnt.
   -- has commit_dependency on deposit.
   SERIAL_ALTERNATIVE;
   const me: STR:= "personal_loan";
   work: STR is
      hosts := "defined if desired";
      subtransaction.name("offer_personal_loan").arg(argv[1]);
   end; --work
end;

4.5.2 Atomic Transactions

The following list shows the classes for some of the atomic transactions in the case study. Since they look similar, we only provide ones with special characteristics.

The class OFFER_HOME_LOAN produces the name of the bank. In its “work” procedure the switch is_producer is set to true and its attribute produced_values is also set to be the name of the bank. The actual application dependent task is also performed here.

The class SUPPORT_HOME_LOAN is the consumer of name of the bank, so in its “work” procedure the produced value is received by calling get_values with the name of the procedure, i.e offer_home_loan.

The class OPEN_ACCOUNT which is a subtransaction of the serial schedule finalize provides an argument for its siblings finance and update. Hence its “work” procedure returns the account number together with the name of the consumer.

The class DEPOSIT a signal-producer. The signal-consumer task is the subtransaction OFFER_PERSONAL_LOAN which has commit dependency on deposit. In its “work” procedure the switch is_commit_dep is set to true. The code for the class OFFER_PERSONAL_LOAN also appears below. It is a compensatable action and therefore inherits the class COMPENSATABLE_ACTION.
class OFFER_HOME_LOAN is
    -- arguments are name, loan_amount.
ACTION;
const me: STR := "offer_home_loan";
work: STR is
    is_producer:= true;
    ... -- do the actual work.
    produced_values := bank_name; --whatever it is.
end;
end; --offer_home_loan

class SUPPORT_HOME_LOAN is
    -- arguments are name, loan_amount.
ACTION;
const me: STR := "support_home_loan";
work: STR is
    ...
    values:LIST{STR} := LIST{STR}::create;
    values := get_values("offer_home_loan");
    ... -- consume the value.
end;
end; --support_home_loan

class OPEN_ACCOUNT is
    -- arguments are name, acnt_info.
ACTION;
const me: STR := "open_account";
work: STR is
    ...
    res:= "account_no:finance; account_no:update";
end;
end; --open_account

class DEPOSIT is
    -- arguments are name, from_acnt, amount.
ACTION;
const me: STR:= "deposit";
work: STR is
  is_commit_dep:= true;
  ...
end;
end; --deposit

class OFFER_PERSONAL_LOAN is
  COMPENSATABLE_ACTION;
const me: STR:= "offer_personal_loan";
work: STR is
  ...
  get_signal("deposit");
  -- get signal from deposit to commit.
end; --work
end; --

The set of classes described above define the transaction tree for the home-loan request transaction. Users should compile all classes and put the object code in TractorS bin directory and execute the top-level task which calls all others.

Atomic transaction do not necessarily have to be written in Sather. They may be written in any language with C binding to PVM for distributed communication and global synchronization. Facilities like commit and value-dependency and unsafe-commit should be either programmed in those languages or the "work" routines imported to Sather and called.
Chapter 5

Implementation

In this chapter different issues regarding the implementation of the scripting language for TractorS are discussed. The previous chapter described how to program using the language. Here we show how things are done. Readers interested in the ideas and reasons behind major decisions in design and implementation of the language should read this chapter.

5.1 Implementation Rationale

Two major parts of the scripting language for TractorS are the host language, the languages that we have chosen to add TractorS facilities to, and the way to do distributed message-passing. Our choice is Sather for the first and PVM for the second one. In this section we talk about available choices for each one and reason about the decisions made.

5.1.1 The Host Language

To implement the scripting language, we believe it makes more sense to pick up an appropriate programming language and add the new capabilities to it rather than to build such a language from scratch. TractorS inherits properties from two language paradigms, namely actor and object oriented languages. An appropriate candidate from each paradigm is discussed below.
ABCL/1: A Pragmatic Actor Language

Our choice from the set of actor languages was ABCL/1 [53] (Actor Based Concurrent Language) which is more pragmatic than the conventional actor model and implements a novel and interesting collection of message passing constructs. ABCL/1 supports *future remote procedure calls* in which the sender process may proceed until the result from the receiver is needed and gets blocked if it is not yet ready. Objects may have *future* parts which declare instance variables of this type. Such a declaration creates other objects that instruct the compiler to expand such references into particular message passing instructions. An object in ABCL/1 can be bound to send reply messages to a destination object. Guards could be used to further limit the messages to match certain patterns. An object may have private functions which can only be used in definition of other methods of the object. Objects are serial as in actor systems, i.e. one method at a time may execute. A class of high-priority messages can however interrupt the execution of ordinary methods.

Message passing in ABCL/1 is asynchronous and point-to-point. There are three modes of message passing:

1. *past* in which the sender issues the message and resumes its activity. The receiver may delegate the message to a third object to directly respond to the original sender.

2. *now* in which the sender sends the message and waits for the response. A delegation process is possible here as well.

3. *future* in which the sender issues the message and specifies the name of a variable in which the response should be directed. It resumes its activity, and refers to the variable whenever the value is needed which will naturally block until it is received.

Auxiliary objects are introduced by which *now* and *future* types can be reduced to *past* type.

An object in ABCL/1 has two message queues for ordinary mode and express mode messages. An arriving message is acted upon depending on the type of message and the object’s state which can be *dormant*, *active*, or *waiting*. An
object is in a dormant state if it is not active and there is no message in either message queue. Active state means currently executing a message. An object is waiting if it has sent a message and expects a response to get active again. Arriving express mode messages interrupt even the waiting object. ABCL/1 and some other actor languages are also supported in multicomputers and supercomputers, including the one at our institution [74] [75].

ABCL/1 looks attractive for TractorS implementation. Both have their main ideas borrowed from the actor model of computation. The three states of objects are also found in TractorS. For example a serial scheduler is dormant before being activated, active while trying to activate its subtransactions, and goes into wait state after activating each one of them. The only message which can re-activate it would be a ready-to-commit or abort message from that child or an express mode message from the parent. TractorS objects however only need one message queue as in the original actor model. TractorS has only two types of express mode messages, abort and timeout, but the target objects should not resume their previous mode after processing such messages. There are different ways of implementing such messages. We have chosen to use Unix-like signals to timeout or abort processes and let the responsible transaction remove their incomplete effects.

ABCL/1 was not chosen for TractorS implementation due to the reasons summarized below:

- it does not support distributed computing and we do not know of an easy way to add this capability to it.
- compared to our other choice which is explained later, it misses a variety of important constructs.
- it has no built-in system for inheritance. The behavior of an object is entirely specified by its script. It would be very hard to build a transaction management system with this restriction.
- message passing is point-to-point while TractorS messages are frequently forwarded, multicasted, and broadcasted to a number of destinations.
• the express mode messages behave differently as already explained. TractorS objects do not require the complex double queueing mechanisms.

• to the best of our knowledge, it is not public domain. All software packages involved in development of the scripting language for TractorS are public domain. As a result any interested user can install and use the whole software package for free.

**Sather: A Public Domain Object Oriented Language**

The family of object oriented languages has grown over time resulting in a vast collection of members, a few of them being public domain. A fairly new one, initially derived from the more well known Eiffel language [76], [77] is Sather [12], [13] which is being developed at the International Computer Science Institute of University of California at Berkeley. It focuses on practical needs of writing efficient, reusable code. It attempts to support a powerful object oriented paradigm without sacrificing performance, safety, and correctness checking. It is simple, efficient, interactive, safe, and non-proprietary. It aims to be as efficient as C, C++, or Fortran, as elegant and safe as Eiffel or CLU, and to support interactive programming and higher-order functions as well as Common Lisp, Scheme, or Smalltalk. Sather programs can be compiled into portable C code and can efficiently link with C object files. This facilitates the linkage of the language to the existing codes. Sather has a very unrestrictive license which allows its use in proprietary projects and encourages contribution to the public library. Its latest version, Sather 0.5 in which our classes are written appears with even more well thought constructs and cleaner syntax. The parallel version of the language [14] provides multiple threads (i.e, lightweight processes) and synchronization mechanisms as the main requirements of concurrent programming.

The language has been chosen to implement the scripting language for TractorS due to the following reasons:

• it provides a rich collection of facilities which is proved helpful in transaction processing, as we have discussed in [9].

• The ability to compile Sather programs into portable C code facilitates the linkage of the language to the real world applications. It is also possible
to use existing packages for various means. We have added distributed computing power to the language via this property.

- it shares the primary ideas of performance, and reusability with TractorS.
- the parallel version of the language provides concurrent programming necessities of TractorS.
- it is public domain.
- having access to its compiler gives us the opportunity of adding new constructs to the language if necessary.

5.1.2 Interprocess Communication

There are four well-known methods of organizing distributed computing communications: remote procedure call (RPC), virtual shared memory, object request broker and message passing.

RPC is a low level mechanism supported directly in operating systems (e.g Unix), which is actually used in developing other distributed facilities as well as in lots of more high level tools in distributed computing. Our purpose can be better served by more high level tools.

Virtual shared memory is the method used in Linda [78] and POSYBL [79]. Linda, a concurrent programming model evolved from a Yale University research project, is the most efficient member of this paradigm [80]. Tuple-space is the primary concept in Linda. It is an abstraction via which cooperating processes communicate. It is proposed as an alternative paradigm to the traditional methods of parallel processing, i.e the ones based on shared memory and message passing. It is actually an abstraction of the distributed shared memory with associative property and some other minor distinctions. Linda is practically a programming language extension for facilitating parallel programming.

The object request brokers approach, still in its infancy, looks promising in near future. Its well known member, CORBA (Common Object Request Broker Architecture) [81] is a set of standards published by Object Management Group in 1991. In this architecture a software component is viewed as an object with its own state which responds to appropriate messages. CORBA does not specify a
programming language, but an interface between objects and the object request broker. Any software system written in any language may have an interface layer between it and the object request broker.

Message passing is a popular and widely used method which has been the basis for several high level packages facilitating the task. Currently CIS belongs to this paradigm, so current TractorS implementation is a concurrent message-passing system. In the rest of this section we examine the well known packages in this paradigm and justify our choice.

Well known message-passing packages are Parasoft Express, P4 (Portable Programs for Parallel Processors) by Argonne National Laboratory, TCGMSG (Theoretical Chemistry Group Message passing system) by Battelle Pacific Northwest Laboratory, ISIS by The ISIS Distributed Systems Inc, and PVM (Parallel Virtual Machine) by Oak Ridge National Laboratory [80] [82], [83]. PVM is considered as the de facto standard of the set with nice properties and good performance. In certain applications it is shown to be much faster than comparable products [80]. PVM looks to be the best choice, but ISIS, despite its comparatively low performance, is also very powerful and provides extra facilities. In fact, at the system analysis phase of the project, we have put a lot of effort and made experiments to choose between ISIS and PVM as the two best possibilities (the last public domain version of ISIS is used for this purpose). PVM looked more suitable and later experiments proved it. We discuss how TractorS can be implemented in each one below and argue about our choice.

**ISIS: a Powerful Package**

ISIS [85] was originally developed by Cornell University and has subsequently been commercialized by ISIS Distributed Systems, Inc. It is a fault tolerant system for building applications of cooperating, distributed processes. ISIS is not chosen for TractorS implementation due to its low performance, complexity, and particularly due to the fact that it is not supported in public domain any more. However, based on our experiments in this regard, we discuss the general considerations that should be taken into account and provide a basis for comparison.

1. as in most similar packages, ISIS guarantees consistent order of arrived messages at different processes and provides synchronization mechanisms.
Chapter 5. Implementation

Its fault-tolerant facilities which automatically replaces failed hosts with the most appropriate available ones is attractive.

2. depending on the application, some parts of ISIS can be utilized in the scripting language for TractorS. Examples are log managers and failure handling. ISIS provides synchronization and locking mechanisms independent of its transaction management facilities.

3. ISIS transaction manager and log manager support atomic transactions which are building blocks of TractorS. Unfortunately it supports neither nested transactions nor concurrency control.

4. ISIS supports process groups (including hierarchical process groups). TractorS is best implemented as a group of processes. Examples of such group are schedulers, compensation, and delegation activities. Also sites in ISIS are organized into clusters in a way similar to cluster definition in TractorS.

5. ISIS also provides a spooler and long-haul facility which is helpful for asynchronous message passing in wide-area networks. The ISIS spooler is also valuable for updating information sources such as close and foreign acquaintances knowledge sources in TractorS.

ISIS transaction manager provides various constructs like beginning, committing, and aborting atomic transactions and a choice between one-phase and two-phase commit protocols. There are two possibilities to consider for using it in the scripting language for TractorS:

- let ISIS transaction manager handle atomic transactions and build the rest of TractorS transaction manager on top of it. This approach makes sense but a lot remains to be done by programmers. One should explicitly lock/unlock data items and control concurrency of transactions. Although components of CIS mostly provide their own mechanisms, it looks attractive for parts of CIS which do not provide such facilities (e.g. file systems). However, operating systems provide such facilities directly. An example is the distributed locking facilities of Unix.

- let the whole top-level transaction be a single ISIS transaction. This does not meet TractorS requirements because top-level transactions are not atomic
here. The major problem is compensation; if the top-level transaction aborts the databases will be restored to their state prior to the one at the beginning of the transaction. This is against the philosophy of compensation because the effect of other concurrent transactions are removed.

As a result ISIS transaction management facilities are not suitable for TractorS due to their restricted ACID property. The rest is common between most packages. PVM is public domain and is more compatible with TractorS as discussed below.

**PVM: the de Facto Standard**

PVM (Parallel Virtual Machine) provides process creation and communication in a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. Its development started in 1989 and is now an ongoing research project involving a number of institutions and universities [15]. Users define their collection of computers as one large distributed memory computer (virtual machine). Tasks defined as a unit of computation analogous to Unix processes, are created on the virtual machine and communicate and synchronize by supplied functions. Applications which can be written in C, Fortran 77, and now Sather (via the interface provided in this work and put in public domain in near future), can be parallelized by using common message passing constructs. PVM supports heterogeneity at the application, machine, and network level and allows the virtual machine to be interconnected by a variety of different networks. There are routines to add/delete hosts, start up and terminate tasks, send/receive signals and messages, and to find out information about the configuration, messages, etc. Processes that enroll in PVM are given a unique integer task identification, and should not halt without exiting PVM (errors may occur otherwise).

The send and receive linguistic constructs covered in major design approaches include asynchronous message passing, synchronous message passing, future remote procedure call, blocking and nonblocking remote procedure call [86]. The model used in distributed transaction processing is primarily the first one. The actor model of computation belongs to this paradigm. In this model a process
blocks until a message is available and may hand the message over to a particular unit without the sender getting blocked.

PVM provides a rich collection of message passing routines in different modes including non-blocking send, blocking and non-blocking receive, non-blocking probe, barrier synchronization, and Unix-like signals. In addition to point-to-point communication, it supports multicast and broadcast to sets of tasks. Routines can be called to return information about received messages before actually receiving them. The message order preservation is guaranteed but messages of different type can be received in non sequential order. Multiple buffers can be defined with one buffer being defined as *active send* and one as *active receive*. A message can contain several arrays, each with a different data type, with no limit to the complexity. Dynamic process groups are also supported. Processes can belong to multiple groups, and groups may change dynamically during the computation. Routines are provided for processes to join/leave a group, broadcast messages, query for information about other group members, and synchronize with them. These facilities can be used to simulate all sorts of services and provide message passing necessities of TractorS.

Here we summarize the reasons for choosing PVM as a vehicle for interprocess communication in the script language for TractorS.

- PVM is the de facto standard of the message passing paradigm and is available on many platforms including supercomputers [80], [82].

- it is compatible with TractorS in nature since it has many commonalities with the actor model of computation. Basic actor functions are directly provided. Furthermore, it provides various facilities (process control and synchronization, dynamic groups, all types of message passing schemes, facilities to query information, etc) needed for TractorS communication and synchronization.

- it has a very good performance [80] which is a major goal in TractorS.

- in contrast to some similar products (including ISIS) it is still available in public domain.

- it is comparatively small and easy to use.
• it provides a good collection of related products such as graphical interface, distributed object PVM, fault tolerant PVM, message passing object oriented PVM, and run time monitoring of PVM programs [84].

5.2 Communication and Synchronization in TractorS

5.2.1 The Sather_PVM Interface

Distributed programming facilities in the scripting language for TractorS are provided via the Sather_PVM interface introduced in this section. The interface is a set of classes in Sather which provides the same functions as the C binding to PVM, with the same name (without the pvm_ prefix) and arguments. Figure 5.1 gives a pictorial view. As seen in the figure, PVM provides the following:

• facilities to define the virtual machine, add and delete hosts, and query about the configuration. They are provided in two ways: command line, and routines to be called in programs.

• routines for process control, i.e creating and killing processes, enrolling and exiting PVM, finding the parent process, etc.

• routines for message-passing, i.e defining buffers, freeing them, packing data into them, sending and receiving messages in different modes, sending signals, etc.

• dynamic group handling, including routines for joining and leaving a group, checking its size and members, and broadcasting messages to group members.

• synchronization which is done basically via dynamic groups. Tasks join particular groups and call the barrier routine with appropriate parameters to synchronize with other groups. Also routines are provided to query for certain messages, including synchronization ones, before attempting to receive the message.
We discuss all these aspects of PVM later in this section and provide a sample of PVM routines.

Sather functions may call C functions and vice versa [12]. Since only Sather code appears at the top level of abstraction in the script language for TractorS, we only need C functions to be called and not the other way around. Access to C functions within Sather is provided by special classes "C" which may contain only shared attribute specifications and routine specifications. The name of each shared attribute and routine corresponds to a C external variable or function. Sather properly links the calls using the type specification given. Similar to calling routines of any foreign class, the prefix "C" is used to call functions defined in class C (e.g. C::func). User routines with a variable number of arguments are not supported. The return value is of the type specified in the "C" class. The following conversions are made in passing arguments:

Figure 5.1: A pictorial view of the Sather-PVM interface
1. Sather BOOLs (for the Boolean type) are passed as chars with value zero for false and non-zero for true.

2. Sather INTs and DOUBLEs are passed as ints and doubles.

3. Sather REALs are passed as floats.

Classes are provided in the language to help convert various types of constructs (e.g., structures, arrays, strings, etc.) back and forth. Using such facilities, parameters to C functions are defined and accessed.

PVM is written in C. We have developed the interface and made all necessary conversions via Sather-to-C facilities. These routines which are written in Sather are located in the class SATHER_PVM and appear with the same names (without the `pvm_` prefix) and arguments as the original PVM routines. To be self-contained, a sample of PVM interface routines is summarized in the table 5.1. Referring to [15] which provides a complete PVM routine list with explanation will help understanding the interface routines.

As an example, consider the routine `pvm_recv(int tid, int msgtag)` in the C binding to PVM which blocks to receive a message in the active buffer from process `tid` with `msgtag` labeling the contents of the message. This routine looks like follows in the SATHER_PVM class implementation:

```sather
recv(tid : INT; tag: INT) : INT is
  res := C::pvm_recv( tid, tag);
  print_res(res, STR::create.s("receive message"));
end;
```

The first line is the signature of the routine. The second line calls the C routine `pvm_recv()` (blocking receive) from PVM which is introduced to sather in the defined class “C” (see below), and stores the result in `res` (Sather keyword for returned result of routines). Finally it calls the appropriate trace routine to report the result and the result code (normally negative codes mean an error in PVM).

The corresponding entry in the class “C” which links the interface routine `recv` to the actual PVM routine `pvm_recv` is just two lines of code defining the routine and its arguments which looks like follows:
### Table 5.1: Most important PVM interface routines

<table>
<thead>
<tr>
<th>Category</th>
<th>Routine Name</th>
<th>Brief Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Machine</td>
<td>addhosts()</td>
<td>add one or more hosts to the virtual machine.</td>
</tr>
<tr>
<td></td>
<td>delhosts()</td>
<td>delete one or more hosts from the virtual machine.</td>
</tr>
<tr>
<td></td>
<td>config()</td>
<td>query about present configuration.</td>
</tr>
<tr>
<td>Process Control</td>
<td>mytid()</td>
<td>enroll in PVM and get a unique task id.</td>
</tr>
<tr>
<td></td>
<td>spawn()</td>
<td>spawn a task at a host.</td>
</tr>
<tr>
<td></td>
<td>exit()</td>
<td>exit from PVM.</td>
</tr>
<tr>
<td></td>
<td>kill()</td>
<td>terminate a specified PVM process.</td>
</tr>
<tr>
<td></td>
<td>parent()</td>
<td>find the process that spawned the present task.</td>
</tr>
<tr>
<td>Message Passing</td>
<td>mkbuf()</td>
<td>create a new message buffer.</td>
</tr>
<tr>
<td></td>
<td>freebuf()</td>
<td>dispose a message buffer.</td>
</tr>
<tr>
<td></td>
<td>initsend()</td>
<td>initiate the active buffer and specify encoding.</td>
</tr>
<tr>
<td></td>
<td>pkint()</td>
<td>pack the active buffer with integer type.</td>
</tr>
<tr>
<td></td>
<td>pkstr()</td>
<td>pack the active buffer with string type.</td>
</tr>
<tr>
<td></td>
<td>upkint()</td>
<td>unpack the active buffer with integer type.</td>
</tr>
<tr>
<td></td>
<td>upkstr()</td>
<td>unpack the active buffer with string type.</td>
</tr>
<tr>
<td></td>
<td>send()</td>
<td>non-blocking send.</td>
</tr>
<tr>
<td></td>
<td>sendsig()</td>
<td>send a Unix signal to a PVM process.</td>
</tr>
<tr>
<td></td>
<td>mcast()</td>
<td>non-blocking multicast.</td>
</tr>
<tr>
<td></td>
<td>recv()</td>
<td>blocking receive.</td>
</tr>
<tr>
<td></td>
<td>nrecv()</td>
<td>non-blocking receive.</td>
</tr>
<tr>
<td>Dynamic Groups</td>
<td>jingroup()</td>
<td>join a group (define if non-existent).</td>
</tr>
<tr>
<td></td>
<td>lgroup()</td>
<td>leave a group.</td>
</tr>
<tr>
<td></td>
<td>gsize()</td>
<td>query about group size.</td>
</tr>
<tr>
<td></td>
<td>bcast()</td>
<td>broadcast to all group members.</td>
</tr>
<tr>
<td>Synchronization</td>
<td>barrier()</td>
<td>increment counter and wait for others to call.</td>
</tr>
<tr>
<td></td>
<td>bufinfo()</td>
<td>find information about specified buffer.</td>
</tr>
<tr>
<td></td>
<td>probe()</td>
<td>check if a msg from a task with a tag has arrived.</td>
</tr>
<tr>
<td></td>
<td>pstat()</td>
<td>find status of a specified PVM process.</td>
</tr>
</tbody>
</table>
As a more comprehensive and detailed example, we present the interface routine `spawn()` from the class SATHER_PVM. Its arguments, the same number and order as in the C binding to PVM, are the name of the task to be spawned, its arguments, a flag to show whether there are user-preferred hosts, a host name (possibly void), number of processes to be spawned, and an integer array to accommodate the identifications of the spawned tasks. After setting the flag it checks and sets the string argument `c_argv` as expected by Sather. Conversion routines like `CSTRAY::strtok()` and `C::to_cstr()` can be looked up in Sather classes. The task is then spawned and the task ids returned.

```c
spawn(t_name: STR, argv: LIST{STR}, flag: INT, host: STR, nproc: INT, t_ids: ARRAY{INT}): INT is
  tmp: INT;
  tid_carray: CARRAY := C::create_carray(nproc);
  if host = void then flag := 0 else flag := 1 end;
  if argv /= void then
    c_argv: STR := STR::create;
    c_argv.c_argv.s(argv[0]);
    i: INT := 1;
    loop
      until! (argv[i] = void or i = argv.size);
      c_argv := c_argv.c(' ').s(argv[i]);
      i := i + 1;
    end;
    tmp := C::pvmSpawn(to_cstr(t_name), CSTRAY::strtok(c_argv, " "), flag, to_cstr(host), nproc, tid_carray);
  else
    tmp := C::pvmSpawn(to_cstr(t_name), void, flag, to_cstr(host), nproc, tid_carray);
  end;
```
Chapter 5. Implementation

```plaintext
code: INT := tmp;
if tmp = 0 then code := -14; end; -- pvm code for failure.

print_res(code, STR::create.s("spawn returned ").i(tmp).s(t_name)
    .s(" process(es) at host ").s(host).s(" It was"));
i: INT := 0;
loop
    until! (i = nproc);
    it: COB := C::carray_get(tid_carray, i);
    t_ids[i] := it.to_int;
    i := i + 1;
end;
res := tmp;
end; -- spawn
```

Besides SATHER_PVM, there are three other related classes:

- the class INITIALIZE defines the instance variables related to the system, including PVM constants. They are put in a separate file for the readers to customize them to their needs without altering the main file.

- the class TRACE provides switches and routines for printing messages at three levels of description. Users can customize the switches. Setting them all to false will silence the execution (after recompilation of course). On the other hand setting them all to true will report each and every activity. Other cases are in between. Setting the is_descriptive switch to false will cut off detailed information about message passing, but still reports major TractorS activities needed by programmers to follow up. If the is_trace switch is set to false, only the end user type of information is printed. The class is inherited and used by other classes.

- the class MESSAGE_PASSING accommodates the more high level message passing routines. A collection of routines are provided here to send, receive and broadcast messages in different modes, query for information about messages (e.g. if a message from a process and/or with a particular tag has arrived), etc.
Chapter 5. Implementation

The higher level routine `receive_str` (blocking receive \(^1\)) located in the class MESSAGE_PASSING looks like this:

```sather
receive_str(code: INT; tid: INT; message: STR): INT

pre tid >= -1 is
  -- returns bufid (negative if error).
  res:= recv(tid, code);
  if res >= 0 then
    if upkstr(message) < 0 then
      res := -1
    end;
  end;
end; -- receive_str
```

The first line defines the routine with its parameters and the second one defines the routine invariant (pre condition). The parameters are a `code` and `tid` specifying the type and sender task of the message, and `message` which is a string to store the arriving string. The routine calls the interface routine `recv` to receive a string. If successful it calls another interface routine `upkstr` to unpack the received string and put it in the argument `message`.

There are some points to notice. The most important one is to declare long enough Sather variables to accommodate values which go back and forth to/from C. Otherwise the results may not be correct. This is how `receive_str` should be called:

```sather
result:STR := STR::create_sized(bufsize);
receive_str(..., ..., result);
```

The variable `result` is defined long enough to hold the received message. Recall that `bufsize` can be customized by the programmer in class INITIALIZE.

5.2.2 Example: a Distributed Program

Here we give a very simple, but complete example of how the facilities can be put together to program a distributed task. The example is a barrier synchronization between programs running on different hosts. The code appears

\(^1\) The corresponding non-blocking receive routine is called `nreceive_str`
below. It consists of a master and two slave classes. The master class spawns the
slaves at other hosts and waits for them to join the group “tst” and calls barrier
for synchronization. Notice that the error cases are already taken care of in the
SATHER_PVM class which prints error messages if something goes wrong. Also
notice the inheritance of the class SLAVE by SLAVE1 and SLAVE2. The “main”
routine in class MASTER enrolls in PVM using the “mytid” routine, spawns the
two slave tasks using the “spawn” routine (described below), and finally synchro-
nizes the activities by calling “joingroup” and “barrier” routines. The two
slave tasks also enroll in PVM, run the inherited routine “synchronize” (from the
class SLAVE) which joins the group and calls barrier, and exits from PVM. The
class MASTER inherits form the class SATHER_PVM which accommodates
interface routines. The arguments to the routine “main” below should be either
void or name of two hosts.

class MASTER is
    SATHER_PVM;
    const me: STR := "master";
    main(args: ARRAY{STR}) is
        OUT::s("master <host1> <host2>: spawns one slave on each host.");
        host1, host2: STR;
        ids: ARRAY{INT} := ARRAY{INT}::nev(2);
            -- used by PVM to store ids of spawned tasks.
        if args.asize < 3 then
            host1 := void; host2 := void; -- PVM selects hosts
        else
            host1 := args[1];
            host2 := args[2];
        end;
            -- enroll in PVM.
        if mytid < 0 then return; end;
            -- spawn the two slave tasks.
        if spawn("slave1", void, 1, host1, 1, ids) > 0 then
            if spawn("slave2", void, 1, host2, 1, ids) > 0 then
                -- join group and try to synchronize.
                if joingroup("tst") > 0 then
Chapter 5. Implementation

if barrier("tst", 3) >= 0 then
    print_msg("tasks joined. Synchronization successful.");
else
    print_msg("tasks didn't join. Synchronization failed.");
end;
end;
end;
end;
-- exit PVM
exit;
end; -- main

end; -- class MASTER

The classes SLAVE and SLAVE1 appear below. SLAVE is inherited by SLAVE1 and SLAVE2. It provides the routine “synchronize” to be inherited. SLAVE1 simply enrolls in PVM and calls the inherited routine “synchronize”. The main difference between SLAVE1 and SLAVE2 is the actual application dependent task being done in their “work” routines.

class SLAVE is
    SATHER_PVM;
    const me: STR := "slave";
    synchronize is
        -- join group and try to synchronize.
        if joingroup("tst") >= 0 then
            if barrier("tst", 3) >= 0 then
                OUT::s("master & slave joined. Synchronization successful.").nl;
            else
                OUT::s("master & slave didn't join. Synchronization failed.").nl;
            end;
        end;
    end; -- synchronize
end; -- SLAVE

class SLAVE1 is
    SLAVE;

const me: STR := "slave1";

main is

  -- enroll in PVM.
  if mytid < 0 then return; end;
  work; -- do the application dependent task.
  synchronize;

  -- exit PVM
  exit;
end; --main

work is
...
end; --work
end; --SLAVE1

5.3 Details of TractorS Library

Class hierarchy of the TractorS library was introduced in the previous chapter. Here we describe the major design decisions behind it. Some aspects of the classes and the way things are designed and implemented are described in this section in an order which makes things more understandable.

5.3.1 Atomic Transactions

The class ACTION supports ordinary atomic transactions and the class COMPENSATABLE.ACTION supports compensatable ones. Object oriented design and abstract data type concepts are used to free the programmers, as much as possible, from details. Instance variables and all sorts of routines like error handling, value and commit dependency, decision making, and even "main" are located in the classes to be inherited. The only part left for the programmers to code is the "work" routine for each task which does the actual application-dependent job. Dependencies, as already discussed, only need minor value settings in the "work" routines (the pre-defined routines are actually called with these values as their parameters). The routine "main" in this class is a
common wrapper for the top level action performed. It calls a number of other routines to follow up the action, and finally reports the result to the routine “decision” which cooperates with the top-level transaction to make the final decision. The two routines “main” and “decision” which are more high level than the rest are described here.

The routine “main” appears below. Its argument is an array of strings. Recall that Sather provides routines to convert strings to other types. After reporting the start of the transaction, an object of the same type as the defined class (SAME) is created. The arguments to the “create” routine are changed from array to list for which Sather provides various functions and makes programming easier and more compact. The object then enrolls in PVM by calling the routine mytid from the Sather_PVM interface class. If successful, it checks for the parent, i.e., the task which spawned it (negative result means no parent, i.e., top-level transaction). If no parent, it begins a rudimentary (single task) top-level transaction which either safe-commits or is undone depending on the result of the work routine. Otherwise the routine “decision” is called to synchronize with the parent and the top-level task, and the result is passed to it. The exit routine appearing at the end is for exiting from PVM. This is important because if the task halts without informing PVM, delayed messages will not be sent. The routine is protected against TIMEOUT condition, for which a particular handler is defined (see [87] for exception handling in Sather).

```sather
main(args: ARRAY{STR}) is
  -- This is a common wrapper for the top level action performed
  -- by ‘self’. All transactions check their arguments first and
  -- do some common error handling. Optionally tracing is supported
  -- based on the boolean ‘is_trace’.
  -- Customize this behavior by redefining the consts as well as the
  -- routine ‘work’ to do the real things that each routine should do
  -- and communicate the results.
  protect
  print_info(STR::create.s("BEGIN ATOMIC TRANSACTION ");
  t: SAME := create(to_list(args));
  if mytid < 0 then
    print_info(STR::create.s("atomic transaction cannot start: ABORT");
    return;
```

Chapter 5. Implementation

116
The routine "decision" appears below. After communicating the result back and passing the possible commit signal and/or produced value to the consumers, it attempts for global synchronization (see comment lines in the routine below). No action should take place if the task has either failed or can commit unsafely due to the termination of the subtransaction (synchronization does not obviously apply to terminated tasks). Otherwise it joins the group "main_group" and waits for a message with a special tag for synchronization. The responsible transaction sends such a message. If the message is "commit" then it tries for the barrier synchronization. If all group members pass the barrier, the final agreement is reached and final commitment is done; otherwise the transaction is undone.

\[
\text{decision(message: STR) is}
\]

-- decision for global commit synchronization.
agree := STR::create_sized(bufsize);
-- communicate the result back.
send_msg(res_tag,my_parent,message);
-- pass the value to the consumer.
if is_producer then
  pass_values;
end;
Chapter 5. Implementation

-- pass the commit signal to the consumer.
if is_commit_dep then
    pass_signal;
end;

-- global synchronization.
if message.is_equal(fail) then
    undo;
elsif is_compensatable then
    unsafe_commit;
elsif joingroup(main_group) >= 0 then;
    print_msg(STR::create.s("waiting for global synchronization.
"));
    receive_msg(synch_tag, -1, agree);
    if agree.is_equal(commit) then
        if barrier(main_group, -1) >= 0 then
            print_msg(STR::create.s("reached final agreement; COMMIT.
"));
            safe_commit;
        else
            print_msg(STR::create.s("cannot reach agreement; ABORT.
"));
            undo;
        end;
    else
        print_msg(STR::create.s("decision is to ABORT.
"));
        undo;
    end;
else
    print_msg(STR::create.s("decision is to ABORT.
"));
    undo;
end; -- decision

There are other routines for checking arguments, passing values and signals back and forth, failure and timeout handling, etc. Various instance variables and constants are also defined. For example there are three “tag” constants for three types of messages. They are listed below. They are used in communicating messages for synchronization, result of the task, and value dependency respectively. Also the constants commit, ready, and fail used in the above routines appear below:

const synch_tag: INT := 1;
const res_tag: INT := 2;
const value_dep_tag: INT := 3;
const fail: STR := "fail";
const ready: STR := "ready_to_commit";
const commit: STR := "commit";
const is_compensatable: BOOL := false;

Atomic transactions may possibly be written in languages other than Sather. Any language with C interface to PVM can be used for this purpose.

5.3.2 Schedulers

Two types of classes support schedulers: the class SCHEDULERS which supports the common aspects of them all, and the four classes SERIAL, PARALLEL, SERIAL_ALTERNATIVE, and PARALLEL_ALTERNATIVE for each specific type.

In the class SCHEDULERS things like defining subtransactions, setting the corresponding hosts, spawning the subtransactions, receiving and checking the results, global commit protocol, etc. are done. Again we present the routines "main" and "decision" which are the first and last tasks and are more high-level.

In the "main" routine appearing below an object is created and the process is enrolled in PVM by calling the interface routine mytid. Also it calls the interface routine parent to get the id of the parent task which has spawned it. If no parent, it begins a top-level transaction and defines (and joins) the group for globalynchronization. It then calls the user-defined routine "work" and sets the subtransactions and the corresponding hosts accordingly. From here on the type of scheduler is important. The abstract (non-instantiable) class SCHEDULERS is inherited by all specific scheduler classes, so "main" is the common main routine in those. The routine "exec" which is defined differently in each type of scheduler is called to execute the subtransactions in the order dictated by semantics of the type (we describe one of the "exec" routines below), and the result is passed to the routine "decision" for final decision.

main(args: ARRAY{STR}) is
  t: SAME := create(to_list(args));
  if mytid < 0 then -- enroll in PVM.
Chapter 5. Implementation

return;
end;
t.my_parent := parent;
if t.my_parent < 0 then -- top-level
    print_info(STR::create.s("BEGIN TOP_LEVEL TRANSACTION ");
    if joingroup(main_group) >= 0 then
        inc_group_size;
    else
        return;
    end;
t.print_type;
end;
t.work;
t.set_subs;
t.set_hosts;
t.decision(t.exec);
exit; -- exit from PVM.
if t.my_parent < 0 then
    print_info(STR::create.s("END TOP_LEVEL TRANSACTION ");
end;
against TIMEOUT then timeout_handler;
end; -- main

The routine "decision" appears below. The final decision is made independent of the type of scheduler. If the result of the task (taken from "exec") shows a failure, it is broadcasted to all possible group members waiting for global synchronization (notice the type of tag in the broadcast message) to abort their tasks. If the task succeeds, depending on being top-level or not, either the global commit protocol is activated or the result as well as the locks are passed to the parent as in the conventional nested transaction model. The actual global commit needs further action which is taken care of by the routine commit_protocol. Notice that intermediate schedulers do not access database items and do not need to take part in the final commit protocol.

decision(result: STR) is
    -- make the final decision in a scheduler.
    if result = fail then
if get_group_size > 1 then
    broadcast_msg(synch_tag, main_group, fail);
end;
print_msg(STR::create.s("incomplete task; ABORT.");
undo;
elseif my_parent < 0 then --top_level transaction
    if not commit_protocol then
        broadcast_msg(synch_tag, main_group, fail);
        print_info(STR::create.s("global commit failed; ABORT.");
        undo;
    end;
else
    send_msg(res_tag, my_parent, result);
    pass_lock(my_parent);
end;
end;--decision

As an example of the specific scheduler classes, we discuss some aspects of the
class PARALLEL_ALTERNATIVE. Besides the instance variables and rou­
tines inherited from the above-mentioned class SCHEDULER, there are routines
in this class to execute subtransactions, follow them up and get their results back.

As an example we present the routine “exec” which is fairly high level. In
its simple loop below, all subtransactions are spawned (in the order defined by
the programmer in the “work” routine). If none is successfully spawned, a failure
result is returned (which is the basis for the final decision). Otherwise the routine
“get_a_result” is activated which uses the buffer information and probe facilities
of PVM to find out if any of the tasks has sent a successful result back. If
so it aborts other subtransactions due to the “one is enough” semantics of the
schedule. Otherwise it tries again until they all terminate.

eexec: STR is
    res := fail;
    print_msg(STR::create.s("spawn subtransactions in parallel.");
i: INT;
loop until!(i > sub_idx);
    if spawn_sub(i) > 0 then res := ready; end;
i := i+1;
Each type of scheduler has its own complexities. Their semantics are different and what they accomplish varies. For example the serial scheduler supports argument dependency while the parallel one supports value and commit dependency. They all may have delegated as well as compensatable subtransactions. Users should investigate TractorS library to see how things are implemented. The pieces of code presented here are just a sample.

5.3.3 Delegation

Delegation is introduced in several places throughout the thesis. Here we show some details of how the class DELEGATE provides meta-knowledge about services provided by sites, which help programmers direct tasks to appropriate ones. Two types of information are provided:

1. information about the sites, clusters, and their relationship.

2. information about services provided at each cluster.

When a service is needed, programmers may request the system to look up the sites which may be able to provide it, as well as their relationship to the current site. This is done via routines, a sample of which is presented in table 4.1 in the previous chapter. Depending on the type of transaction, it is then decided to which site to direct the service. For example, programmers may define a single task as children of a parallel-alternative scheduler to try them at a number of sites, get the quickest response, and abort the rest. The abstract data types help concentrating on what the routines provide rather than how they do it.
Due to the significance of unsafe-commit in delegated transactions which helps releasing locks of partially independent tasks, the compensatability property of services are attached to them. Services are defined as a tuple \((\text{service}, \text{host}, \text{is\_compensatable})\). The class \text{SERVICE} provides facilities to define and check properties of services. Notice that users are only allowed to alter certain classes, mainly the class \text{INITIALIZE} and their own user-defined ones, and hence they cannot add or delete services or host. Such facilities are for authorized users only. A portion of the class \text{SERVICE} (inherited by \text{DELEGATE}) which helps assigning services to sites is shown. Notice that despite procedural languages, in object oriented ones typically the same name can be used multiple times in the same routine with different meanings (e.g. \text{"host"} in the routine \text{service\_assign} is an attribute of the object and a parameter to the routine at the same time).

\begin{verbatim}
class SERVICE is
  -- name of the service together with the host providing it
  -- and its compensatability condition.
  attr service: STR;
  attr host: STR;
  attr compensatable: BOOL;
  ...

  service_assign(service,host:STR, compensatable: BOOL): SAME is
    -- assign services to hosts.
    if is_host(host) then
      res := new;
      res.service := service;
      res.host := host;
      res.compensatable := compensatable;
    else
      print_msg(STR::create.s(host).s('is an unknown host.'));
    end;

end; -- service_assign
...
end; -- class SERVICE
\end{verbatim}

A hash table of service objects at each site keeps track of the services provided by that site. Auxiliary arrays help define relationship between hosts. Routines are provided to add, delete, and change aspects of services.
As an example of an interface routines, the code for `is_close_acquaint()` is shown below. The routine checks whether its argument is a close acquaintance of the current site. The cluster that the two hosts belong to are looked up first. If not the same, then the table which holds the information about relationship between sites is looked up. The routine can be called multiple times to see whether any two sites are close acquaintances.

```plaintext
is_close_acquaint(host:STR): BOOL is
    -- whether 'host' is a close acquaintance of the current host.
    -- the array close_acquaintances[] defines the relationship.
    c_host: STR := STR::create_sized(bufsize);
    SATHER_PVM::current_host(c_host);
    idx1: INT := which_cluster(c_host);
    idx2: INT := which_cluster(host);
    if idx1 = idx2 then
        res:= true;
        return;
    end;
    -- they are not in the same cluster.
    i,j,k: INT;
    loop until!(i = acq_row);
        j:= 0;
        loop until!(j = acq_col);
            if close_acquaintances[i,j] = idx1 then
                k:= 0;
                loop until!(k = acq_col);
                    if close_acquaintances[i,k] = idx2 then
                        res:= true;
                        return;
                    end;
                    k:= k+1;
                end;
            end;
        end;
    end;
    j := j+1;
```
Chapter 5. Implementation

5.3.4 Transactional Events

The four termination events are safe-commit, undo, unsafe-commit, and cancel. While most of such activities are local to components of CIS, there are classes to provide special TractorS needs. The two events unsafe-commit, and cancel are special due to their new semantics in TractorS. Each action is described below.

Currently safe-commit only happens whenever all the atomic transactions in the context of a top-level transaction are done. At this stage the compensatable ones are already committed, but others have successfully carried out their tasks without releasing their locks. A mechanism based on the two-phase commit idea takes care of global synchronization. If successful, all data items are released and all logs are discarded; otherwise Undo and/or Cancel actions are activated.

Undo is necessary in the following cases:

- nothing is wrong but some subtransactions should be undone. This happens in parallel-alternative schedulers where all subtransactions should be undone except the one completed first. Subtransactions could possibly be in one of three stages when this happens: active, ready-to-commit, or already committed unsafely. The last two should be rare, unless the communication is too slow, due to the fact that completion of the first subtransaction will cause immediate abortion of others. Hence unreasonable cancel procedures do not happen frequently.

- the parent decides to abort subtransactions due to the failure of the whole task. This happens if none of the subtransactions of alternative schedulers succeed or any of the subtransactions of a serial or parallel scheduler fail (not completed before the timeout period or some sites refuse to give the service). This is handled similarly to the first case. The global transaction does not necessarily fail.
• a subtransaction is aborted by a site after announcing its "ready to commit" state. Due to the properties of groups in PVM the top-level transaction will not be completed. The PVM barrier mechanism is used to make sure that all members of the group stay alive till the end. This case also demands a similar abort action.

• a client or server site recovers from a crash. The run-time system aborts incomplete subtransactions and restarts them if no abort or timeout message is received. This works fine since the databases are restored and the possible restarted subtransactions proceed as usual. Compensatable subtransactions which have done their tasks are not in-progress, their locks are already released, and the data items they have updated are not (and should not be) restored by the recovery procedure.

In unsafe-commit the data items are released to all other transactions. The subtransaction terminates successfully, but the cancel-log manager keeps enough information about all such committed transactions and their compensating counterparts, together with their parameters as well as certain dependencies between them. If the top-level transaction safe-commits later, the log is discarded; otherwise the cancel procedure is activated.

Cancel should take place whenever some subtransactions of an aborted top-level transaction have committed unsafely. Using the cancel-log, the compensating transaction is formed, executed, and committed. Only one compensating transaction is needed for all unsafe-committed subtransactions of each top-level transaction. We also stated the orders to consider in scheduling compensating counterparts. Clearly compensation is done in the context of a single top-level transaction, i.e. compensatable subtransactions are independent of all other top-level transactions that may have committed in the mean time.

As an example the "main" routine in the class CANCEL is presented here. It takes the identification of the failed top-level transaction as input and uses the cancel-log. Like any other task it tries to enrol in PVM for communication and synchronization. However, if it cannot do so, it passes enough information to the programmer to be able to start the task again because it has to be done. If the enrollment is successful, it continues by reading the cancel-log. The procedure extract_log() returns a list containing the begin record as well as all subtransaction
records (second type in the log) belonging to the top-level transaction. The procedure \texttt{form\_comp\_trans} called next is of particular importance. It takes the above list as input and returns a list of compensating transactions as output. Each member of the output list is of the following type:

\begin{verbatim}
class CANCEL\_REC is
   attr is_serial: BOOL;
   attr compensating: STR;
   attr comp\_args: LIST\{STR\};
   attr host: STR;
\end{verbatim}

The first declaration shows whether the subtransaction should go in a serial order. Others give the name, parameters, and the host that committed the subtransaction unsafely. The compensating counterparts appear in the right order in the list, i.e:

- the ones which should go serially appear in the reverse order of the corresponding subtransactions in the top-level hierarchy.
- all records with a \texttt{true} in their \texttt{is\_serial} declaration appear before any record with a \texttt{false} statement. This is to take care of the ones that should go serially first, and then execute the rest concurrently.

The rest of the routine "main" spawns the subtransactions in the right order at their hosts and follows up their execution. It uses the facilities provided in classes SERIAL and PARALLEL. Attributes are set in the class CANCEL to make the link. If anything goes wrong, the routine asks for help and gives the users the necessary information to start the task again. Otherwise it outputs the successful end of the cancel process.

\begin{verbatim}
main(args: ARRAY\{STR\}) is
   -- this routine governs the top-level task of forming and
   -- executing a cancel process.
   print\_info(STR::create.s("BEGIN CANCEL TRANSACTION FOR TASK ").s(argv[1]));
   t: SAME := create(args);
   if mytid < 0 then
      print\_info(STR::create.s("cancel transaction cannot start because it
       cannot enrol in PVM. Should be RESTARTED with argument").s(argv[1]));
\end{verbatim}
return;
end;
all:LIST{STR}:= LIST{STR}::create;
all:= extract_log(argv[1]);
c: CANCEL_REC;
cancel_list:LIST{CANCEL_REC}:= LIST{CANCEL_REC}::create;
cancel_list := form_comp_trans(all);
success: STR := "commit";
spawn_ids: ARRAY{INT}; --needed by PVM.
i: INT := 0;
loop while!(cancel_list[i].is_serial); -- for serial ones
  c := cancel_list[i];
  if spawn(c.compensating, c.comp_args, 1, c.host, 1, spawn_ids) < 0 then
    success := "fail";
    break;
  end;
  success := SERIAL::get_a_result;
  if success = "fail" then
    break;
  end;
i := i+1;
end; --loop
loop until!(i > cancel_list.size - 1); -- for remaining parallel ones.
if success = "fail" then
  break;
end;
c := cancel_list[i];
if spawn(c.compensating, c.comp_args, 1, c.host, 1, spawn_ids) < 0 then
  success := "fail";
end;
i := i+1;
end; --loop
success := PARALLEL::get_all_results;
if success = "fail" then
Chapter 5. Implementation

```
print_info(STR::create.s("cancel transaction failed in the middle. Should be RESTARTED with argument").s(argv[1]));
else
  print_info(STR::create.s("END CANCEL TRANSACTION FOR TASK ").s(argv[1]));
end;
exit;
end; --main
```

Concurrency Control for File Systems

While most of the CIS components have their own local transaction managers, file systems do not. Unix locking mechanisms however can be used for this purpose. Different facilities are provided in Unix. The high level SCCS (source code control system) utilities are aimed at locking a whole file and controlling access to its different versions. The `flock()` and similar routines aim at single host problems. The best tool for a distributed environment such as TractorS which needs to lock portions of files remotely is the network lock manager that supports file and record locking over the network. It allows cooperative processes to synchronize access to shared files via `lockf()` and `fcntl()`.

The user calls to `lockf()` and `fcntl()` are mapped to RPC-based messages to the local lock manager at the target host. The main problem in locking across multiple machines is occurrence of crashes. In the case of a server crash, client applications will sleep until it comes back up and their operations can complete. It however loses all its lock information when it recovers. On the other hand if the client has crashed, the lock can be held for ever by the server. The network lock manager solves these problems by cooperating with the network status monitor to ensure notifying relevant machine crashes. It has protocols to recover the necessary lock information when crashed machines recover. At each server site, a lock manager process accepts local as well as remote lock requests. The client and server lock managers communicate with RPC calls. The key in this approach is the network status monitor which helps the lock manager to detect and recover machine failures.

The routine `fcntl()` is a record locking facility which provides shared or exclusive locks. It performs a variety of functions on descriptors including access...
modes (read, write or read/write), set or clear a file segment lock, and specify processes to receive signals. We refer readers to the corresponding *man pages* of the Unix system for details. The followings are important to notice about the behavior of the lock manager for synchronizing client/server operations:

- when a client crashes, the lock managers on all of its servers are notified, and they release all associated locks on the assumption that it will request when it needs them again. When a server crashes, the clients will wait for it to come back up, and when it does, its lock manager will give the client lock managers a grace period to submit lock reclaim requests, during which only such requests will be accepted. The client lock managers will be notified when the server recovers. There is a *timeout* option to retransmit lock requests to the remote server.

- if a client is not able to recover a lock that it had on a crashed server, a signal is sent to the process (the default action is to kill the application).

- the lock manager does not reply to local and new server lock requests until the old server lock manager has gotten back to it.

By means of lockf() one may place, remove, or test for exclusive locks. These locks are either advisory or mandatory. If a process holds a mandatory exclusive lock, all read and write accesses to the segment block until the lock is removed. Such a lock is considered dangerous because it can cause the entire system to hang or crash if such a lock is held by an out-of-control process. For this reason only advisory locks are used. An advisory lock may be used by cooperating processes which observe read and write restrictions voluntarily, however, a locking call on an already locked file section fails. The lockf() function has descriptors for checking and acquiring locks for shared or exclusive use. The scheme is deadlock-free; if the danger exists, it returns an error value without putting the process in the queue.

### 5.4 Linking TractorS to the External World

In this section we discuss how TractorS can be linked to external world information systems and report our experiments. TractorS, written in Sather,
produces portable C code. On the other hand lots of the existing real world information systems either have C-interface or interfaces which provide C code (e.g C++ interface). Hence it is not difficult to make a link to TractorS. Two cases are discussed in the following two subsections below, one from each major paradigm. The first case is a public domain object-oriented information system. We have installed and made some experiments with this system to make sure that things work properly. The second one is a relational database. The main goal behind this choice is to show that the script language for TractorS interfaces the heterogeneous environments.

It should be noted that existing interfaces for distributed information system environments do not provide the full functionality required by TractorS (after all, our main goal has been to overcome drawbacks of the existing methods). For instance the most popular one is the XA interface [88] which is part of the Common Application Environment (CAE) by X/Open, a worldwide, open systems organization supported by most of the world’s major information system suppliers and software companies. CAE attempts to combine existing and emerging standards into a usable system environment, in order to provide portability and interoperability of applications. The XA interface is a facility for commercial applications to achieve distributed transaction processing on Unix system. It is a bidirectional interface between a transaction manager and resource manager. The interface is already in use by major database and information systems. In such an environment an application program defines transactions and their boundaries by calling resource managers and transaction managers (which have mutual interface as well). The XA interface is a system-level interface between these components. A transaction is an atomic one here. Global transactions are provided but they are also atomic. It is interesting however that some of the ideas in advanced transaction models are partially supported. For example a subtransaction which has reached the ready-to-commit stage may be committed (similar to unsafe-commit in TractorS) or rolled back by its resource managers independently of the corresponding transaction manager (called heuristic decision-making). The inconsistency is rectified later on by appropriate actions. The transaction manager may accept the decision or reject it by calling a xa_forget service [88]. Restrictions however do not allow enough flexibility. For instance an application can
Chapter 5. Implementation

not have more than one global transaction in progress which means that ideas such as delegation and value dependency cannot be easily implemented.

5.4.1 OBST: An Object Oriented Persistent Storage System

OBST is a public domain object-oriented persistent storage system. Since end 1990 the first prototype of OBST is available and is shipped to interested universities and research institutions. The current version (OBST3-3.5 [89]) is publicly available via FTP (see below). OBST was developed by Forschungszentrum Informatik (FZI) as a contribution to the STONE project (supported by the German Ministry for Research) and was originally designed to serve as the common persistent object store for the tools of a software engineering environment.

OBST is not implemented as an enhancement to existing programming languages. It is designed to be independent from any host language. The embedding into a host language is done by generating appropriate interfaces from OBST schemas. An embedding into C++ is implemented and one for Common Lisp and other languages are planned [89].

The OBST data model provides mechanisms for defining types in terms of modules called schemas. Several generic types are predefined, like Set, List, or Array. External types are used to manage the link to types of object-oriented programming languages. The notion of class is central to OBST. Class inheritance is a mechanism for factoring out common properties of classes in parent classes. All classes have a common parent class. Parameterized classes are supported. OBST comes with several predefined types. An important one which should be noticed is the type container which serves as an interface to the persistence object store manager. There exists a distinguished container, called root container which contains a root object that serves as entry point to an OBST database. When an object is created, it is associated with exactly one container. All operations that modify the value of an object require the corresponding container to be opened first. The containers provide mechanisms for synchronization, recovery, and clustering of objects, transaction management (e.g. start, commit, abort) and locking data items with a mechanism for breaking deadlocks arising from multiple
access to the same set of containers.

The system comes with the schema compiler, a library of predefined classes, a graphical object browser, the structurer, etc. It provides all manuals and a very helpful tutorial tool. For the installation of OBST, a C++ compiler and the X-Windows system for the graphical tools are required. BST3-3.5 is now available at ftp.fzi.de under /pub/OBST/OBST3-3.5. Installation has been tested for SunOS4.1.3, Solaris 2 and LINUX.

OBST provides a mechanism to incrementally load methods. This enables programs to deal with objects whose type is defined after the program itself has been developed. This is useful in systems that supports schema evolution.

In summary, the OBST data model can be characterized by the following properties:

- schema definition language syntactically similar to C++;
- support of multiple inheritance;
- generic classes;
- abstract classes and methods;
- distinction between public, protected, and private methods;
- redefinition of methods;
- overloading of methods;
- persistence;
- transaction management facilities.

We have installed OBST at our institution and made some experiments in linking it to the scripting language for TractorS. We created a simple object store and executed primitive atomic transactions, written in C++. These transactions were successfully linked with PVM for communication and synchronization purposes. Transactions were separately compiled and objects stored in TractorS bin directory. We then combined them into a hierarchy of subtransactions by calling the as subtransactions into script classes, similar to the ones written for the case
study. The dynamic group mechanism for synchronization worked out easily be-
cause both languages, Sather and C++, called PVM routines with no problem.
All these components actually provided C codes, which were put together in a
transaction tree.

5.4.2 ORACLE: A Relational Database Management Sys-
tem

ORACLE is a popular relational database management system. Interactions
with it is basically done via SQL (Standard Query Language). SQL is a non-
procedural language. Although it is a very powerful one, it has some limitations
without procedural capabilities. Understanding its limitations, the originators
of SQL also explicitly designed SQL constructs to be embedded in procedural
languages (called the host language) such as C. The combination is obviously
more powerful than SQL or C alone.

ORACLE includes several tools to allow programmers to write transactions
in a host language, including C. For example the Pro*C tool [90] provided with
ORACLE is designed to convert a C program with SQL statements into a C pro-
gram which accesses and manipulates data in the database. It converts the SQL
statements to appropriate C routines. Another tool provided by ORACLE for
accessing data items of the database in procedural languages is called ORACLE
Call Interface [91]. Programmers embed ORACLE calls directly in high level
languages. Transactions are accomplished through multiple calls.

Pro*C calls have some benefits over ORACLE Call Interface calls. For in-
stance they are more conceptual and easier to understand due to the separation
of database accesses from other routines. Also they are automatically translated
to the equivalent of several run-time library calls, reducing programming time
[90]. Any valid SQL statement may be executed from a C program. It does a fair
amount of work on behalf of the programmer. Statements such as COMMIT
and ROLLBACK are provided for transaction management.

Due to the fact that Sather programs are compiled into portable C code,
it can be easily seen how a C program which accesses ORACLE database is
linked to the script language for TractorS. It is similar to linking to any other
C program, including PVM to which we have developed an interface. Since
atomic transactions are building blocks of our model, they can be programmed in such an environment, linked with the PVM library for communication and synchronization, and called in the user-defined classes of the script language. At run-time they are actually not different from the ones written in Sather itself, because Sather is compiled into C. It is interesting to observe how a relational database is linked with an object-oriented environment.

5.5 Achievements Gained Through Building a Prototype

Building the TractorS prototype was the most valuable experience in this project. After the feasibility study and analysis phases, it started by programming in Sather which was a developing object-oriented language at that time. It followed by implementing complex programs. Having interactions with one of the original designers of the language as my supervisor was very constructive. I had the chance to learn how to handle complex tasks in an object-oriented environment. The power of simplifying, structuring, hiding details of tasks from end-users in an abstract data type manner, and so on in this paradigm was surprising compared to my prior experience in using a number of procedural languages (the latest one being Modula-2).

Designing and programming distributed computation was very interesting too. There was a number of choices available. Experiencing them all was impossible, however investigated several, experiences two, and eventually chose PVM. Getting experienced with PVM and learning most of its aspects by writing programs in the C language helped a lot in linking it to the rest of the prototype.

The next step was building the Sather-PVM interface. Extending the language to support such a major task looked hard at the beginning, but went smoothly and was a useful experience. Simple programs were developed quite easily, but problems arose when complex tasks interacted. A deep understanding of both Sather and PVM was required to handle the situation. It ended successfully.

The most important, and hardest part, was the design and implementation of advanced transaction management classes. The task involved a range of varied
and difficult activities. Just as an example, the way to handle multiple messages arriving from different sources to a parallel-alternative scheduler, including from its children and parent, was hard. Messages should be distinguished and proper actions which depended on previous and future messages should be taken.

Over all, I was very happy with the experience and the result.
Part III

TractorS Formalization

Chapter 6

Formal Semantics

In this chapter TractorS is expressed in a formal manner in order to eliminate ambiguities and to facilitate its implementation and other models. We intentionally do not relate the formal semantics of the model to the message-passing paradigm to allow various implementation options. The ACTA meta-model [92], [93], [94] is used to specify TractorS.

6.1 A Short Summary of ACTA

The ACTA meta-model \(^1\) is a notation for formal specification and analysis of transaction models. In ACTA a transaction model is characterized by the interactions of concurrent transactions as well as objects. Visibility properties of transactions are defined in terms of a view for each transaction which is the state of objects visible to it at any point in time. Similarly a conflict set is defined as the set of in-progress operations which have potential conflicts with the transaction. Also a dependency set is defined as a set of inter-transaction dependencies developed during the concurrent execution of transactions. Such dependencies result either from structural properties of transactions or are developed as a result of their interactions over shared objects during their execution. Different types of dependencies are defined regarding the order of execution, commitment and abortion orders of concurrent transactions. The effects of transactions on objects is specified in terms of conflicts between operations invoked by concurrent trans-

\(^{1}\) Some aspects of ACTA have changed in course of time. We use the latest version presented in [94].
Chapter 6

Formal Semantics

In this chapter TractorS is expressed in a formal manner in order to eliminate ambiguities and to facilitate its comparison with other models. We intentionally do not relate the formal semantics of the model to the message-passing paradigm to allow various implementation options. The ACTA meta-model [92], [93], [94] is used to specify TractorS as a transaction model. A short summary follows.

6.1 A Short Summary of ACTA

The ACTA meta-model ¹ is a notation for formal specification and analysis of transaction models. In ACTA a transaction model is characterized by the interactions of concurrent transactions as well as objects. Visibility properties of transactions are defined in terms of a view for each transaction which is the state of objects visible to it at any point in time. Similarly a conflict set is defined as the set of in-progress operations which have potential conflicts with the transaction. Also a dependency set is defined as a set of inter-transaction dependencies developed during the concurrent execution of transactions. Such dependencies result either from structural properties of transactions or are developed as a result of their interactions over shared objects during their execution. Different types of dependencies are defined regarding the order of execution, commitment and abortion orders of concurrent transactions. The effects of transactions on objects is specified in terms of conflicts between operations invoked by concurrent trans-

¹some aspects of ACTA have changed in course of time. We use the latest version presented in [94].
actions. The ACTA model encompasses both object-specific and transaction-specific semantics. There is a transaction responsible for ending each operation in ACTA. The notion of delegation is used to delegate such a responsibility from a transaction to another which affects the above aspects of transactions as well. Here we summarize the basic definitions as well as the the characterization of atomic transactions as described in [94].

**Definition 1** Invocation of any operation by transaction \( t \) is called an Event. \( E_t \) denotes the set of all events that can be invoked by \( t \) which is the union of the following two disjoint sets:

1. \( O E_t \): Invocation of an operations \( p \) on object \( ob \) by transaction \( t \) is called an object event and is denoted by \( pt[ob] \). The set of object events that can be invoked by transaction \( t \) is denoted by \( O E_t \).

2. \( S E_t \): Invocation of transaction management primitives by a transaction is called a significant event. The set of significant events that can be invoked by transaction \( t \) is denoted by \( S E_t \) with two disjoint subsets:
   
   (a) initiation events \( (I E_t) \) that are invoked to initiate a transaction .
   
   (b) termination events \( (T E_t) \) that are invoked to terminate a transaction.

   Notice that \( (I E_t \cup T E_t) \subseteq S E_t \)

Events invoked by any two transactions are either disjoint or nested, i.e

\[ t \neq t' \Rightarrow (E_t \cap E_{t'} = \phi) \lor (E_t \subseteq E_{t'}) \lor (E_{t'} \subseteq E_t) \]

Using the above, the term transaction is formally defined in ACTA as:

**Definition 2** Execution of a transaction \( t \) is a partial order of events \( E_t \subseteq (O E_t \cup S E_t) \) with relation \( <_t \) denoting temporal order of events invoked.

The concept of history defined below helps to characterize concurrent transactions:

**Definition 3** A history \( H \) of a set of concurrent transactions indicates a partial order of all the events associated with those transactions. The current history \( H_{ct} \) is a subset of \( H \) that occur until a point in time.
Precedence is defined as:

**Definition 4** The predicate $\epsilon \rightarrow \epsilon'$ is true if event $\epsilon$ precedes event $\epsilon'$ in a history $H$. It is false otherwise (thus, $\epsilon \rightarrow \epsilon'$ implies that $\epsilon \in H$ and $\epsilon' \in H$).

Any transaction satisfies the following **fundamental axioms of transactions**:

**Definition 5** Let $t$ be a transaction and $H^t$ the projection of the history $H$ with respect to $t$.

1. $\forall \epsilon \in I E_t(\epsilon \in H^t) \Rightarrow \exists \epsilon' \in E_t(\epsilon' \rightarrow \epsilon)$

2. $\forall \epsilon \in T E_t \exists \epsilon' \in I E_t(\epsilon \in H^t) \Rightarrow (\epsilon' \rightarrow \epsilon)$

3. $\forall \epsilon \in T E_t(\epsilon \in H^t) \Rightarrow \exists \epsilon' \in E_t(\epsilon' \rightarrow \epsilon)$

4. $\forall \text{bob} \exists p(p_t[\text{bob}] \in H) \Rightarrow ((\exists \epsilon \in I E_t(\epsilon \rightarrow p_t[\text{bob}])) \land ((\exists \epsilon' \in T E_t(p_t[\text{bob}] \to \epsilon')))$

Axiom 1 states that a transaction cannot be initiated more than once and an initiation event is the first event of a transaction. Axiom 2 states that if a transaction has terminated, it must have been previously initiated. Axiom 3 states that a transaction cannot be terminated more than once and a termination event is the last event of a transaction. The last axiom states that only in-progress transactions can invoke operations on objects.

**Definition 6** Transactions that are initiated and not yet terminated are called **in-progress**.

The terms *commit* and *abort* are defined as:

**Definition 7** The effects of an operation $p$ invoked by a transaction $t$ on an object $\text{ob}$ are made permanent in the database when $p_t[\text{ob}]$ is committed and are obliterated when $p_t[\text{ob}]$ is aborted.
Commit and abort of a transaction in ACTA depend on the way the transaction model defines them. Commit generally means the successful end of a transaction so that its effects on databases will not be lost and the data items are released. Abort generally means the unsuccessful end of a transaction so that its effects are somehow undone. We use the same concept in TractorS. We define two different types of action completion and failure, but commit and abort convey the general meaning as in ACTA.

Dependencies also provide a convenient way to reason about the behavior of concurrent transactions. The notion is defined in ACTA as:

**Definition 8** Dep-set is a set of inter-transaction dependencies developed during the concurrent execution of a set of transactions. Dep-set\(_{ct}\) (the current dependency-set) is a subset of the Dep-set until a point in time.

Dependencies control the order of certain events in a history. They are the basic mechanism in ACTA to enforce the commit and abort ordering of concurrent transactions and subtransactions. Various types of dependencies between transactions in TractorS are defined later.

Conflict between transactions is characterized in terms of conflict sets defined as:

**Definition 9** Conflict-set\(_t\) is the set of those in-progress operations of transaction \(t\) with respect to which conflicts have to be determined. Conflict-set\(_t\) is related to the events in \(H_{ct}\) and dependencies in Dep-set\(_{ct}\).

In ACTA transaction effects on objects are expressed in terms of its view and access set. The status of an object with respect to a transaction depends on whether the object is in its view or access set which are in turn related to the concept of responsibility for operations. The three terms are defined here:

**Definition 10** \(\text{Responsible}_t(p_{t_i}[ob])\) identifies the transaction responsible for committing or aborting \(p_{t_i}[ob]\) with respect to the current history \(H_{ct}\).

**Definition 11** Access-set\(_t\) is the set of all invoked operations for which \(t\) is responsible, i.e., \(\text{Access-set}_t = \{p_{t_i}[ob]|\text{Responsible}_t(t_i)\}\)

**Definition 12** View\(_t\) specifies the states of objects visible to transaction \(t\) at a point in time.
Correctness of the results of transactions are also formalized in ACTA. A transac­tion produces correct results if all the objects in its access-set are atomic upon its end (commit or abort). An object is atomic if it is serializable and behaves correctly. We refer the readers to [94] for formalism of serializability but repeat the definition of correct behavior here:

Definition 13 An object \( ob \) behaves correctly if and only if
\[
\forall t_i, t_j, t_i \neq t_j, \forall p, q
\]
\[
(return-value-dependent(p, q) \land (p_i[ob] \rightarrow q_j[ob])) \Rightarrow
\]
\[
(Abort[p_i[ob]] \in H^{(ob)} \Rightarrow Abort[q_j[ob]] \in H^{(ob)})
\]

This definition implies that for an object to behave correctly it must ensure that when an operation aborts, any return-value-dependent operation that follows it must also be aborted. This definition assumes immediate effects of operations on objects.

6.1.1 Characterization of Atomic Transactions

Atomic transactions are building blocks of nested transactions in TractorS and they are the only ones invoking events on objects. For the self-contained presentation, we briefly describe how atomic transactions are characterized by ACTA formalism and refer the readers to [94] for more details.

Definition 14 Let \( t \) denote an atomic transaction.

1. \( SE_t = \{\text{Begin}_t, \text{Commit}_t, \text{Abort}_t\} \)
2. \( IE_t = \{\text{Begin}_t\} \)
3. \( TE_t = \{\text{Commit}_t, \text{Abort}_t\} \)
4. \( t \) satisfies the fundamental axioms of transactions; i.e.

- \( \text{Begin}_t \) is the first and \( \text{Commit}_t \) or \( \text{Abort}_t \) is the last event invoked; i.e.,
\[
\forall \epsilon \in E_t((\text{Begin}_t \rightarrow \epsilon) \land ((\epsilon \rightarrow \text{Commit}_t) \lor (\epsilon \rightarrow \text{Abort}_t)))
\]
• **Begin** can start a transaction only once; i.e
  \[(\text{Begin}_t \in H) \Rightarrow \neg(\text{Begin}_t \rightarrow \text{Begin}_t)\]

• only an initiated transaction can commit or abort,
  \[(\text{Commit}_t \in H) \Rightarrow (\text{Begin}_t \rightarrow \text{Commit}_t)\]
  \[(\text{Abort}_t \in H) \Rightarrow (\text{Begin}_t \rightarrow \text{ Abort}_t)\]

• a transaction cannot commit or abort more than once, it cannot be committed after it has been aborted, and vice versa,
  \[(\text{Commit}_t \in H) \Rightarrow ((\text{Abort}_t \not\in H) \land \neg((\text{Commit}_t \rightarrow \text{Commit}_t)))\]
  \[(\text{Abort}_t \in H) \Rightarrow ((\text{Commit}_t \not\in H) \land \neg(\text{Abort}_t \rightarrow \text{Abort}_t))\]

5. \(\text{View}_t = H_t\)

6. \(\text{Conflict-set}_t = \{p'_i[ob]\mid t' \neq t, \text{In-progress}(p'_i[ob])\}\)

7. \(\forall \text{ob} \exists p(p_i[ob] \in H) \Rightarrow (\text{ob is atomic})\)

8. \((\text{Commit}_t \in H) \Rightarrow (t \text{ is serializable})\)

9. \(\exists \text{ob} \exists p(\text{Commit}_t[p_i[ob]] \in H) \Rightarrow (\text{Commit}_t \in H)\)

10. \((\text{Commit}_t \in H) \Rightarrow \forall \text{ob} \forall p((p_i[ob] \in H) \Rightarrow (\text{Commit}_t[p_i[ob]] \in H))\)

11. \(\exists \text{ob} \exists p(\text{Abort}_t[p_i[ob]] \in H) \Rightarrow (\text{Abort}_t \in H)\)

12. \((\text{Abort}_t \in H) \Rightarrow \forall \text{ob} \forall p((p_i[ob] \in H) \Rightarrow (\text{Abort}_t[p_i[ob]] \in H))\)

Briefly, axioms 1-3 define the significant events, initiation event, and termination events of atomic transactions. Axiom 4 states that such transactions should satisfy the fundamental axioms of transactions. Axiom 5 relates the view of an atomic transaction to a current history, i.e excludes the state of objects accessed by other in-progress transactions. Similarly axiom 6 restricts the conflict set of an atomic transaction to all operations of other concurrent transactions. Notice that all such operations are not necessarily in conflict with transaction \(t\), but the conflict should be considered and determined. Axioms 7 and 8 state the failure atomicity and serializability properties of atomic transactions. Axioms 9-12 state the fact that an atomic transaction cannot commit or abort unless all its operations have done so and vice-versa.
6.2 Schedulers

The concept of transaction in TractorS is the same as in ACTA. In this section we define the notion of nested transactions and relate the scheduler concept in TractorS to it. One should notice that the events Commit and Abort are general terms in TractorS and will be redefined later on, whereby the definition will include open-nested transactions.

**Definition 15** A nested transaction $T$ of subtransactions $t_1, \ldots, t_n$, $n > 1$ is a transaction such that:

1. \[ \{ \exists i, j | a_i, a_j \text{ atomic transaction } \land E_{a_i} \subseteq E_T \land E_{a_j} \subseteq E_T \} \]
2. \[ OE_T = \bigcup_{i=1}^{n} OE_{a_i} \]
3. \[ SE_T \supset \{ \bigcup_{i=1}^{n} SE_{t_i} \} \cup \{ \text{Begin}_T, \text{Commit}_T, \text{Abort}_T \} \]
4. \[ \forall i (\text{Begin}_{t_i} \in E_T \Rightarrow \text{Begin}_T \rightarrow \text{Begin}_{t_i}) \]

and

\[ \epsilon \in E_T \Rightarrow \forall \epsilon' \in \bigcup_{i=1}^{n} E_{t_i}, \epsilon' \rightarrow \epsilon \]

where $\epsilon \in \{ \text{Commit}_T, \text{Abort}_T \}$

5. \[ \forall \epsilon \in E_T (\epsilon \rightarrow \text{Commit}_T \lor \epsilon \rightarrow \text{Abort}_T) \]

The first and second items states that atomic transaction are building blocks of nested transactions in TractorS and they are the only ones invoking events on objects. The third item states that a nested transaction has its own significant events set which is a superset of the significant events of subtransactions. Some elements of the set are defined but there are others to be defined later. The forth item states that all subtransactions should begin after the begin of the nested transaction and terminate before it terminates. The fifth item states that a nested transaction cannot be aborted after its commitment and vice-versa. This
is important in TractorS because subtransactions of a nested transaction may abort later on, before global commitment stage, but the top-level transaction is responsible for that.

**Definition 16** A serial scheduler $S$ of subtransactions $t_1, ..., t_n$ is a nested transaction with the following ordering of events:

1. **specific scheduler events:**
   \[ \exists i (Abort_{t_i} \in E_S) \Rightarrow Abort_S \in E_S \]
   and
   \[ Commit_{t_n} \in E_S \Leftrightarrow Commit_S \in E_S \]

2. **subtransaction events:**
   \[ \forall i, 1 < i \leq n, (Begin_{t_i} \in E_S \Rightarrow Commit_{t_{i-1}} \rightarrow Begin_{t_i}) \]
   and
   \[ \exists i (Abort_{t_i} \in E_S) \Rightarrow \forall j (Begin_{t_j} \in E_S \Rightarrow Abort_{t_j} \in E_S) \]

The first item states that the scheduler aborts if and only if any subtransaction aborts, and commits if and only if the last one commits. Besides the necessity condition for commitment of the scheduler, the \textit{if and only if} condition emphasizes that it may not commit or abort by any other means. The second item describes the execution order of the subtransactions, i.e. one cannot begin executing until the previous one commits, and the fact that if one is aborted all started ones get aborted as well.

Before defining the next type of scheduler we introduce the **maximal members** of a partial order set. An element $y \in P$ is called a maximal member of $P$ relative to partial ordering $\leq$ if for no $x \in P$ is $y < x \ [95]$. Such a member is not necessarily unique. Let $\text{max}(P)$ denote the set of maximal members of $P$.

**Definition 17** A parallel scheduler $S$ of subtransactions $t_1, ..., t_n$ is a nested transaction with the following ordering of events:

1. **specific scheduler events:**
   \[ \exists i (Abort_{t_i} \in E_S) \Rightarrow Abort_S \in E_S \]
and
\[ \exists j (\text{Commit}_{t_j} \in \max(\bigcup_{i=1}^{n} E_{t_i})) \iff \text{Commit}_{S} \in \max(E_{S}) \]

2. subtransaction events:
\[ \exists i (\text{Abort}_{t_i} \in E_{S}) \Rightarrow \forall j (\text{Begin}_{t_j} \in E_{S} \Rightarrow \text{Abort}_{t_j} \in E_{S}) \]

The first item states that if any subtransaction is aborted the scheduler gets aborted and that if the maximal member of the set of events invoked by subtransactions is a commit, i.e. if the last invoked event by subtransactions is a commit, then the scheduler commits (its maximal member is \( \text{Commit}_{S} \)). It does not enforce any order for start or end of subtransactions. The second item states that if a subtransaction aborts then all started subtransactions get aborted.

**Definition 18** A serial-alternative scheduler \( S \) of subtransactions \( t_1, \ldots, t_n \) is a nested transaction with the following ordering of events:

1. specific scheduler events:

\[ \text{Abort}_{t_n} \in E_{S} \iff \text{Abort}_{S} \in E_{S} \]

and
\[ \exists i (\text{Commit}_{t_i} \in E_{S}) \iff \text{Commit}_{S} \in E_{S} \]

2. subtransaction events:
\[ \forall i, 1 < i \leq n (\text{Begin}_{t_i} \in E_{S} \Rightarrow \text{Abort}_{t_{i-1}} \rightarrow \text{Begin}_{t_i}) \]

and
\[ \exists i, j ((\text{Commit}_{t_i} \in E_{S} \land \text{Commit}_{t_j} \in E_{S}) \Rightarrow i = j) \]

The first item states that if any subtransaction commits the scheduler commits but unless the last one is failed, it will not fail. The second item states the serial order of preference of subtransactions, i.e one is not tried unless the previous one fails. It also states the “one is enough” semantics, i.e, only one subtransaction may succeed.

**Definition 19** A parallel-alternative scheduler \( S \) of subtransactions \( t_1, \ldots, t_n \) is a nested transaction with the following ordering of events:
Chapter 6. Formal Semantics

1. specific scheduler events:

\[ \forall i (\text{Abort}_{t_i} \in E_S) \iff \text{Aborts} \in E_S \]

and

\[ \exists i (\text{Commit}_{t_i} \in E_S) \iff \text{Commits} \in E_S \]

2. subtransaction events:

\[ \forall i, j, i \neq j (E_i \cap E_j = \emptyset) \]

and

\[ \exists i (\text{Commit}_{t_i} \in E_S) \Rightarrow \forall j \neq i (\text{Begin}_{t_j} \in E_S \Rightarrow \text{Abort}_{t_j} \in E_S) \]

and

\[ \exists i, j (\text{Commit}_{t_i} \in E_S \land \text{Commit}_{t_j} \in E_S) \Rightarrow i = j \]

The first item states that if one subtransaction succeeds the scheduler commits and it only aborts if they all fail. The second item states that the subtransactions are independent, i.e. the invoke different events. It also states that if one subtransaction succeeds all others get aborted and that only one subtransaction may succeed.

The term scheduler represents any of the above types of schedulers. The schedulers in TractorS have some attractive properties. One such property is that although vital and non-vital subtransactions are supported, formalizing the latter explicitly is not required and extra constructs are not needed since a proper combination of schedulers can express this situation. This is explained below.

**Definition 20** A subtransaction of scheduler S is vital if its abortion forces the scheduler to abort, i.e.,

\[ t_i \text{ is vital } \equiv (\text{Abort}_{t_i} \in E_S \Rightarrow \text{Abort}_S \in E_S) \]

All subtransactions of a serial scheduler are vital because one cannot start executing until the previous one commits. On the other hand subtransactions of
serial-alternative and parallel-alternative schedulers have the "one is enough" semantics which is close to the non-vital property. The parallel schedulers, however, may have vital and non-vital subtransactions.

Before relating vital subtransactions to schedulers, we define a null transaction:

**Definition 21** A transaction $e$ is called a null transaction if it commits immediately after it has begun without accessing any data items, i.e:

$$E_e = \{ \text{Begin}_e, \text{Commit}_e \}$$

and

$$\text{Begin}_e \in E_e \Rightarrow \text{Commit}_e \in E_e$$

A non-vital subtransaction can be simulated by a serial-alternative scheduler whose second subtransaction is the null transaction. If the first one being the actual subtransaction with the non-vital semantics does not commit then the second one commits and the parent always receives a success result. By using this approach the non-vital property is satisfied without needing extra constructs.

The four types of schedulers are general enough to support all types of temporal dependencies which appear in the literature [35]. For example the so called commit-serial dependency which requires a transaction not to commit before another concurrent one, is easily stated by defining a commit dependency between the two (see the following section). It is not directly supported as a scheduler type due to its rareness. Schedulers are basic elements of transaction scheduling in our model. There may be any combination of them in a transaction-tree.

We conclude this section by formalizing the top-level transaction and transaction tree concepts. The notion of responsibility is the same as in ACTA as already defined.

**Definition 22** A top-level transaction $\text{top}$ is a transaction which is responsible for all atomic transactions it activates; i.e,

$$\text{top is top-level} \equiv \forall t_i(E_{t_i} \subseteq E_{\text{top}} \Rightarrow \text{Responsible}_{\text{top}}(t_i))$$

Notice the following:


- schedulers do not access database items, so there is no need to define their responsible transaction.

- a top-level transaction is not responsible for delegated subtransactions.

**Definition 23** A transaction tree is a tree in which atomic transactions are leaves and all non-leaf nodes are schedulers as defined later on.

Notice the following:

- the leaves of the transaction tree are not restricted to atomic transactions. Delegators as defined later also appear as leaves in a transaction tree at a site.

- the definition includes multi-level transaction tree which is a completely decomposed one with no delegators.

Since TractorS addresses distributed environments, we introduce some relevant notations and assumptions at this point for subsequent use.

**Notation 1** Let parent\(_i\) denote the parent of subtransaction \(t\), ancestor\(_i\) denote the set of ancestors of \(t\) in the transaction tree, path\(_i\) denote the set of sites that ancestors of \(t\) have gone through, site\(_i\) denote the site handling it, and foreign\(_i\) denote foreign acquaintances of the site\(_i\).

**Assumption 1** The number of sites in the network is a finite set [95].

### 6.3 Scheduler Dependencies

In this section we summarize the dependencies in schedulers. They are of two types:

(i) dependencies derived from the definitions of schedulers, and (ii) dependencies which are application dependent, i.e. are enforced if declared in the transaction tree.
Chapter 6. Formal Semantics

Dependencies Derived from Definitions

1. in a serial scheduler:
   (a) \textit{Begin-on-Commit Dependency} of a subtransaction over previous one \((t_i, BCD t_{i-1})\): A subtransaction cannot begin executing until the previous one commits; i.e,
   \[
   \forall i, 1 < i \leq n (\text{Begin}_{t_i} \in E_S \Rightarrow \text{Commit}_{t_{i-1}} \rightarrow \text{Begin}_{t_i})
   \]
   (b) \textit{Abort Dependency} of the scheduler on a subtransaction \((S AD t_i)\): if a subtransaction aborts then the scheduler aborts; i.e,
   \[
   \exists i (\text{Abort}_{t_i} \in E_S) \iff \text{Abort}_S \in E_S
   \]

2. in a parallel scheduler:
   Let \(t_i\) denote the last committed subtransaction, i.e \(\text{Commit}_{t_i} \in \max(\bigcup_{i=1}^{n} E_{t_i})\):
   (a) \textit{Abort Dependency} of the scheduler on a subtransaction: if a subtransaction aborts then the scheduler aborts; i.e,
   \[
   \exists i (\text{Abort}_{t_i} \in E_S) \iff \text{Abort}_S \in E_S
   \]
   This dependency implies that all subtransactions are vital. Non-vital subtransactions are supported as explained before.

3. in a serial-alternative scheduler:
   (a) \textit{Begin-on-Abort Dependency} of a subtransaction on the previous one \((t_i, BAD t_{i-1})\): a subtransaction cannot begin executing until the previous one aborts; i.e,
   \[
   \forall i, 1 < i \leq n (\text{Begin}_{t_i} \in E_S \Rightarrow \text{Abort}_{t_{i-1}} \rightarrow \text{Begin}_{t_i})
   \]
   (b) \textit{Abort Dependency} of the scheduler on the last subtransaction: if the last subtransaction aborts then the scheduler aborts; i.e,
   \[
   \text{Abort}_{t_n} \in E_S \iff \text{Abort}_S \in E_S
   \]
(c) **Exclusion Dependency** of a subtransaction on another \((t_i \mathcal{E}D t_j)\): if one subtransaction commits others must abort; i.e.,

\[
\exists i(\text{Commit}_{t_i} \in E_S) \Rightarrow \forall j \neq i(\text{Begin}_{t_j} \in E_S \Rightarrow \text{Abort}_{t_j} \in E_S)
\]

4. **Disjoint Dependency** of any two subtransactions \((t_i \mathcal{DD} t_j)\): any two subtransactions invoke disjoint events; i.e.,

\[
\forall i \neq j(E_i \cap E_j = \emptyset)
\]

5. in a parallel-alternative scheduler:

(a) **Exclusion Dependency** of a subtransaction on another \((t_i \mathcal{E}D t_j)\): if one subtransaction commits others must abort; i.e,

\[
\exists i(\text{Commit}_{t_i} \in E_S) \Rightarrow \forall j \neq i(\text{Begin}_{t_j} \in E_S \Rightarrow \text{Abort}_{t_j} \in E_S)
\]

(b) **Abort Dependency** of the scheduler on the maximal member of the events invoked by subtransactions: if the latest subtransaction aborts then the scheduler aborts; i.e,

\[
\exists (\text{Abort}_{t_i} \in \text{max}\bigcup_{t=1}^{n} E_i) \Leftrightarrow \text{Abort}_S \in E_S
\]

6. in general:

(a) **Begin Dependency** of a subtransaction on the scheduler \((t_i \mathcal{BD} S)\): a subtransaction cannot begin executing until its parent has begun; i.e,

\[
\text{Begin}_{t_i} \in E_S \Rightarrow \text{Begin}_S \rightarrow \text{Begin}_{t_i}
\]

(b) **Strong-Commit Dependency** of a subtransaction on its responsible transaction \((t_i \mathcal{SCD} \text{ Responsible}_{t_i})\): if the responsible transaction commits then any of its atomic subtransactions which have already succeeded must commit; i.e,

\[
\forall i((\text{Commit}_{\text{Responsible}_{t_i}} \in E_{\text{Responsible}_{t_i}} \land (\text{Commit}_{t_i} \rightarrow \text{Commit}_{\text{parent}_{t_i}})) \Rightarrow \text{Commit}_{t_i} \in E_{\text{Responsible}_{t_i}}))
\]

The responsible transaction concept is generalized later (delegated transactions are responsible for their subtransactions). Also **Commit** and **Abort** operations are of two types as defined in definition 35.
Chapter 6. Formal Semantics

2. **Strong-Abort Dependency** of a subtransaction on its responsible transaction \((t_i \text{ SAD Responsible}_{t_i})\): if the responsible transaction aborts then any subtransaction aborts; i.e., Let \(H\) be a history;

\[
\forall i (\text{Abort}_{\text{Responsible}_{t_i}} \in E_{\text{Responsible}_{t_i}} \Rightarrow \text{Abort}_{t_i} \in E_{\text{Responsible}_{t_i}})
\]

Notice that despite the Commit, Abort of the transaction may not be part of the events of its top-level transaction (see next section for the second type of Abort).

(d) **Abort-on-Cycle Dependency** of subtransactions at a site \((t_i \text{ ACD } t_j)\):

if the whole or part of a scheduler is re-submitted to the site issuing it by any means, the scheduler is aborted; i.e.,

\[
(E_j \subseteq E_i) \land E_c \Rightarrow \text{Abort}_{t_i} \in E_i
\]

where

\[
E_c \equiv (\text{site}_i = \text{site}_j \land \{ \exists s, E_j \subset E_s \land \text{site}_s \neq \text{site}_i \})
\]

Notice that equal subset condition states that \(i\) and \(j\) could be the same transaction.

Only the top-level transaction has strong commit and abort dependencies over the subtransactions. This is because other schedulers have intermediate roles, i.e., their task is to lead their own children and come up with a successful result if possible. The final synchronization and decision is left for the responsible transaction. Only the top-level transaction and atomic subtransactions participate in the final commitment protocol.

**User Defined Dependencies**

While all the dependencies stated above are enforced automatically, the following two types of dependencies which apply only to parallel schedulers are application dependent, i.e., may be declared in the transaction tree if necessary.

1. **Commit Dependency** of a subtransaction on another \((t_j \text{ CD } t_i)\): if both transactions commit then commitment of \(t_j\) precedes commitment of \(t_i\); i.e.,

\[
(\text{Commit}_{t_i} \in E_S \land \text{Commit}_{t_j} \in E_S) \Rightarrow \text{Commit}_{t_i} \rightarrow \text{Commit}_{t_j}
\]
2. **Value Dependency** of a subtransaction on another \((t_j \text{VD} t_i)\): \(t_j\) is the consumer of a value (to be) produced by \(t_i\).

Commit dependency and value dependency impose abort dependency in the opposite direction, i.e

\[(t_j \text{CD} t_i \lor t_j \text{VD} t_i) \Rightarrow (\text{Abort}_{t_i} \in E_S \Rightarrow \text{Abort}_{t_j} \in E_S)\]

3. **Argument Dependency** of a subtransaction on its sibling \((t_i \text{ARD} t_j)\): a subtransaction cannot start executing unless one or more of its arguments is provided by one of its sibling.

The only scheduler whose subtransactions may produce arguments for their siblings is the serial scheduler because in other types either the subtransactions run concurrently (one doesn’t necessarily start after the other to get its arguments from it) or there is at most one subtransaction at a time (choice schedulers). Due to the semantics of the serial scheduler, no extra abort dependency should be defined.

### 6.4 Transaction Completion and Failure

We distinguish between two different kinds of transaction completion in TractorS which leads to two different types of transaction failure. Recall that TractorS is a generalization of the nested transactions model [1]. The types of action completion and failure which are the same as in the nested transaction model are defined here. Others are defined in the following subsection. Unlike the nested transaction model we use distinguished names for each type to emphasize the difference and reason about them more clearly, but still the terms *Commit* and *Abort* stand for any type of action completion and failure respectively in TractorS.

The term *Safe-commit* which is the counterpart of *Commit* in the nested transaction model, where the top-level transaction also commits, is defined as follows:

**Definition 24** Let \(H\) be a history;

1. Commit of a top-level transaction is called *Safe-commit*; i.e,
Chapter 6. Formal Semantics

\[ \text{Commit}_{\text{top}} \in H \equiv \text{Safe-commit}_{\text{top}} \in H \]

2. an atomic transaction Safe-commits if its last event has been a Commit and its top-level transaction Safe-commits; i.e.,

\[ \text{Safe-commit}_{ti} \in H \iff \text{Commit}_{ti} \in \max(E_{ti}) \land \text{Safe-commit}_{\text{top}(ti)} \in H \]

It is important to notice that in TractorS a top-level transaction may commit while some of its subtransactions are aborted (e.g. in alternative schedulers).

Also the term \textit{Undo} which is the counterpart of \textit{Abort} in the nested transaction model is defined as:

\textbf{Definition 25} \textit{Undo} of transaction occurs whenever all the object accesses in its access-set are aborted; i.e,

Let \( H \) be a history.

\[ \text{Undos} \in H \iff \{ \forall p \in \text{Access-sets}(\text{Abort}_p \in H) \} \]

\textbf{6.5 Correctness of Schedulers}

In this subsection we show that schedulers produce correct results, i.e. all the objects in their access-set behave atomically (correctly and serializeably as defined in ACTA) whenever committed or aborted.

\textbf{Definition 26} A transaction produces \textbf{correct results} if whenever terminated, all the objects in its access-set are atomic.

\textbf{Lemma 1} If all subtransactions of a scheduler produce correct results, the scheduler produces correct results whenever it is committed or undone.

\textbf{Proof:}

Case 1: the scheduler is committed:

1. correct behavior of objects:
Chapter 6. Formal Semantics

• if $S$ is a serial or parallel scheduler:
  All the operations on all the objects are committed, so all objects behave correctly.

• if $S$ is a serial-alternative scheduler:
  A subtransaction cannot begin executing unless the previous one aborts due to the $BAD$ dependency, so any two operations in two different subtransactions are return-value-independent (serial execution is a stronger condition than return-value-dependency). Only one subtransaction may commit due to the $ED$ dependency. Hence all objects behave correctly since all subtransactions are assumed to produce correct results.

• if $S$ is a parallel-alternative scheduler:
  Any two subtransactions invoke disjoint set of objects due to the $DD$ dependency, so any two operations in two different subtransactions are return-value-independent. Only one subtransaction may commit due to $ED$ dependency. Hence all objects behave correctly since all subtransactions are assumed to produce correct results.

2. serializable behavior of objects:
  All objects behave serializeably due to assumption 4.

Case 2: the scheduler is undone:
  The proof is trivial since all the operations on all objects are aborted independent of the type of the scheduler.

\[ \Box \]

**Theorem 1** Any scheduler produces correct results whenever it is committed or undone.

**Proof:**
  We prove the theorem by induction on the levels of the transaction tree. Assume that it is of depth $n+1$, $n \geq 1$. Notice that if the tree is of depth 1 the result is correct since it is a single atomic transaction.

  1. Only atomic transactions appear on level $n+1$ by definition of transaction tree, so all schedulers on level $n$ produce correct results due to lemma 1 since their subtransactions (atomic transactions) produce correct results.
2. Assume that all schedulers on level k produce correct results.

3. Prove that all schedulers on level k-1 produce correct results.

It is so due to lemma 1 since the subtransactions of any scheduler on level k-1 are either the schedulers on level k which are assumed to produce correct results, or atomic transactions.

\[ \diamond \]

### 6.6 Compensation

The notion of compensation in TractorS is based on top of certain predefined atomic transactions at component databases for each of which another predefined compensating atomic transaction exists to remove its effects.

**Definition 27** A transaction with atomic transaction \( a_1, ..., a_n \) in its transaction tree is **compensatable** if:

1. each atomic transaction is return-value-independent with respect to any other transaction \( S \); i.e

   \[
   \forall a_i, \forall S \notin \bigcup_{i=1}^n \{a_i\}(\text{Abort}_{a_i} \neq \text{Abort}_S)
   \]

2. effects of each \( a_i \) could be removed, regardless of the states of the databases, by execution of a unique pre-defined atomic transaction \( \text{ca}_i \); i.e

   \[
   \forall a_i(\exists c(a_i)((\text{Commit}_{a_i} \rightarrow \text{Commit}_{c(a_i)}) \equiv \text{Abort}_{a_i} \in H))
   \]

The compensating transaction is **dynamically formed** at run time. It is made of \( \text{ca}_i \)s for committed \( a_i \)s in the right order which is the reverse order of start or commitment of certain component transactions as defined below.

**Definition 28** A compensatable transaction \( t \) with atomic transaction \( a_1, ..., a_n \) has a **compensating counterpart** \( \text{comp}_t \) which is a top-level transaction so that:

1. \( \text{Commit}_{a_i} \in E_t \Rightarrow \text{Begin}_{ca_i} \in E_{\text{comp}_t} \)

2. \( \exists a_i, a_j | a_j \text{ BCD } a_i \Rightarrow \text{ca}_i \text{ BCD } \text{ca}_j \)
The first item states that only committed atomic transactions need compensation. The second one states that only $BCD$ needs be considered in ordering the compensating transactions. Effectively, this means that $comp_i$ is made of parallel and serial schedulers only; all subtransactions go in parallel except for the above constraint which is proved to be the necessary and sufficient ordering dependency below. For all compensatable atomic transactions in the context of a top-level transaction, a single compensating transaction is formed which is a top-level transaction due to the fact that the corresponding top-level transaction is already terminated.

**Theorem 2** $BCD$ is the necessary and sufficient dependency for ordering subtransactions of a compensating transaction.

**Proof:**

1. necessary condition:

   Only subtransactions of serial schedulers have $BCD$ dependency. One may alter data items used by later ones and may produce arguments for them, so these values should be restored in the reverse order.

2. sufficient condition.

   We prove that no other dependency remain to be considered. Let's observe each type separately:

   - dependencies derived from definitions:

     There is no order for execution of subtransactions of parallel schedulers. On the other hand the serial-alternative and parallel-alternative schedulers have at most one committed subtransaction. Hence only two types of dependencies should be observed: dependencies in serial schedulers and the general ones for schedulers. These dependencies are $BCD$, $AD$, $BD$, $SCD$, $SAD$, and $ACD$. Only $BCD$ from this set is about ordering of committed subtransactions, so only this one should be considered.

   - user defined dependencies:

     They are $CD$, $VD$, and $ARD$. The first one only orders the commit of subtransactions. There is no order in accessing data items by them.
The second one does not enforce any compensating orders due to the assumption that the consumer is only allowed to consume the value, and not alter relevant data items\(^2\). Therefore only ARD remains to be considered. However, ARD is contained in BCD because the subtransactions will be ordered any way\(^3\).

One should notice that the dependencies defined later are either in the context of compensation or related to concurrent top-level transactions, and therefore are irrelevant to forming a compensating transaction. Dependencies considered in the above theorem are not necessary unless the subtransactions access the same data items. This is however not known to the global transaction manager, so we consider the general case.

Based on compensation, TractorS supports a second type of commitment:

**Definition 29** Let \(H\) be a history and \(t\) be a compensatable transaction. Unsafe-commit of \(t\) is defined as:

\[
\text{Unsafe-commit}_{t} \in H \iff \text{Commit}_{t} \rightarrow \text{Safe-commit}_{\text{Responsible}, t} \land (\neg \text{in-progress}_{t})
\]

Only the related top-level transaction has the knowledge and is concerned about further possible (logical) undo actions related to such a subtransaction. To others it has committed as any transaction does.

We use the term Cancel for removing the effects of an already unsafe-committed transaction:

**Definition 30** Let \(H\) be a history. Cancel of transaction \(t\) is defined as:

\[^2\]a compensatable transaction with value dependency may alter other data items based on that value. This has nothing to do with order of subtransactions, but the compensating counterpart may also need the value to restore the data items. The value should be provided by the compensating transaction of the original producer. Hence, depending on the application, compensation may involve value dependency as well.

\[^3\]When coding the compensating transactions, it is important to consider the application dependent argument dependencies. This is however irrelevant to the order of subtransactions.
Cancel_t ∈ H ⇔ (Unsafe-commit_t → Abort_responsible_t) ∧ Begin_compt → Safe-commit_compt

Recall that the precedence operation implies that both sides are in the history, in the above formula besides defining the precedence relation of the events their existence in the history is implicitly declared. Such a process involves forming, executing and committing the (top-level) compensating transaction. The following dependencies are enforced by definition:

- execution of a compensating transaction is subject to unsafe-commitment of the corresponding compensatable transaction and later failure of its responsible transaction; i.e

  Unsafe-commit_t → Abort_responsible_t ⇔ Begin_compt ∈ H

This is stated by conjunction of the following two dependencies:

1. Strong-Begin-on-Unsafe-commit Dependency (SBUD) of a compensating transaction on its compensatable counterpart.

2. Strong-Begin-on-Abort Dependency (SBAD) of a compensating transaction on its compensatable counterpart’s responsible transaction.

- Strong-Commit-on-Begin Dependency (SCBD) of a compensating transaction on itself, a compensating transaction must eventually commit once it has begun; i.e,

  Begin_compt ∈ H ⇒ Safe-commit_compt ∈ H

Recall that as in ACTA, the terms Commit and Abort stand for any type of action completion and failure respectively. Commit denotes Safe-commit or Unsafe-commit, and Abort means Undo or Cancel.

One top-level cancel transaction is formed and executed for each failed top-level transaction involving unsafe-commit. Following the above theorem, all compensating transactions form a parallel scheduler except subtransactions of each serial scheduler which define a serial scheduler (recursively) with the same subtransactions in the reverse order. We apply the following reasonable convention for arguments (parameters) of the compensating subtransaction:
Assumption 2. A compensatable transaction has the same number of arguments in the same order as its compensating counterpart.

Correctness of Compensated Transactions

Four operations may possibly end a transaction in TractorS, namely Safe-commit, Unsafe-commit, Undo, and Cancel. We have already proved the correctness of Safe-commit, and Undo. Here we prove the correctness of the other two:

Theorem 3. A transaction t produces correct results whenever it is unsafe-committed or canceled.

Proof:

1. the transaction is unsafe-committed:

From the definition, a transaction may unsafe-commit if it is compensatable. By definition such a transaction is return-value-independent with respect to any other transaction, i.e. other transactions which have used its effects do not need to be aborted. Therefore it produces correct results.

2. the transaction is canceled: This process involves initiating and safe-committing (both enforced) comp. Since this transaction is top-level it produced correct results due to theorem 1.

6.7 Delegation

Another significant event in TractorS is Delegate. Delegation has a quite different meaning here than in ACTA. In TractorS a transaction is delegated to a Global Transaction Manager (GTM) rather than to another ongoing transaction. This happens if the delegated transaction falls in the domain of expertise of the target site which is different from that of the delegator so that it is not able to decompose the transaction. The delegated transaction becomes responsible for itself, having the advantage of localizing decisions and responsibilities.
to sites with relevant expertise. The target site may decompose, decide about compensability, further delegate the whole or parts of a delegated transaction to other sites, etc, and guarantees correctness of its result. The delegator cannot terminate unless the delegated transaction terminates.

If a delegated transaction is further decomposed at the target site, the process is called **cascading delegation** as opposed to **simple delegation**. The distinction is particularly important due to their different linguistic necessities (system driven versus user oriented). Current scripting language for TractorS does not support cascading delegation, but the idea is developed completely. If a target site decides to further delegate the whole transaction to more expert sites, direct communication is established between the original delegator and the final site without making the intermediate sites involved any more. This happens in simple delegation because the transaction is not decomposed (it is further delegated as a whole). These concepts are formalized below:

**Definition 31** Let $H$ be a history. Let $D$ be a transaction and $j$ be a leaf sub-transaction of $D$. Delegate$_D[s,j]$ means that:

1. $D$ is the delegator;
2. $s$ is a foreign acquaintance not already in the path of $D$; i.e.
   \[ s \in \text{foreign}_D \land s \not\in \text{path}_D \]
3. $j$ is passed to $s$ and its responsibility is passed to itself; i.e.,
   \[ \text{site}_j = s \land \text{Responsible}_j(j) \]
4. if $j$ is delegated to another site then the intermediate action is eliminated; i.e.
   \[ \text{Delegate}_D[s,j] \rightarrow \text{Delegate}_j[s_1,j] \Rightarrow \text{Delegate}_D[s_1,j] \in H \]
5. if $D$ aborts then $j$ must abort and $D$ cannot commit unless $j$ terminates; i.e.,
   \[ \text{Abort}_D \in H \Rightarrow \text{Abort}_j \in H \]

and

\[ \text{Commit}_D \in H \Rightarrow \text{Commit}_j \in H \lor \text{Abort}_j \in H \]
A transaction may only delegate its own leaf subtransactions (or itself if it is a leaf). One implication is that if a subtransaction is decomposed then it cannot be delegated as a whole. Cycles are stopped by making sure that the new site was not already involved in executing ancestors of the same transaction ($s \notin path_D$). Mutual termination dependency between the delegator and the delegated transaction is enforced; the latter cannot commit safely and the former cannot terminate independently. Since choice schedulers may delegate their subtransactions, a delegator may commit while its delegated subtransactions abort. Also nothing stops the delegated transaction to commit unsafely; it may do so like any other subtransaction. Passing responsibility of a delegated transaction to itself means that it produces correct results independently. This is used in assumption 3 below. However, the reason that we do not let such a transaction safe-commit independently is due to the fact that it is part of a global problem whose partial results, although correct, are not useful alone. This may not be true in some application, but needs further investigation.

**Assumption 3** In $\text{Delegate}_D[s,j]$, $j$ produces correct results at $s$. If $j$ succeeds then $\text{Commit}_j$ is passed to $D$. Otherwise $\text{Abort}_j$ is passed to $D$.

Types of delegation and correctness of their results are formalized below.

**Definition 32** $\text{Delegate}_D[s,j]$ is **simple delegation** if no ancestors of $j$ has been delegated; i.e.

$\text{Delegate}_D[s,j]$ is simple $\Rightarrow$

$$\{ \exists D_1, s_1, j_1 | (\text{Delegate}_{D_1}[s_1,j_1] \in H \land j_1 \in \text{ancestor}_j) \}$$

It is **cascading delegation** otherwise.

The crucial thing in correctness of the results is correctness of the delegator which has passed responsibility of some parts to themselves (they produce correct results by assumption 3). It is proved for simple and cascading delegation separately below.
Lemma 2 Let \( \text{Delegate}_D[s, j] \) be a simple delegation. \( D \) produces correct results whenever committed or aborted.

Proof:
The proof is trivial due to theorem 1 and assumption 3.

Theorem 4 Let \( \text{Delegate}_D[s, j] \) be a cascading delegation. \( D \) produces correct results whenever committed or aborted.

Proof:
The number of invocation of \( \text{Delegate} \) is finite in any transaction due to assumption 1 and item 2 of definition 31 (no cyclic delegation). Let the transaction tree of \( D \) be of depth \( d+1 \), \( n_i \) denote the number of invocations of \( \text{Delegate} \) for level \( i \) of the transaction tree, and \( \text{Delegate}^k_{D_i}[s_i^k, j_i^k] \) denote the \( k^{th} \) invocation (from left to right) at level \( i \). We prove the theorem by induction:

1. each one of \( \text{Delegate}^k_{D_i}[s_i^k, j_i^k] \) produces correct results due to lemma 2 since they are all simple delegations ending into atomic transactions by definition of transaction tree.

2. Assume that all \( \text{Delegate}^k_{D_{l-1}}[s_{l-1}^k, j_{l-1}^k] \) at some level \( l \geq 2 \) produce correct results.

3. Prove that all \( \text{Delegate}^k_{D_{l-1}}[s_{l-1}^k, j_{l-1}^k] \) produce correct results.

Any of the delegators \( D_{l-1} \) is a schedule due to definitions of transaction tree and cascading delegation. Its subtransactions are of three types:

(a) atomic transactions which produce correct results by definition.

(b) other schedulers which produce correct results by theorem 1.

(c) delegated transactions which produce correct results by item 2 above.

Therefore each one produces a correct result. The overall result is correct because the set is finite.
6.8 Concurrent Transactions

We argued in chapter 3 that quasi-serializability is one of the most appropriate concurrency control mechanisms for TractorS. Here we briefly define quasi-serializability, introduce notations and express an appropriate assumption to ensure correctness of concurrent transactions in TractorS.

**Notation 2** Let $H_k$ denote a local history at site $k$ and $SR^{Hk}$ denote (conflict preserving) serializability on $H_k$, i.e;

$$SR^{Hk} = (\forall t_i, t_j \in H_k, t_i \text{ and } t_j \text{ are serializable}).$$

**Definition 33** Let $H$ denote a history of top-level transactions $T_1, ..., T_m$. Let $T_k$ comprise $T_{k1}, ..., T_{kn}$ and $H_{k1}, ..., H_{kn}$ denote the corresponding local histories. $H$ is quasi-serial if:

1. $\forall H_{ki}, 1 \leq i \leq n, SR^{H_{ki}}$

2. $\forall T_i, T_j \in H(Begin_{T_i} \rightarrow Begin_{T_j} \Rightarrow ((\forall p_a[ob], a \in (T_{i1}, ..., T_{in}) \forall q_b[ob], b \in (T_{j1}, ..., T_{jn}), \exists k_i(p_a[ob] \in H_{ki} \land q_b[ob] \in H_{ki}) \Rightarrow (p_a[ob] \rightarrow q_b[ob]))$.

Informally this means that a global history is quasi-serial if:

1. all local histories are (conflict) serializable; and

2. there exists a total order of top-level transactions such that for every two, if one precedes the other in one history then all its operations in all histories that they both appear precede the other’s operations.

**Definition 34** A history is quasi-serializable if it is (conflict) equivalent to a quasi-serial history.

We use the notation $QSR^H$ to denote the quasi-serializable relation on histories in TractorS.

The following dependency is enforced to satisfy quasi-serializability condition; *Abort Dependency* of siblings in a top-level transaction at a site: if a subtransaction belonging to a top-level transaction arrives at a site while another subtransaction belonging to the same top-level transaction has arrived before and is not
terminated, the new one gets aborted; i.e,
\[ E_{t_i} \cap E_{top} \cap E_{t_j} \cap E_{top} \cap site_{t_i} = site_{t_j} \land (Begin_{t_i} \rightarrow Begin_{t_j} \land in-progress_{t_i}) \Rightarrow Abort_{t_j} \in E_{top} \]

Notice that the concurrency control mechanism of component databases may not suspend such subtransactions and cannot be changed due to the local autonomy property, so we abort the younger subtransaction if such an error occurs.

Correctness of concurrent transactions is expressed as an assumption below:

**Assumption 4** Quasi-serializability is enforced for all histories in TractorS.

This assumption emphasizes that the implementation mechanism of TractorS will enforce quasi-serializability.

We conclude the chapter by redefining the significant event set of transactions in TractorS.

The general terms *Commit* and *Abort* are redefined as:

**Definition 35** The term **Commit** stands for Safe_commit or Unsafe_commit. Similarly the term **Abort** stands for Undo or Cancel, i.e

\[ Commit_{t_i} \in E_i \Rightarrow (Safe-commit_{t_i} \in E_i \lor Unsafe-commit_{t_i} \in E_i) \]

\[ Abort_{t_i} \in E_i \Rightarrow (Undo_{t_i} \in E_i \lor Cancel_{t_i} \in E_i) \]

The final significant event set is defined as:

**Definition 36** Let \( t \) be a transaction, \( s \) be a site, and \( j \) be a leaf subtransaction of \( t \):

\[ SE_t \supset \{Begin_t, Safe-commit_t, Unsafe-commit_t, Undo_t, Cancel_t, Delegate_t[s, j]\} \]

The set is open-ended for future evolutions. An examples of a possible events is **Negotiate**.
Chapter 7

Conclusion and Future Research

This thesis presents an advanced transaction model for cooperative information systems called TractorS, together with its underlying scripting language. The model adopts the attractive properties of the actor model of computation in transaction management paradigm. The novel notion of transaction tree in TractorS includes subtransactions as well as a rich collection of decision making, chronological ordering, and communication and synchronization constructs for them. It provides flexible mechanisms to easily combine semantically dependent activities into a single top-level transaction, and facilitates concurrency by supporting parallel tasks and flow of information between them. Advanced concepts such as blocking/ non-blocking synchronization, vital and non_vital subtransactions, contingency transactions, temporal and value dependencies, and delegation are supported in this way. Compensatable subtransactions are distinguished and committed unsafely, and locks are released in order to facilitate cooperative as well as long-duration transactions. Automatic cancel procedures are provided to logically undo the effects of such commits if the global transaction fails.

In contrast to other models, TractorS is not only an extension of the nested transaction concept, it also suggests the use of a highly structured scripting language to complement open and closed nested transaction management. Database programming can gain in performance and problem-orientation if they can express the semantic dependencies between transactions directly. Simple and flexible mechanisms are provided for advanced users to query the databases, script their transactions accordingly, and accept weak forms of semantic coherence that allows for more concurrency. The transaction model is grafted onto the concurrent
object-oriented programming language *Sather* developed at UC Berkeley which has a nice high-level syntax, supports advanced object-oriented concepts, and aims toward performance and reusability. We have augmented the language with distributed programming facilities and various types of message passing routines as well as advanced transactions management constructs.

The vision of future cooperative information systems is compelling. It involves large numbers of heterogeneous, intelligent agents distributed over large computer and communication networks. The agents can be humans, humans interacting with information systems, and information systems performing tasks autonomously. The problem of *existing systems* developed using ancient technology will however exist for ever. No matter how great the vision, it will be of little value if it cannot be worked into the current technology base. The transactional necessities of such systems therefore have two aspects: supporting advanced concepts and respecting local autonomy of existing subsystems. The goals are hard to achieve and a lot is left to be done.

In future we will conduct research in a number of areas. For example, the concept of delegation needs further development. The current version of the scripting language for TractorS does not support cascading delegation due to lack of automatic decomposition power. Further research is to be done on both: transaction decomposition and cascading delegation. Linguistic necessities of this type are different. It needs a system driven code rather than users to program decomposed transactions and let TractorS manage their execution. There are other open problems in delegation. For example in some applications it may be helpful to let certain delegated transactions commit safely in an independent manner in order to use their results in subsequent transactions. Currently only top-level transactions are allowed to commit safely, but in some applications safe-commitment of delegated transactions is appealing. An example is CASE tools in which partially independent tasks go in parallel and are put together at points in time to test a whole software system. Automating such a task needs to save the amount of work being done by components, although the whole system may need further development.

Another major task left for future is query language and user interface for TractorS. Instead of typing their script classes in, users may be provided with
a graphical interface to program their transactions. Query language in object-oriented information systems is still an open problem. Most of the research being done either are based on the query languages for relational database or have presented ad-hoc query languages which are mostly application dependent. A well-accepted query language in this area is still to come. We propose a graphical interface which inherits the information from the type system of an application and provides mechanisms for transaction programming. The core of such a system is not application dependent. The need is served by inputting the class hierarchy of a particular application into it.

Another area which is left for future is transactional necessities of negotiation. We have reported our points of view in [56] and [9]. A lot of work is however left to be done. A new type of meta-actor called negotiator is proposed which govern a negotiation process involving a number of sites to find a solution or refutation for a global problem. It ends into the other types of actors, i.e the task is decomposed into schedulers, delegators, and atomic transactions to do the job. Mechanisms are needed for negotiating with sites in order to find the most appropriate ones for parts of a transaction, and decomposing it accordingly.
Bibliography


Proceedings of the The First International Conference on Intelligent and Co-operative Information Systems (ICICIS), Erasmus University Rotterdam, The Netherlands, May 12-14 1993,


[40] B. Liskov, R. Scheifer "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 3, 1983.


