# SPECIFICATION OF TRANSACTION SYSTEMS PROTOCOLS

by

**Sylvanus Agbonifoh Ehikioya**

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science

Winnipeg, Manitoba, Canada, 1997

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

# SPECIFICATION OF TRANSACTION SYSTEMS PROTOCOLS

BY

SYLVANUS AGBONIFOH EHIKIOYA

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree

of

DOCTOR OF PHILOSOPHY

# Declaration

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

# Thesis use Form

The University of Manitoba requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

| Date | Name | Signature | Address |
|---|---|---|---|

# Dedication

To my mother, *Madam Oba Lydia Ehikioya*, for all her self denials throughout my early education **and** to all mothers who suffered for the good of their children.

# Abstract

A fundamental requirement in specifying transaction systems is the need for a clear, concise, unambiguous, and rigorous behavioural and functional description of the systems' crucial features like concurrency, nondeterminism, mutual exclusion, synchronization, and deadlock avoidance. To write a specification that exhibits these characteristics requires a formalism that has both expressive power and the functionality for specifying and reasoning about the structure and behaviour of transaction models. To ensure that the specifications are consistent and verifiably correct requires expressing the specifications using mathematical notations and then using the notations' underlying formalism to prove correctness properties. Such requirements can only be satisfied within a *formal* framework.

However, most of the present transaction systems (models) are not formally specified or at best use methodologies that are *ad hoc* or semiformal. Unfortunately, when insufficient formalism is used to specify transaction systems protocols they are open to different interpretations thereby violating the preservation of specification interpretations requirement. Therefore, there is need for a thorough modelling of the systems based on formal models that are easy to use, verify and validate.

In this thesis, a Timed CSP based formal framework for transaction management is given. This framework is more general and not biased towards specific types of transaction. It integrates temporal behaviour of individual transactions with the dependencies among transactions that can arise e.g., when accessing shareable data objects. Further, the framework uses an event-based model based on causality and time because the partial orders together can naturally model concurrent events between transaction. In addition, the causality and time information are useful in analysing transaction execution for determining correctness and recoverable histories.

In brief, this thesis provides a taxonomy of a transaction's specification characteristics against which any specification can be assessed; presents a suite of requirements for an adequate formalism in which various concurrent activities and interactions of transactions can be naturally expressed; provides record data type extensions to CSP; specifies transactions correctness criteria and concurrency control protocols; and presents an ab-

stract level specification of an application, the Electronic Shopping Mall, to illustrate the concepts introduced.

# Acknowledgement

I would like express my sincere thanks to the following people and institutions that contributed to the successful completion of this work.

To my supervisor, Dr. Ken Barker, for his constructive comments and guidance as the thesis evolved and for his considerable patience in reading my specifications and proofs in a paragon of notations and styles. His immeasurable academic guidance, many invaluable discussions with me and suggestions, and patience in listening to and constructively critiquing my ideas led to the accomplishment of my academic goal. My sincere thanks to him.

To my examiners, Prof. David Scuse and Dr. James F. Peters III, for making a number of suggestions for improving the presentation and the focus of this thesis. Your continued interest and encouragement kept me going at some critical points during my study.

To Dr. Kasi Periyasamy for being a source of inspiration, information, encouragement and friendship. Thanks for being there for me for all the assistance and emotional support you provided and for keeping me going at the end. He has provided many invaluable comments and useful suggestions, and pointed out errors in the specifications when I was going astray. I do appreciate it.

To Dr Randal Peters and Dr. Peter C.J. Graham (Professors in the Dept.) and Dr. Sheela Ramanna (Dept. of Business Computing, University of Winnipeg) for their helpful comments and assistance with research materials.

To the Government of Canada for financial support in the form of Canadian Commonwealth Scholarship throughout my study.

To the University of Benin, Benin City, Nigeria for granting me study leave for the period.

To Prof. S. O Fatunla (late), University of Benin, Benin City, Nigeria for his steadfast belief in my academic abilities and encouragement.

To Mr. E. E. Oghuman, (President), Bonaventure Limited, Lagos, Nigeria for his moral

and initial financial support during my preparations to come to Canada. My dreams would have vanished into thin air if not for his support.

To my family, particularly my children (Sylvia, Sylvanus Jr., and Augusta), for their continuing love, patience and understanding, and for bearing with me during those difficult times I was unable to satisfy their material needs.

Finally, to my friends Oghre Emmanuel and Chris Oriakhi. I express my love and gratitude for their encouragement during my years of study.

# Contents

# List of Figures

# Chapter 1

# INTRODUCTION

Users interact with the database through the execution of special application programs called transactions [AE92] that are composed of a sequence of database steps derived from a program whose combined execution is known to preserve the database in a consistent and correct state. Thus, from a user's viewpoint, a transaction is an independent task or activity performed by the system. Transactions have a beginning; perform user defined tasks; and terminate leaving the system in a consistent state. Therefore, a transaction that updates database objects must preserve their integrity constraints.

Integrity is maintained by allowing only *safe transactions* to update the database. A safe transaction is one that does not violate database integrity constraints [SS89]. Motro [Mot89] states that database integrity has two complementary components: *validity*, guarantees the exclusion of all false information from the database, and *completeness* guarantees all true information is included in the database. This is achieved if a transaction moves the database from one consistent state to another. Correctness is partially supported by the *all-or-nothing principle*, (often called *atomicity*). The all-or-nothing principle states that "each transaction should either execute in its entirety or have no effects at all" [Ber90]. Thus, transactions in database systems are defined in terms of consistency, recovery, and permanence. Consistency implies that a committed transaction produces a consistent database state[1] if the database state before its execution was

[1]Consistent states generally satisfy some constraints involving relationships between the values of different data objects in the database.

consistent; recovery refers to the ability, upon failure, to take the database to some state considered correct; and permanence is the ability of a transaction to record its effects in the database. Therefore, a transaction should have the properties of atomicity, consistency, isolation, and durability. These properties are often collectively called the ACID test [Des90, Gra81, OV91].

The desire to support concurrency means synchronization is necessary to guarantee data/operations dependencies are maintained, thereby ensuring the correctness of results. These properties are embodied in *transaction protocols.* A transaction's protocol is a body of rules that defines (or describes) the correct behaviour of the transaction's encompassing model. The transaction's protocols require that transactions perform desired tasks correctly. To achieve consistency, the interleaved execution of concurrent transactions must be properly synchronized so some form of serializability is guaranteed.

One way to minimize the processing required to maintain integrity during transaction processing is to prove, at compile time, that transactions cannot [if run in isolation and serially] disobey integrity constraints [Gra94, SS89]. To determine this, a precise and independent *description* of the transaction protocols is required. Such a description is called a *specification.* The specification is implementation independent and concentrates on properties rather than mechanisms. That is, it is only *what* is required of a system that must be described, not the detail of *how* to do it.

Various terms have been used synonymously to denote a specification (see [Ger83] and [Dav88]). This thesis defines a specification as:

**Definition 1** *Specification:* A specification is a precise, independent, implementation independent description of the properties (statement of requirements) of a system, against which the system can be verified. ■

The IEEE Software Engineering Technical Committee [ANSI84] stipulates the system's properties that are specifiable. These are: system's functionality, performance (such as accuracy and timing performance), constraints (such as restrictions on functions, database integrity, and operating environments), attributes (such as portability, security, and maintainability), and external interfaces with other software/hardware.

A specification states properties that must be guaranteed of a system to ensure

correct behaviour[2]. The specification is structured as a collection of requirements, each describing some property that the system must satisfy. In other words, a specification describes what a system should do rather than the mechanics of the system.

The New Mariam-Webster Dictionary defines a system as "an organized integrated whole made up of diverse but interrelated and interdependent parts" [TMW89]. Systems have internal structure and exhibit behaviour. A system's behaviour is the result of the behaviour of its parts (which may themselves be systems) and of the interrelationships among those parts [CHJ86]. Birell *et al.* [BGHL87] state that any [concurrent] system's behaviour is describable using a sequence of atomic actions' executions.

A system has three representation forms: functional, structural, and behavioural. The description of the three distinct views given below adopts Harel's [Har86] methodology.

1. The functional view shows the system as a set of entities performing relevant tasks. This includes a description of the task performed by each entity and the way the entity interacts with other entities and the environment. Ideally, the functional view should complement the behavioural view so each transaction in the behavioural view is traceable through the system.

2. The structural view shows the composition of the system — the components, the interfaces, and the flow (data and control) between the components through the interfaces. The structural view also shows the environment, the interfaces, and information flow between it and the system.

3. The behavioural view shows how the system will respond to specific inputs: what state it will adopt and what output it will produce; what boundary conditions exist for valid input and which states are considered correct and consistent. This includes a description of the environment that is producing the inputs and consuming the outputs. It also includes constraints on performance imposed by the environment and function of the system.

[2]Correct behaviour is the degree to which a systems satisfies its specified requirements and to which requirements meet their associated needs.

## 1.1 Statement of the Problem

Several questions remain open vis-a-vis transactions:

1. What mechanisms can achieve the desired transaction correctness properties?
2. Can heterogeneous transaction models be integrated?
3. What environmental circumstances can a transaction complete successfully?
4. When is a nested transaction most appropriate?
5. When a transaction is initiated, when does its execution begin and end?
6. How do we reason about and verify the behaviour of transactions?

These questions can best be answered with a formal framework that captures transaction interactions. The first step is to formalise the concurrency algorithms using a suitable specification language. Correctness conditions for the specifications can be formulated in terms of the properties of the system's behaviours[3].

Some recent work on formal specification of transaction systems has appeared in the literature (see [AJR95, Chr91, EB93, EBO95, Ken96]). Gray and Reuter [GR93] state that "no grand unified theory of transactions has yet been developed". Further, there is currently no satisfactory or unified formalism for specifying and reasoning about database transactions [EB93] and most specification methodologies currently applied are *ad hoc*.

Currently, transaction model descriptions and their protocols consist of an informal explanation supported by the use of pseudocode to characterize the operations within a system. Unfortunately, when insufficient formalism is used to specify transaction systems protocols they are open to different interpretations. Li and McMillin [LM93, LM94] state many published (distributed) deadlock detection/resolution algorithms are found to be incorrect [MM79, Obe82, SN85,CKST89] because they have used informal approaches

[3]It is natural to express correctness conditions as restrictions on the actions of the system. This approach is simple but shows explicitly the action that are under the control of different components.

to reason about and verify the correctness of these algorithms. Only rigorous proofs are sufficient to show the correctness of such algorithms [Kna87] because intuitive operational arguments are error prone and subject to misinterpretation. Since formal correctness proof is difficult [Sin89] only a few sophisticated formal methods are available for proving correctness of algorithms [LM93]. Moreover, writing correct software requirements is a difficult problem [LS85] for which there are few analytical tools available.

A formal method supports formal reasoning about the specifications of the transaction systems as well as provides the basis for verification of the resulting software product. Jarvis [Jar92] noted that informal approaches are notoriously unreliable so formal methods pay dividends. The use of formal methods permits the analysis of complex protocols since they eliminate ambiguity in specifications.

A formal model for specifying transaction protocols is described here. The formal specification utilizes CSP to capture static and dynamic properties of transaction protocols. The thesis demonstrates that the following properties of transaction systems can be captured using this technique:

- correctness
- concurrency
- safety properties
- liveness properties
- timing relationships between transactions (or subtransactions) as a control mechanism
- dependency relationships between transactions.

The CSP specification language presents and proves the correctness of a number of transaction processing algorithms, including locking and timestamping, correctness criteria such as CSR and VSR, and the hierarchical transaction models. Finally, a sample application, the Electronic Shopping Mall, is presented to illustrate the technique. The structural and behavioural capabilities of the system are formalized in terms of assertions and constraints that *must* be preserved by any implementation of the system.

## 1.2 Motivation

Transaction systems must transparently demonstrate integrity, consistency[4], safety and liveness, and dependability properties. These properties are vital to all transaction models though the degree of importance placed on each varies between models. The new complex transaction systems [AE92] must satisfy the dependability requirement. Laprie defines dependability as "that property of a computing system that allows reliance to be justifiably placed on the service it delivers" [Lap89]. Dependability is a property that has other measures such as safety, reliability[5] and availability. Availability is a measure of the frequency the system is available to the users when required by the users (i.e., of being operational — not failed — at a given instant in time). Correctness ensures system behaviour conforms to the specified requirements. Formal methods address correctness aspect which is the focus of this research.

There are four approaches to achieving system dependability. These are:

- Fault avoidance to prevent fault occurrence or introduction.
- Fault tolerance to provide a service complying with the specification in spite of faults.
- Fault removal to minimize the presence of faults.
- Fault forecasting to estimate the presence, the creation, and the consequences of faults.

Formal methods are classified as fault avoidance and removal techniques which can increase dependability by removing errors from the requirements specification and by verifying the specification is correct. Dependability is further enhanced when formal methods are used with other techniques.

Transaction systems specifications need a clear, concise, unambiguous, and rigorous behavioural and functional description of crucial features. Concurrency, nondeterminism,

[4]Sometimes consistency property is relaxed.

[5]Recovery is a subset of reliability.

mutual exclusion, synchronization, and avoidance of deadlock are key features of transaction systems so developing a formal specification and verification techniques for such systems components is very desirable. Unfortunately, none of the present transaction systems (models) have been successful in fully defining transactions formally.

Formal specification of network protocols is simpler than transaction systems because discrete synchronizable points are identifiable. Unfortunately, database transaction can share data (objects) simultaneously so operation synchronization is more difficult to specify. Formal method will clarify underlying transaction concepts and lead to simple, reliable, and correct protocol design. Additional formal methods research is required in the database transaction domain to fully understand the theoretical underpinnings and techniques. Furthermore, the notion of serializability that incorporates causality is required for indepth study.

Formal methods are rarely used in commercial developments [Col90, BSC92]. Widespread commercial acceptance of formal methods requires carefully documented, realistic case studies. In database applications, in particular, the few case studies [Wal90, Sha90b] available examine sequential and deterministic systems so they do not examine concurrency which is the core of any complex transaction systems.

### Benefits of Formal Methods

Using a formal framework for specifying transaction systems protocols offers the following benefits [NW90, NJH92, Ost91, Pre92, Win90, Flo85].

- It assists in deriving an independent and precise description of the behaviour and effects of transactions in a given transaction model.
- The formalization process can reveal ambiguities, incompleteness and contradictions in the informal product definition. Thus a formal method provides a means for specifying a system in a precise language so that consistency, completeness, and correctness can be assessed in a systematic fashion.
- It allows for correctness verification of the transactions protocols. Verifying correctness can be done using rigorous mathematical analysis and logical reasoning,

as well as tools where they are available.

- Different transaction models can be evaluated and compared easily.
- A formally verified system can be used with greater reliability. "The use of formal techniques should be seen as a way of achieving a high degree of confidence that a system will conform to its specifications." [Spi88]
- Formal methods influence the automation of the production/development of software. Also, a tool for a formal method can lead to automation of the specification model.
- Formal methods provide abstraction so a precise behavioural specification focused on a system's functionalities can be defined. Abstraction is the process of identifying the key properties being modeled while ignoring unimportant details to manage complexity and promote correctness, extensibility, maintainability, reusability, and understandability.
- Formal methods help programmers to reason carefully about the correctness of implementations of transaction processing systems since many of the proposed algorithms for transactions are complex.
- Additional benefits arising from using formal techniques are:

    1. Reduction in system development costs because errors are detected and corrected early.
    2. Reduction in the time and effort required at the detailed design and coding stages (e.g., the Customer Information Control System [CICS] project [HK91]).
    3. Identifying when reusing program modules is possible [ST90] from the formal specification.

## 1.3 Objectives of the Study

The primary focus of this research is to specify database transaction systems protocols using formal techniques. The structural and behavioural capabilities of the system can be formalized in terms of assertions and constraints that must be satisfied by any implementation. In particular, this research:

- identifies the requirements of any formal language suitable for the specification of transaction systems;
- identifies and emphasizes the importance of using formal methods in the specification of transaction systems protocols;
- formally specifies transaction systems protocols using the Timed CSP specification language; and
- provides proof of correctness for the specifications.

This research adopts the following methodology in specifying transaction systems protocols. First, the CSP's process language describes the communication interaction patterns of the system. Second, first order logic expressions specify the system's functional properties while the behavioural specifications describe system properties with traces. Finally, satisfaction relations and rigorous proof mechanisms ensure the processes described exhibits these properties. An equivalent approach but orthogonal methodology to the one adopted in this research uses refinement to investigate whether or not the process description exhibits these properties, and finally refine the process description towards implementation. This is outside the scope of this research.

## 1.4 Significance of the Research

The main significance of this research is the integration and formalization of solutions to individual transactions requirements within a single uniform transaction specification framework ( — Timed CSP specification language). This framework is more general and

not biased towards specific types of transactions. It integrates temporal behaviour of individual transactions with the dependencies among transactions that can arise when accessing shareable data. Transaction dependencies are analyzed using the notion of predicate satisfaction.

### Contributions

In summary, the following are the specific contributions of the thesis:

1. Provides a taxonomy of a transaction specification characteristics against which any specification can be assessed.

2. Identifies the requirements of any formal language that is appropriate for the specification of transaction systems.

3. Provides record data structure support for CSP.

4. Provides a formal framework for analysing database transaction functionalities and behaviours and the constraints imposed by the underlying structure of the system to ensure correctness and enhance system reliability. Specifically, the thesis formally provide:

   - specification of hierarchical transactions (closed and open models),
   - specification of *2PL* and *Timestamp Ordering* concurrency control protocols, and
   - specification of correctness criteria (e.g. *CSR* and *VSR*).

5. Provides a formal case study analysis of a new application — the Electronic Shopping Mall in the domain of electronic commerce.

6. Demonstrates correctness of schedules.

7. Other contributions are:

- demonstrates the use of software engineering discipline in the design, development and evaluation of transaction systems for cooperative computing workflow architectures or environments.
- provides solutions to problems of electronic commerce transactions correctness and reliability and identifies new challenges and directions for future research.
- provides methodology to accommodate the new complexities introduced by asynchronous operations of distributed systems.

## 1.5 Limitations

This research is limited to the formal specification of database transaction systems protocols using the formal specification language Timed CSP. The language and models of CSP are employed to specify the transaction protocols at an abstract level and to establish a proof of correctness. The implementation of the specifications is outside the scope of this research. Further, no attempt has been made to develop a new formal specification language (or calculus) but extensions to CSP include a structured data type to support transaction system specifications.

## 1.6 Problem Domain

A multidatabase (MDB) environment where each database is autonomous provides the framework to reason about formal specification. The local databases may be heterogeneous. Within the MDB environment various architectures may be supported, see Figure 1.1. A schematic representation of an MDB model components is shown in Figure 1.2. The autonomy and heterogeneity of the local databases (LDBs) provides design autonomy so different LDBs may support different concurrency control protocols, data models, data manipulation languages, and correctness criteria. Therefore, LDBs independently execute transactions in any order.

To support heterogeneity, database drivers (including ODBC[6] [Mic96]) and other

[6]ODBC (open database connectivity) technology provides seamless access to enterprise data using a

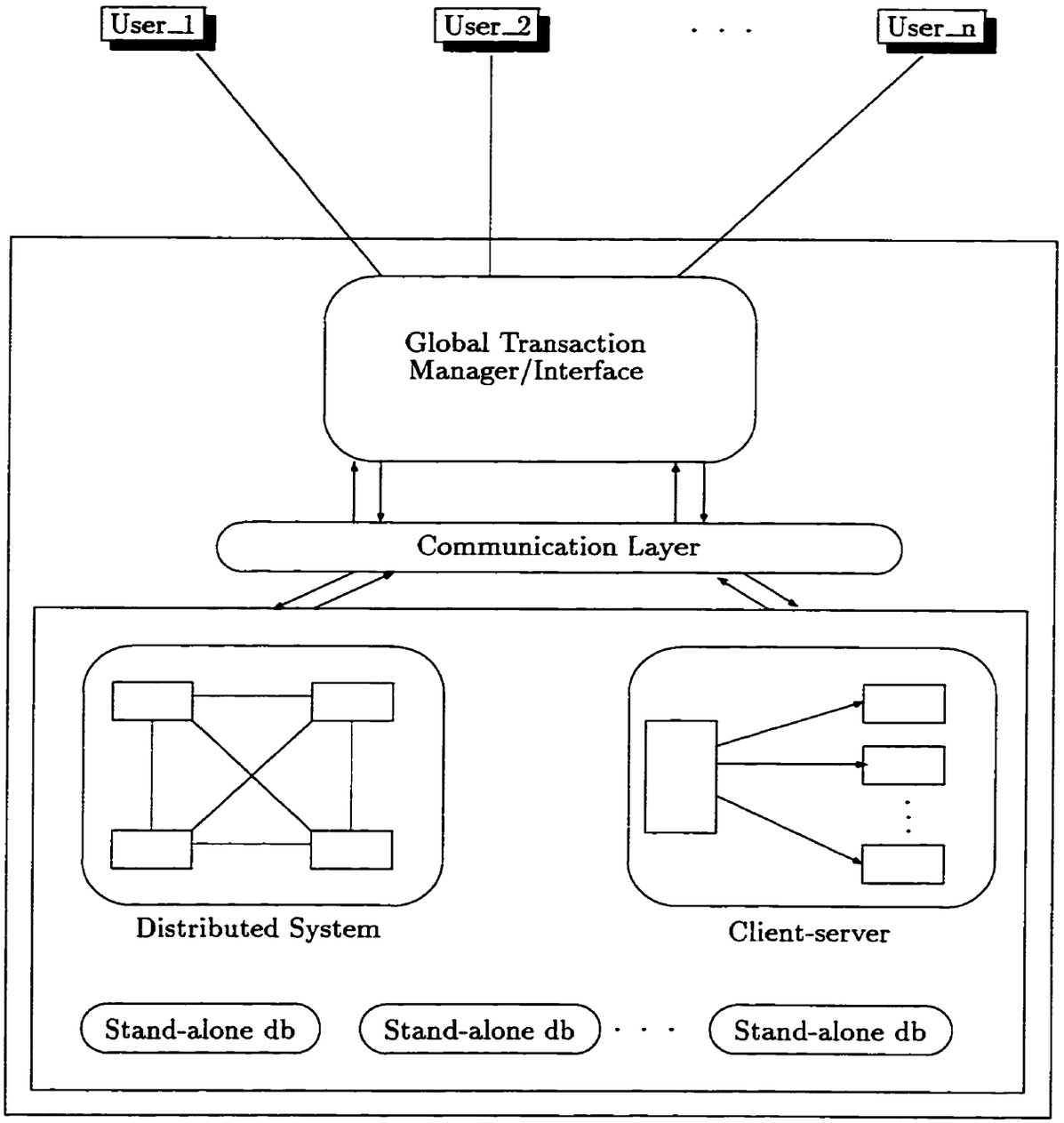


Figure 1.1: Architecture of the Problem Domain

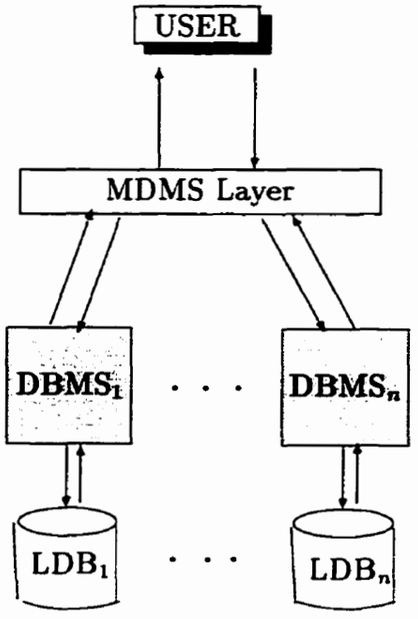


Figure 1.2: Components of an MDB Model (Adapted from [Bar90] with permission)

APIs provide access to data since the LDBs' participation in the MDB requires no modification of their code or functionality. Also, the MDBMS requires an appropriate network library to enable global transactions connect to specific databases. An intelligent agent subsystem (of the MDBS) provides the MDBS with information about the services the component LDBs can render and the data they may contain. This information is necessary to appropriately direct a query to the LDBs.

A user can submit transactions to the *global transaction manager* (GTM) to access one or more *local databases* in the system. These transactions are called *global transactions* (GT). Users at each LDB can submit *local transactions*[7] (LT) to their respective local databases. Both local and global transactions can be submitted simultaneously and executed concurrently. Figure 1.3 illustrates the computational model.

The two classes of transactions thus require two levels of transaction mechanism

single API.

[7]Local transactions are accepted directly by the individual autonomous component LDBs.

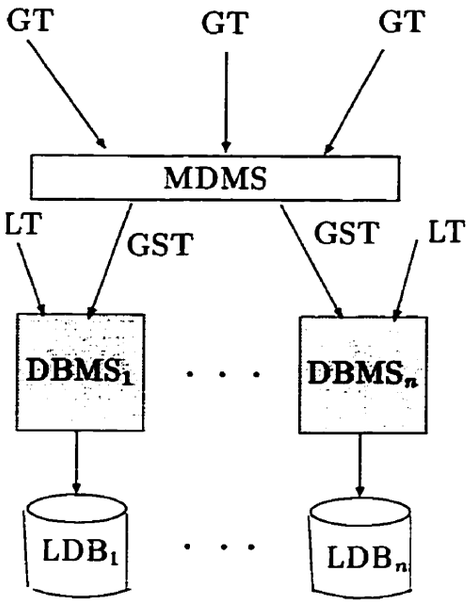


Figure 1.3: Transactions in MDB Model

support: the GTM and the *local transaction manager* (LTM) for each individual component LDB. The GTM receives all the incoming transactions (i.e., GTs), schedules transactions and assigns them to the participating LDBs for execution and supervises their (interleaved) execution. Each LDB receives the global subtransactions (GSTs) and process them with local transactions. The GTM maintains the correctness of the global transactions. Management of *commit* and *abort* operations of a transaction is the responsibility of the recovery manager module of the originating site. See [LR82, Bar90, OV91] for details of MDBMS's architecture.

The GTM uses several data structures. The *transaction queue* (TRANQ) holds incoming transactions using a FIFO discipline. The *wait queue* (WAITQ) holds transaction *ids* sent for processing or those awaiting further processing. The *status queue* (STATQ) contains status messages from the LDBs (where available) indicating transaction status such as: (a) aborted, (b) committed, or (c) active.

The component of the GTM that schedules transactions for execution is the *trans-*

*action scheduler* (SCHED). The SCHED (i) selects a transaction from TRANQ and distributes it to the appropriate participating LDBs and (ii) sends it to the WAITQ until acknowledgement is received from the LDBs. Transaction operation orderings are serialized and recovery is ensured by the scheduler. Once all subtransactions submitted for a global transaction have completed, the GT is removed from the WAITQ. The LDBs often communicate with the GTM to indicate execution status of a transaction.

The SCHED uses the following iterative steps when scheduling transactions.

1. Select a $GT$, say $T_i$, and decompose it into its constituent subtransactions (GSTs).

   - For each GST do

     Determine the appropriate LDB and assign the GST to it.

     Add the GST's *id* to the WAITQ

     End{for}

   - Then remove $T$ from $TRANQ$

2. If $STATQ = \langle\ \rangle$ then get transaction $T_i$ from TRANQ else get transaction $T_j$ corresponding to the message status from the WAITQ. That is,

   $STATQ = \langle\ \rangle \Rightarrow T_i = \textbf{head}(TRANQ)$

   $STATQ \neq \langle\ \rangle \Rightarrow T_i = T_j \mid T_j \text{ in } WAITQ \wedge T_j = \textbf{head}(STATQ)$

   One of the following actions will be taken depending on $T_j$'s status.

   **case of** *status* **do**

   *active*: schedule **head**($TRANQ$)

   *abort* : **if** GST = vital **then** abort($T_j$)

   **else** schedule next GST of transaction $T_j$

   *commit*: schedule next GST of transaction $T_j$

   end {case of}

3. If $T$ is partially processed transaction then mark the completed GST.

4. If $T$ is completely processed, remove $T$ from $WAITQ$ and $STATQ$.

The GTM submits operations to LDBs one at a time using a blocking protocol where each submission is acknowledged before subsequent operations can be submitted.

This environment is probably only feasible if a flexible transaction model is employed that allows for compensating and/or contingency transactions to recover from potential semantic failures [HPS93]. Possible problems include:

- Concurrent access to a data item (by both local and global transactions).
- Deadlock problems (queue resolutions)
- Temporal components of some transactions need explicit specification so that we can capture interleaving order of transactions.
- Reliability issues are particularly important and difficult problems in this domain.

Application areas of the above model abound such as: (1) Shopping malls with several business concerns each maintaining their own local database (i.e., the electronic shopping mall paradigm), (2) Travel and associated operations, e.g., booking flight, hotel reservation, and car rental services, (3) Governmental operations (two or more government ministries or organizations interacting), (4) School environment (library system, accounts system, students records, registration system, etc), and (5) Co-authoring systems.

### 1.6.1 An Example — The Electronic Shopping Mall

The Electronic Shopping Mall (ESM) illustrates the pragmatic aspects and clarifies semantic related problems. The intent here is to demonstrate the application of formal methodology in protocol design and analysis. The application easily scales up to other designs without loss of analytic power. A diagrammatic representation of the transactions in the application domain under consideration is shown in Figure 1.4.

ESM is an electronic commerce[8] application. ESM is an electronic equivalence of ultra-large department stores (e.g., the West Edmonton Mall in Alberta, Canada) with a large number of vendors and products. ESM provides a common online access point (the global transaction interface/manager) where customers can obtain information

[8]Electronic commerce is the sharing of business information, maintaining business relationships, and conducting business transactions by means of telecommunications networks [Zwa96].

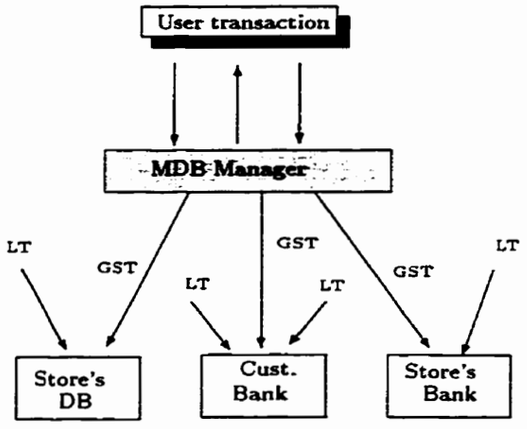


Figure 1.4: Abstract Representation of ESM Transactions

about products and place orders. It must *automatically* process customer's electronic orders that requires the services provided by a company's inventory, billing, accounts, and logistics operations. Thus, it supports a user's access to multiple databases. ESM integrates electronic payment into the buying process and *information kiosks* [Jac96][9] to build a consumer marketplace. The ESM has added a new dimension to satisfying vendor/customer needs. The ESM uses a combination of videotext, graphics, and other multi-media to provide detailed product information for the customer. The ESM dialetic provides both effective product differentiation and increased ability to market products for the vendors and an efficient comparison mechanism for the customers (by increasing the ability of customers to shop and compare prices and products). The focus of this thesis is only on the specification of the underlying sales transactions aspects of ESM functionalities.

A request to the ESM typically involves many subactivities that may share resources, demand explicit expression of temporal relationships, or require concurrent and parallel operation execution. Thus, the application must capture concurrency, share resources, temporal relationships, and the correctness properties necessary in a transaction

[9]Information kiosk is an interactive display module that guides shoppers towards a store that carries an item of interest and the inventory

system.

If a user wants to buy merchandise, say a bicycle, the user issues a transaction againsts the global database. The GTM sends the transaction to all the participating merchandise stores' databases that sell the product. That is, the GTM decomposes the user transaction and queries all participating local databases for availability of the item. Local databases that have the item acknowledge availability status to the GTM while the local database continues to process LTs at its location and hold the requested item for the "global" customer. The GTM immediately aborts the transaction if the item is not available otherwise the GTM requests payment, validates account balance, and completes payment if account balance is greater than the selling price or immediately aborts otherwise. If the payment action succeeds, then the local store database transaction (inventory update) can commit so the GT can commit successfully.

The same global transaction, however, may have to access the customer's bank account to check for fund availability. In many cases, that information may be kept on a different system. This introduces heterogeneity thus resulting in transactions that may have to be split and executed on different systems or possibly on a remote LDB. Also, there may be value dependencies whereby the action taken at one LDB depends on the value of data item in another. For example, in the preceding bicycle example, the decision to complete the inventory operations depends on the amount of money the customer has (the customer's account balance which may reside in a bank's database).

The GTM may present a variety of information to the user. Such information may include: a list of all stores selling bicycles, the different models of bicycles, the price listing, the list of quantity available, the locations of the stores selling bicycles, stores having sales promotion currently, and so on. The items filtered out from the user's view depend on the requirements of the GTM. The user pays for the item if the product available meets the description in the user's transaction. A user can pay for an item using a variety of methods such as direct debit from an account, electronic fund transfer, telepay, credit card, interbank transfer, cash, or any combination of the above. The user's transaction completes after completing payment and the account balance is updated. Further, the user's transaction should not prevent other transactions from

taking place concurrently both at the participating stores' local databases, the global database, or actions executing at other locations resulting from the execution of the user's transaction.

A summary of additional constraints are: (1) a customer may buy as many quantities of the same item at once as allowed by the participating stores, (2) a customer cannot buy items worth more than the customer can afford, (3) all items in the participating local databases must be sold or available for sale, (4) only authorized persons may add or remove an item from the store, (5) no item may be sold and available for sale, and (6) the set of participating local databases should be nonempty.

To the user, the provision of a fast service, correct account balance after payment, ability to pay using any method, and transparency of the services provided by the participating local databases constitute good behaviour. In addition, the participating local databases enforce the user transaction's integrity constraints, allow operations concurrent execution on the database simultaneously as the user transaction execution progresses, terminate global subtransactions after processing, and finally make the service they provide to the GTM transparent to their respective local users. The coordination and complexity of all the activities are transparent to all users. Lastly, the GTM can terminate any activity requested by the user after execution and delegates any activity to participating local databases for the satisfaction of a user's transaction.

For discussion purposes, suppose only one store has a bicycle matching the specifications in the transaction. Further, assume that only one bicycle remains in the store. Unfortunately, several scenarios may prevent a smooth purchase transaction.

1. If a local transaction simultaneously requests the same bicycle a decision must be made as to how we resolve this situation.

2. Suppose the user in the global transaction decides to pay for the item with the *Interac*™ system on a bank account while his/her spouse is at a local branch processing a withdrawal request at the same time. How do we resolve the concurrent update problem *vis-a-vis* the bank?

3. With current technology, suppose the global transaction request was successfully

processed and a signal was sent via satelite to update the inventory. If another user requests the same item while this is yet to complete processing, the system might incorrectly reports that a bicycle is still available. Thus the timing aspects of transactions must be further examined.

To exploit the technologies at our disposal, multidatabases must be formally specified to ensure correct concurrency, reliability, and recovery. Additional considerations include:

- It is possible that more than one local database can satisfy the request of a global transaction.
- The global transaction will always spin off a subtransaction at a local database. Thus, we have a collection of subtransactions nested within the global transaction at the top level. This ultimately amounts to *nested transaction* management [Gra81, EGLT76]. Allowing concepts like vital and non-vital subtransactions and supporting both closed and open nesting is likely required.
- The local DBMSs manage the subtransactions spawned at their sites by the global transaction to ensure correct execution. This immediately suggests two levels where temporal properties need explicit specification, at the global transaction level and at the local transaction level.
- Different levels of correctness are possible in this environment, some more restrictive than others.

In summary, ESM is characterised by active capabilities (for timely response to events and changes in the environment), support for long-running transactions (and possible partial sharing of results), allows compensation to undo effects of undesirable committed transactions, and support for heterogenous and autonomous environments.

## 1.7 Notations

Let us make some comments about a few notations that we shall encounter. The syntactic form

$$function\_name : argument\_type \longrightarrow output\_type$$

defines the signature of a **function**: where its name is *function_name*, *argument_type* is the data type of the argument(s), $\longrightarrow$ is read as "produces", which relates the input data to the output, and finally *output_type* is the data type of the output. Where the function is a relational type, $\longrightarrow$ is replaced by the relational function symbol, $\longleftrightarrow$ and by the partial function symbol, $\nrightarrow$, for partial functions. This convention bears close resemblance to that used in the Z [Spi88] specification language. Also, $\mathrm{seq}_1$ denotes a nonempty sequence, $\mathbb{F}_1$ denotes a nonempty finite set while $\mathbb{P}$ is a powerset. Additional symbols are defined in context when necessary.

It may help to browse through Appendix A for list of symbols at this point before proceeding.

## 1.8 Organization of the Thesis

This chapter introduces the concept of transaction models and formal specification and demonstrates why a formal methodology is appropriate. Chapter 2 reviews related literature on transaction models, concurrency control for complex objects, and specification techniques currently in use. Chapter 3 presents a taxonomy of transaction specification features and the fundamental issues that should be addressed before specifying transaction systems formally. Chapter 4 briefly describes the notations of timed and untimed models for CSP. Also, the necessary record data type extension to CSP for supporting transaction specification are presented. Chapter 5 presents the specifications itself. It focuses on using the formal languages to define existing hierarchical transaction models. The ones addressed are: nested transactions, multidatabase transactions, and an emerging application called the Electronic Shopping Mall concept. Chapter 6 provides the proof of specifications found in Chapter 5 leaving Chapter 7 to make some concluding comments.

# Chapter 2

# LITERATURE REVIEW

This chapter is organized into two major parts. First, formal methods and the current state of the art for specifying transaction systems protocols are described. A survey of existing formal specification tools is provided to assess their strengths and weakness, and to determine their suitability for specifying transaction protocols. Secondly, relevant database material including concurrency and recovery is discussed. These parts are necessary to understand the link between specification and transaction systems.

## 2.1 What is a Formal Method?

A formal method is the combination of a precise operational system abstraction and the ability to argue rigorously about the behaviour of the system. Formal methods are "mathematically based techniques for describing system properties. Such formal methods provide framework within which people can specify, develop, and verify systems in a systematic, rather than *ad hoc* manner" [Win90]. The sound mathematical basis provides the means to precisely define notions such as consistency, correctness, and completeness of specification and the resulting implementation.

Formal methods, therefore, "are essentially formal systems; they provide frameworks for inspecting the satisfiability of specifications, for proving the correctness of an implementation of a system, and for proving the properties of a system without the need to have an executional representation of the system" [NJH92]. In other words, a for-

mal method consists of a formal model and the associated mathematical techniques that provide the user with a framework for specifying and analysing the system [MLTL93]. The basic components of a formal method are: the computational model of development; a specification language; properties (liveness and safety properties) to be preserved; a proof system; and guidance in applying these in a coherent manner. Most methods lack a proof system and guidance about how development should proceed while others lay very little emphasis on an underlying model that encompasses each stage of the system development.

Formal methods are useful at various *abstraction levels* because each provides different levels of assurance for the software developed [WKC91]. Thus, formal methods are used in three ways [Lus94]: for expressing the statements; for verifying that the input and output of a step are in agreement; and for transforming the input into some output. This research uses formal methods in the first two ways.

### 2.1.1 Formal Specification Language

A formal specification language consists of three parts: the *syntax* that describes the set of allowable alphabets and the grammar of the language; the *semantics* which makes it possible to denote the meaning of a specification in the language without ambiguity; and a set of *relations* that defines which objects satisfy each specification. Thus [GHW82] provides the following definition:

**Definition 2** *Specification language:* A formal specification language is a triple, $\langle Syn, Sem, Sat \rangle$ where $Syn$ and $Sem$ are sets, and $Sat = Syn \times Sem$, is a relation between them. ■

$Syn$ is called the language's syntactic domain, $Sem$ its semantic domain, and $Sat$ its satisfies relation.

The syntactic domain of a specification language is usually based on the principles of set theory, predicate logic, relations and functions. The satisfies relation defines the rules for forming well-formed formulae (*wffs*). A well-formed formula (wwf) over a set of syntactical rules, $G$, is a finite sequence of symbols that is syntactically correct. That

is, it belongs to the set of all sequences of symbols that can be constructed by using the grammar of $G$.

A specification language may be viewed as a set of specification building operations together with some syntax. In choosing the class of operations there is a trade-off between the expressive power of the language and the ease of understanding and dealing with operations [ST84] in addition to the levels of abstraction that it supports [Win90]. Making a language more expressive does indeed facilitate briefer and more elegant specifications, but can make reasoning more difficult [BH94]. An example from the programming language environment is *APL* which is a very powerful language having operators that concisely perform actions requiring numerous statements in other languages. Thus, an APL program can be extremely abstract but may be "difficult to explain and understand which in turn becomes a potential hindrance to producing correct programs" [McC83].

### 2.1.2 Taxonomy of Formal Specification Language

Several formal specification languages are currently being used in different semantic domains. Wing's taxonomy will suffice [Win90]. These consist of: (1) Model-Oriented approach, (2) Property-Oriented approach, (3) Visual languages, (4) Executable codes, and (5) Tool-supported.

Model-oriented approaches specify explicitly a state space (or an abstract) model of the system's behaviour by using mathematical structures such as sets, relations, functions, sequences, and cartesian products. Examples of languages in this class are Z [Spi88] and VDM [Jon86] for describing sequential systems and CCS [Mil80], CSP [Hoa85], ACP [Ber88, BB91], Raise [Nie89], and Petri nets [Pet77] for concurrent systems.

In the property-oriented approach, the specification language defines the system's behaviour indirectly by defining a set of properties or constraints which the system must satisfy. These constraints of the system's behaviour are often stated either axiomatically using first order predicate logic or algebraically using axiomatic equations. The axioms specify fundamental properties of a system and provide a basis for deriving additional properties implied by the axioms. To establish a valid mathematical system, the set of axioms must be complete and consistent (that is, it must be possible to prove desired

results using the axioms, and it must not be possible to prove true contradictory results). Examples of formalisms in this category are LOTOS [ISO87], Temporal Logic [RU71], Larch [GHW85], ACTA [Chr91], Anna [LH85], and Clear [BG80]. LOTOS, Temporal Logic, and ACTA are methods for specifying concurrent and distributed systems.

Visual languages are graph theoretic based formalisms. "Visual methods include any whose language contains graphical elements in their syntactic domains" [Win90]. In other words, visual specification methods use visual expression where the graphics themselves are the syntax of the language. Visual languages use some visual representation to accomplish what would otherwise have been written in traditional prose. Examples are Petri nets and the higraph based Statecharts [Har88].

Executable formal methods support specifications that can be executed. Examples of this category are the programming language Prolog and OBJ [Gog88]. Although a logic based programming language, Prolog can be used as a specification language in property-oriented form by defining logical relationships on objects.

Languages that have tool support for at least one of syntax checking, semantics analysis, theorem proving, animation and graphical user interface are classified as tool-supported. Most of the above mentioned languages including VDM, Z, and LOTOS are examples.

The following section provides a brief discussion of some current methods (informal and formal) used in specifying transaction systems protocols. Some of the formal specification languages mentioned are assessed for their suitability in specifying transaction systems protocols.

## 2.2 Current Transaction Specification Techniques

Current techniques utilize informal methods classified broadly into: natural language, pseudocode, and state-oriented.

### 2.2.1 Natural Language

Most requirements specifications are written in natural language and range in length from a few pages to several thousand pages. The size of the document rarely has any relationship to the complexity of the problem [Dav88]. Breitbart *et al* [BST90], Eswaran and Chamberlin [EC75], and Gracia-Molina and Salem [GS87] use natural language to state a transaction models' properties and behaviours. Such specifications tend to be imprecise, incomplete, and unverifiable; and they are inherently ambiguous because they are subject to different interpretations.

### 2.2.2 Pseudocodes

In [BR90, Bar90, Des90, KJ90, KS86, OV91] a transaction model's properties and behaviours are defined using pseudocode based on more pragmatic English-like syntax rather than on mathematical formalism. The pseudocode expresses synchronization algorithms such as locking, timestamping, serializability and recovery protocols that define how a transaction model should behave at any point in time given some conditions or stimuli. Proofs of correctness for these systems rely on intuitive operational arguments that are potentially ambiguous, error prone, and difficult to formally verify. Thus, systems developed from such specifications are less than ideal because they lack an underlying abstract mathematical model and consequently the precise semantics are not fully specified.

### 2.2.3 State-Oriented Techniques

State-oriented methods such as tables and graphs are often used to describe certain properties of a transaction model. Compatibility tables (also called compatibility matrices) are often used to define the behaviours of transaction operations over shared objects. Compatibility tables are useful in specifying system behaviour when different input data give different actions (states or outcomes) for each of several different modes of operation. This tool is used extensively [BR90, DK83, Des90, KL83, KS86, OV91] to define the lock compatibility function and the commutativity of operations required for synchronization.

The example [BR90] in Figure 2.1 shows the compatibility of the set of operations insert, delete, and member. Two operations are compatible if their effects are independent of execution order. In Figure 2.1, **Yes** means the operations are commutative while **Yes-DP**

| Operation Requested | Operation Executed | | |
|---|---|---|---|
| | ***Insert*** | ***Delete*** | ***Member*** |
| ***Insert*** | Yes | Yes-DP | Yes-DP |
| ***Delete*** | Yes-DP | Yes-DP | Yes-DP |
| ***Member*** | Yes-DP | Yes-DP | Yes-DP |

Figure 2.1: Operations Compatibility Matrix [BR90]

means they commute only when different input parameters are used.

Another state-oriented technique is demonstrated by digraphs which are used to model the serializability criterion (See Section 2.4 page 42). Testing the graph for cycles can unambiguously determine if the transaction's execution sequence is valid. Another useful application of digraph is for modelling deadlock via wait-for-graphs (WFG) among transactions[1]. The WFG represents wait-for relationships among concurrent transactions in a system. Like serialization graphs, an acyclic WFG guarantees the system is deadlock free. Digraphs are used in [BST90, Des90, KS86, OV91] to specify aspects of concurrency and the transaction model's correctness criterion. The algorithms for constructing these digraphs often use natural language or pseudocode in their description.

A state-oriented form of table and graph usage is the finite state machine (FSM) approach. Behaviour specification using regular expressions and FSM are by no means a new idea. The transaction's behaviour is taken as the sequence of actions of the FSM. This approach was adopted in [AFLMW88]. When using FSM to specify transaction systems, control data flows coming out of a node may be "and-ed" [Lus94] together thereby defining concurrent executions of some parts of the transaction. A variant of

[1]The wait-for-graph is a directed graph containing nodes (representing active transactions) and arcs (representing wait-for resource relationship between the nodes). See [Des90] pages 595-598 for the WFG's construction algorithm.

FSM is communicating real-time state machines (CRSM) [Sha93]. A major shortcoming of CRSMs is that the communicating machines cannot access shared data (see Section 2.3 of [Sha93]) which renders it inappropriate for database applications.

These approaches (tables, graphs, FSM) have limited practical use because they:

- capture only some of the transaction model's properties and cannot be used to reason fully about the structure and behaviours of transaction models,
- lack formal semantics and therefore are unverifiable, and
- cannot capture timing properties of complex transactions.

The following additional problems are associated with state-oriented approaches:

- The diagrams are "flat", so complex transaction systems cannot be adequately described without providing a hierarchy to highlight the appropriate level of details.
- Conventional state diagrams are sequential and do not cater for concurrency.
- They suffer from exponential state space growth that must be explicitly represented as the system grows.

Another example of behaviour specification using FSM is *Statecharts*. They specify the input and output of a system in a hierarchical manner in terms of operations among sets of states. Statecharts specify a *mealy machine*[2] [HU79] so they specify how a deterministic system should behave and react to environmental inputs.

Statecharts have recently been integrated with object-model diagrams (a kind of entity-relationship diagram with higraph encapsulation that describes classes and their structures) [HG97]. They are one of the (seven) models of an emerging standard for object-oriented modelling called the Unified Modelling Language (UML). A Statechart in UML describes the dynamic behaviour of objects instantiated from a single class.

Using Statecharts as a stand-alone technique suffers from discontinuity in its transition to design. Most of the problems associated with structural analysis methods are still

[2]A mealy machine is a model in which the output depends on both the state and the input.

present in Statecharts. Mechanisms for synchronizing concurrent access to data (which is essential in database transactions) are not provided. Thus, inter-object communications and collaborations cannot be effectively modelled using Statecharts because synchronization during concurrent data access cannot be captured. Further, according to Ostrof [Ost94] time constraints are not treated in sufficient details in Statecharts. Statecharts require notations for periodic timing functions and the specification of timing exceptions without the need to introduce additional states.

### 2.2.4 Functional Decomposition Techniques

Many functional decomposition techniques, such as Structured Analysis [DeM78], Structured Design [YC79], and Structured Analysis and Design Technique [Sof78], provide an organized set of system specifications and a structure of the system. The specifications include views such as (i) data flow diagrams to decompose the system, its functions and its data flow, (b) control flow diagrams to represent the system dynamics, (c) a specification dictionary listing all inputs, outputs, and control flows, and (d) a table of response times for all events.

These methods use different diagrams (e.g., DFDs, Structure charts) and different ideas during development so it is difficult to transform from one to another. Further, there is no explicit indication of the control flow specifying the execution order of the various objects. Although dataflows generally carry data, they may also be triggering signals for functions but this is not explicitly captured. The ability to specify explicitly the execution *order* of the operations that would trigger subtransactions is required.

Ostrof [Ost94] argues that although these methods are quite successful in industrial applications with little or no concurrency, they have the following additional shortcomings: (i) these methods have no formal semantics, (ii) the timing properties are not particularly well integrated with the rest of the requirements, (iii) there is no support for formal verification, and (iv) nondeterministic systems behaviour cannot be suitably modelled.

*In summary*, using a state-oriented notation in conjunction with algebraic axioms allows precise specification of interactions between operations and the behaviour of the

individual operations. The axioms can be manipulated in a rigorous manner to provide for the proof obligation of a specification language.

Apart from the methods discussed above, other approaches for describing transaction systems protocols are: ACTA, First order logic, and Prolog. ACTA is a semi-formal method while Prolog is a programming language. These approaches are discussed below.

### 2.2.5 ACTA

ACTA is a unique formal framework for specifying and reasoning about the structure and behaviour of transaction systems. ACTA is proposed as a mechanism to unify all transaction models. ACTA does not propose any particular correctness criterion but it provides components for the description of transaction models. Components are the set of symbols that are put together to form the sentences or statements of the formalism (language). These components can be combined into logical statements where correct behaviour satisfy the statements. Its approach is an axiomatic property-oriented method. The ACTA framework has explicit notions for some model components that are implicit (e.g., begin transaction operations) in transactions and it has been used to model both atomic and complex transaction systems (see [Chr91, CR90, CR91, CR92a, CR92b]). With dependence analysis between transactions and transactions operations on shared objects and by associating semantic elements with the effects of transactions operations, ACTA can capture transaction properties related to visibility, failure atomicity, permanence, and consistency.

However, ACTA is not standardised and it lacks a computational model so it is *ad hoc*. ACTA has no proof obligation and guidance for applying these in a coherent manner. Furthermore, "it is not clear how ACTA would represent the fact that a dependency can be transient, nor is it clear how proofs could be developed with transient dependencies involved" [KS94].

### 2.2.6 Prolog

Another language used to specify transaction protocols in a communication network [VW87] is Prolog [CM87]. In this sense, Prolog is used in property-oriented style to state logical relationships between objects thus providing a means for expressing and handling concurrent interactions. Vuong and Weber [VW87] use Prolog to specify the communication behaviour of transactions based on communicating FSMs and validate the specification based on reachability analysis principles for detecting state deadlocks, nonexecutable interactions, and unspecified receptions. Although Prolog is the most widely available logic programming language and is useful as a specification language in communicating systems, its use in database transaction systems is limited because it lacks the semantic analysis essential for defining correctness and recovery properties in complex transaction systems. Further, the reachability analysis suffers from state explosion when the system becomes complex or large which means exponential search spaces become problematic.

## 2.3 A Survey of Existing Tools

This section assesses the strengths and weaknesses of the existing formal specification languages and discusses their applicability to transaction systems.

### 2.3.1 Z

Z is a mathematical notation based on set theory, first order predicate logic, sequences, functions, and relations. In general, Z is non-executable and is typically used for human readable specifications. It is essentially a two-value logic in which every proposition is considered either true or false. That is, if $A$ is true then $B$. However, when $A$ is false the consequent is unspecified. It is model-oriented but can also be used in the property-oriented approach when used axiomatically.

The *schema language* is a graphical extension of Z. Schemas are devices for organizing the presentation of the mathematical notations of Z specifications. A schema has two

parts: a declaration part [above the dividing line] contains the declaration (definition) of one or more identifiers, and a predicate part (below the dividing line) contains zero or more predicates separated by semicolons. The predicates describe a property of the schema's declared variables. The schema language is flexible so modularity is relatively easy to describe and implement [HO93].

In practice, a Z specification consists of a series of paragraphs of formal notations interleaved with informal prose that explains the content of the formal notations. It uses pre- and post- conditions implicitly as constraints that must be satisfied and has a proof assistant (type checker) for consistency checks. It has been found useful in specifying large commercial systems, for example, the CICS [HK91] project. Z expresses functionality but not concurrency. Further, Z presently lacks extensions to capture time properties.

### 2.3.2 VDM

VDM is based on denotational semantics. VDM is similar to Z except it is model-oriented and pre- and post- conditions are explicitly stated in separate clauses. A precondition on an operation is a predicate that must hold on each invocation of the operation. If it does not, the operation's behaviour is unspecified. A postcondition is a predicate that holds in the state upon return [Win90].

VDM encourages hierarchical system development by supporting abstraction at the highest level of description and by providing a powerful and succinct tool for expressing specifications. However, in its currently published form, VDM is most suitable for specifying sequential information processing systems. There is no concurrency or real-time support in standard VDM. Different system views or schemas are not supported. The degree of modularity is limited to the level at which operations are defined. Furthermore, VDM has no graphic support [HO93] and lacks convenient facilities for defining and handling errors in a specification to eliminate the difficulty of distinguishing error behaviour from that of normal behaviour [JDS85]. Although several case studies involving the use of VDM exist [JS90], these systems are sequential and deterministic.

**Summary**

- Z and VDM are adequate for handling sequential deterministic systems (i.e., they can capture many interesting sequential systems properties). However, they are less attractive for expressing concurrent system behaviour (where properties such as freedom from deadlock and fairness often need to be shown).

- Synchronization of access to shared variables can be handled in Z and VDM by building models of histories (mutual exclusion). Mutual exclusion is a simple version of concurrency control but in transaction systems it is necessary to describe the interleaving of concurrent operations/processes.

- Z and VDM lack notions of logical time and relative order.

### 2.3.3 Temporal Logic

Temporal Logic is a property-oriented approach for specifying concurrent systems. First order predicate calculus is used to reason about expressions containing time variables. A sequence of program actions is modelled as the basic unit of specification.

It uses special modal operators to describe the past, present, and future states (events) of the system's behaviour. For example, the $\Box$ (rectangle) symbol means "in all future states"; $\Diamond$ (diamond) means "in some state"; while O means "next state".
$\Box P$ means in all future states the predicate P holds while $P \longrightarrow \Diamond Q$ means if P holds in the current state, Q will eventually hold.

Unfortunately, no standard set of operators are used (the above symbols are the most common representation). It captures time properties using different types of temporal semantics (such as linear, parallel, branching, continuous, or discrete) and it has hidden clocks (bounded operators) and explicit clocks. For example $A \longrightarrow \Diamond_{\leq 5} Q$ means if A occurs then eventually within 5 time units B must occur.

Temporal Logic captures concurrency correctness (i.e., "safety properties of the system and its environment" [Win90]) as well as liveness properties. It uses an unstructured set of predicates. Specification proofs utilize proof diagrams. A proof diagram is an abstract view of a state reachability graph representing the sequences of behaviour of

the system. Various forms of Temporal Logic exist, e.g., Metric Temporal Logic (MTL) and Real-time Temporal Logic (RTTL) [Ost91].

### 2.3.4 Petri Nets

A Petri net is a graph based formalism developed in the early 1960s as a solution to some of the limitations of finite state mechanisms. Petri nets are an interesting graphical technique used to describe the operation of a system as a state transition network. Causal dependencies and independencies in some set of events are explicitly represented. The basic net model is of the form *condition-events*. Petri nets are used for modelling and analysis of systems. They can specify synchronization and mutual exclusion among concurrent operations. Petri nets have been used in performance modelling [CL92], for specifying safety requirements [LS87], and to specify process synchrony during the design phase of time-critical applications [Dav88]. A formal definition of Petri nets follows.

**Definition 3** *Petri nets:* A Petri net is a 4-tuple, $C = \langle P, T, A, M \rangle$ where $P$ is a set of places (representing conditions); $T$, a set of transitions (representing events); $A$, a set of arcs denoting the flow relation (i.e., elements of $A$ are arcs between places and transitions such that $A \in \{P \times T\} \cup \{T \times P\}$, $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$); and $M$, a marking (i.e., $M : P \longrightarrow I \mid I \in \mathbf{N}$) is a distribution of tokens[3] to the places of the Petri net. ■

Usually, a Petri net model is represented as a bipartite digraph where the nodes represent places and transitions. Places are drawn as circles (◯) while transitions are drawn as bars (|). For each transition, the directed arcs define its input places (arc from place to transition) and its output places (arc from transition to place). A Petri net is executed by defining a marking and then firing transitions. Tokens move from place to place through the firing of a transition. A transition fires by removing one token from each of its input places and adding one token to each of its output places. The firing rule for a transition is *enabled* when all input places contain at least one token. If two transitions share input places then they are in conflict and only one of them can fire.

The behaviour of a Petri net is captured by the reachability graph. The reachability graph of a Petri net $N$ is the edge-labelled graph $N = (V, E)$ whose vertices $V$ are $(M_0)$

[3] A token is represented on a Petri net by a small solid dot (•) in a place.

the reachable markings of $N$ and edges $E$ such that there is an edge labelled $t \in E$ from $M$ to $M'$ if $M \xrightarrow{t} M'$ for a reachable marking $M$. Petri nets do not support functionality and it is impossible to determine the interleaved partial orders of concurrent operations. In addition the reachability graph for analysing Petri nets suffers from state explosion when the system is large. The reachability problem is exponential time and space-hard [Ost91]. A further problem of Petri nets is that numerous dummy states are generated during transition delays to maintain the feature of instantaneous firings [GF90] and to maintain logical consistency. In addition, the semantics of Petri nets make it difficult to distinguish between precedence and causality [TM91].

The general Petri Nets model has often been criticised for the following additional reasons.

- inability to deal with fairness and data structures,
- structuring mechanisms such as composition operators are not inherently part of the theory,
- lack of a calculus to transform a net into a real-time programming language, and
- a place in a Petri Net cannot easily be identified with a place in the corresponding program code.

Variations of Petri Nets model exists such as Colour Petri Nets, Timed Petri Nets, Timed Interval Coloured Petri Nets, and Hierarchical Petri Nets. In Colour Petri Nets (CPN) tokens themselves may have different values (or colours). CPN can distinguish between different data types (objects, resources, tokens) to further define different executions based on token types. The expressive power of coloured petri nets depends on the cardinality of the colour set. Although this is more concise system representation and able to model data and resources, as long as the number of colours is finite, a coloured net is equivalent to a much larger ordinary Petri Nets and thus inherits the problems associated with ordinary Petri Nets. An infinite number of colours provides CPN greater expressive power. However, an immediate consequence of the universal expressiveness is

that boundedness, safety, liveness, and reachability of markings properties become undecidable in the general case. For a detailed discussion of the variations of Petri Nets model and their attendant problems see [Aal92, CK92]. Petri Nets are still undergoing standardization[4] [WD97] and intensive research [PNPM91] aimed at putting Petri Nets theory on firm mathematical grounds and some experimental and commercial simulators (tools) are available.

To summarize, although Petri nets are a good modelling tool they do not provide design details. They cannot model the data managed by transactional systems. Petri nets generally serve as a model for understanding and system analysis. The transformation from petri nets to an implementation is a difficult process.

### 2.3.5 Communicating Sequential Processes

Communicating Sequential Processes (CSP) is a model-oriented approach to specifying concurrent processes but uses property-oriented approach for stating and proving properties about the model. CSP is concerned purely with the communication patterns of processes, abstracting away internal state information which may be separated from communication behaviour. It is based on the model of traces (behaviour of a process or event sequences) and assumes that processes communicate by sending messages across channels. It views a process exclusively in terms of its observable behaviour [Hoa85].

Processes synchronize on events. For example, let $c$ be an event, and $P$ and $Q$ be processes. Then $(c \longrightarrow P) \parallel (c \longrightarrow Q) = (c \longrightarrow (P \parallel Q))$ means that the operations P and Q are parallel operations which are triggered by the event c. It uses handshaking, via semaphore or condition queues, to synchronize events. It uses primitive simple data types and cannot adequately capture the functional aspects of a non-deterministic system. However, CSP provides mechanisms and tools (techniques and languages) for the design of large systems in a modular and extensible way.

Algebra of Communicating Processes (ACP) [Ber88, BB91] is similar to CSP. They have the same underlying formal model but different notations. CSP is much more

[4]The time table for the standardization stages proposes December 1998 for the adoption of the draft International Standard [PNS96].

matured and standardised than ACP but may prove useful in subsequent research. The Timed CSP language is used here so a detailed description is provided in Chapter 4.

### 2.3.6 Calculus of Communicating Systems

Calculus of Communicating Systems (CCS) models a system's behaviour as a set of states and associated events. It models concurrency control but not functionality. The inventor [Mil80] states "... the behaviour of a system is nothing more or less than its entire capability of communication". CCS embodies a relatively complex algebra and a relatively fine grained process concept that makes learning and writing specifications difficult. However, CCS is not standardised [NPL91] so many variants exist.

#### CSP and CCS Compared

A brief discussion of the similarities and differences between CSP and CCS is illustrative. For a detailed discussion refer to Hull and O'Donoghue [HO93], Formal Systems [FS93], Hoare [Hoa85], and Cohen *et al* [Coh86].

1. Both methods support communication by message passing. The primitive for communication is handshaking where the action of sending a message by one process and receiving the message by another process is regarded as a single indivisible atomic action. Either the sending process must wait until the receiver is ready to receive or the receiver must delay until the sending process sends.

2. Also common to both methods is the decomposition of a system into a hierarchy of parallel processes which communicate by message passing. Only the notation used makes CCS's form of decomposition different from that of CSP. For example, $P \mid Q$ in CCS is similar to the CSP $P \parallel Q$

3. Both methods allow explicit rather than implicit specifications. Both represent the execution of a process as a sequence of events which may be finite or infinite in length.

4. Events consume no time so two processes operating concurrently without interaction are described as interleaved events in both. Where processes do interact, internal communication events force sequencing in both processes. Therefore, systems whose processes execute in parallel only produce a sequence of events.

5. Both methods provide selection between subprocesses depending on the last event. A process can be defined recursively to achieve the effects of event repetition.

6. Constraints can be imposed on processes by restricting their alphabets (that is, the events in which they can engage), the order in which the events can take place, and the conditions under which they can execute.

In summary, Hull and O'Donoghue [HO93] state that the strong family relationship between CSP and CCS is they share a common explicit specification method which has few technical differences. They relate to each other also by their ability to express concurrent applications and by the message passing approach taken. Some of the differences between the two languages are:

1. CCS provides both a diagrammatic and textual view of the system decomposition. There is no graphic or diagrammatic support in CSP. The semantic domain of CSP processes is the set theoretic model while that of CCS is expressible with *action trees* [HO93]. Moreover, the semantics of CCS is based on the structural approach modelling a process in terms of its possible states and transitions. The semantics of CSP, on the other hand, models a process in terms of the sequences of events it can undergo and the possible sets of events it can refuse at any time [Smi92].

2. CSP provides a unique facility for specifying allowable event traces formally and implicitly. The mathematical notation provides representation and reasoning facilities for traces, abstracts them into lists and their members to set elements. Such a facility for formally specifying the properties which a system must satisfy does not exist in CCS.

3. In CCS there is a special operator $\tau$ that denotes an internal communication event (i.e., the occurrence of a hidden event or an internal transition). There is no

equivalent symbol in CSP. The behaviour of a process in CSP is described in terms of externally observable events drawn from its alphabets. In CCS, both internal and externally observable events describe the behaviour of a process.

4. CSP has no notion of a communication's dual (complementary communication). In CSP, it is identical communication that synchronize rather than complementary ones. In CCS, there is the concept of communication and its dual (for example, the dual of $\alpha$ is $\overline{\alpha}$).

5. There is a single choice operator, +, in CCS, whereas there are two in CSP: the deterministic $\Box$ and nondeterministic $\sqcap$ choice operators. In CCS, there is no special operator for specifying non-determinism but it can be modelled by using the $\tau$ operator. The CCS operator can achieve effects close to each of the two CSP ones because the semantics of CCS is sensitive to $\tau$'s (internal) actions. In CSP, the equivalent of the process $\tau \cdot P$ is indistinguishable (in any context) from P.

6. The treatment of concurrency differs in both languages. In CCS, concurrency is represented by | while in CSP concurrency is defined by || for synchronization and ||| for interleaving. This separation of the concurrency concept into the synchronizable parallel processes and interleavable processes makes reasoning in CSP much easier. Therefore, the semantics of the CCS operator is much more complex because it includes aspects of hiding, non-determinism, interleaving and synchronization.

7. Other notable differences between CSP and CCS are [FS93, CK92]:

   - When CSP process communication events synchronize they are not automatically hidden (becoming $\tau$-events). This means that more than two processes can synchronize on a single event and that synchronized events can be observed from the outside. This gives rather more freedom to use CSP as a specification language and allows the parallel operator to act rather like logical conjunctions on behaviours.
   - Since synchronization does not automatically conceal an event, CSP requires a hiding operator (\) to do this explicitly. This looks like the CCS restriction

operator and appears in similar places in programs, but it is semantically different. For example, the CSP process

$$(sit \longrightarrow run \longrightarrow walk \longrightarrow run \longrightarrow crawl \longrightarrow STOP)\backslash run$$
$$\Rightarrow\ sit \longrightarrow walk \longrightarrow crawl \longrightarrow STOP$$

Its CCS equivalent is:

$$(sit\ .\ run\ .\ walk\ .\ run\ .\ crawl\ .\ \mathbf{nil})\backslash run\ \Rightarrow\ sit\ .\ \mathbf{nil}$$

The CCS (\) operator stops both the action indicated on the right side of the operation and all the actions following this action.

- In CCS, the interface between a pair of processes is enabled by a combination of their ability to synchronize with each other and the use of the restriction operator (e.g., $\backslash\alpha, \overline{\alpha}$) to forbid them from using the same events elsewhere. In CSP, interfaces are defined either by means of process alphabets or as a parameter of the parallel operator. It is logically impossible for any process to engage in any event outside its alphabet.

The choice of CSP over CCS as the formal specification language for transactions models is because it has *explicit* constructs to capture deterministic and non-deterministic choices, parallel composition, interleaving, synchronization, and other transaction systems properties. Each CSP specifiable subsystem can be implemented as an independent module because of the bottom-up design approach it supports. In addition, CSP has resource primitives *ACQUIRE* and *RELEASE*. This is a form of locking. When more than one resource requires sharing in this form, the possibility of deadlock exists. To prevent deadlock occurrence, these primitives adopt a form of the two-phase locking protocol. The complex and abstract nature of CCS notations make it difficult to specify and reason accurately in the language so it is less attractive.

## 2.4 Traditional Transactions

Traditional transactions [Gra81, EGLT76] provide failure atomicity, consistency, isolation, and durability which are collectively referred to as the ACID properties. Chrysan-

this [Chr91] argues that the transaction model adopted in traditional database systems is inadequate for new complex applications. Therefore, both types are reviewed. For a detailed discussion of these concepts refer to [BGH87, Pap86, Des90].

### Notations

The following notations apply. Let $O$ denotes an operation, $O_i(x)$ denotes an operation $O_i$ on a data item $x$, $T$ denotes a transaction, the index of a transaction, as in $T_i$, denotes the $i^{th}$ transaction while the index of an operation, as in $O_i$, denotes the $i^{th}$ operation. An operation $O \in \{r(x), w(x)\}$ where $r(x)$ is a *read* operation and $w(x)$ is a *write* operation while $x$ is an arbitrary data item. Also, let $OT_i$ denotes a set of operations invoked by a transaction $T_i$ and $E$ denotes the transaction termination operation where $E \in \{a, c\}$ where $a$ and $c$ stands for *abort* and *commit* operations, respectively.

Further, let $\prec$ denotes the *happens-before* relation[5] [Lam78]. An operation $O_i$ happens-before operation $O_j$, written $O_i \prec O_j$, if $O_i$ precedes $O_j$ at execution time. Further, operations which conflict are ordered by $\prec$. Two operations *conflict* when they both access the same data (at least one of them is a write operation) and the operations give different results if their relative order is changed. For any two operations $O_i, O_j \in OT_k$ which conflict, then either $O_i \prec O_j$ or $O_j \prec O_i$ occurs. Therefore, executing multiple concurrent transactions requires paying particular attention to the ordering of their operations to guarantee correctness.

### Transactions

The execution of a transaction $T$ is a partial order[6] of events with ordering relations $\prec_T$, where $\prec_T$ denotes the temporal order in which the related events invoked by $T$ occur.

**Definition 4** *Transaction:* A transaction $T_i$ is a partial order $T_i = (\sum_i, \prec_i)$ where

1. $\sum_i = OT_i \cup \{E_i\}$ ,

[5] $\prec$ is an irreflexive and transitive binary relation that indicates the execution order of the operations involved.

[6] It is a partial order since some of the operations of a transaction may be executed in "parallel".

2. For every $O_i, O_j \in OT_i$ and $i \neq j$, if $O_i$ and $O_j$ conflict then either $O_i \prec_i O_j$ or $O_j \prec_i O_i$ , and
3. $\forall O_k \in OT_i$, $O_k \prec_i E_i$ ■

This states that the execution of $T_i$ contains all operations of $T_i$ and all operations which conflict are ordered by the $\prec$ relation. Finally, property (3) ensures that no operation of a transaction will execute after the transaction terminates.

### Histories

A *history* is a record of transaction executions. Usually, in a system, multiple transactions may execute concurrently. Since conflict can occur between any two operations, the relative order of the execution becomes significant[7]. A history includes at least the ordering relation of all conflicting operations.

**Definition 5** *History:* Given a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ executed concurrently, a history is a partial order $H = (\sum, \prec)$ where

1. $\sum = \bigcup_{j=1}^{n} \sum_j$ where $\sum_j$ is the domain of transaction $T_j \in T$,
2. $\prec_H \supseteq \bigcup_{j=1}^{n} \prec_j$ where $\prec_j$ is the ordering relation at the database management system level, and
3. for any two conflicting operations $p, q \in H$, either $p \prec_H q$ or $q \prec_H p$. ■

All operations of $H$ submitted by the transaction set $T$ obey orderings within each $T_j$ and orders all conflicting operations of all transactions.

### Correctness Criterion

A correctness criterion specifies properties that guarantees database integrity. The most popular criterion is *serializability.* A concurrent execution of transaction set is serializable if it is equivalent to some *serial* execution. Various forms of serializability appear in the literature. The most common forms are conflict equivalence and view equivalence leading to *conflict* and *view* serializability [BGH87, Pap86], respectively.

An important tool for checking serializability of a history is the *serialization graph.*

[7]The result of a concurrent execution of transactions depends on the relative ordering of conflicting operations.

**Definition 6** *Serialization graph:* A serialization graph (denoted SG(H)) is a digraph $SG(H) = (V, E)$ with a vertex $v \in V$ for each committed transaction in $H$ and an arc $e \in E$ from $T_i \longrightarrow T_j$ if and only if an operation of $T_i$ precedes and conflicts with an operation of $T_j$ in $H$. ■

A history is serializable if and only if SG(H) is *acyclic* (see [BGH87] for proof). The serialization graph is constructed with committed transactions only.

Other types of serializability are view and final-state serializability [BGH87, Pap86]. These allow more concurrency than conflict serializability and they have different semantics. While a conflict serializable history must be conflict equivalent to a serial history; a view serializable history ensures each transaction sees the same data as it would in some serial execution. Similarly, *final-state* serializability allows concurrent transaction execution such that the final state of the database is equivalent to the result of some serial execution of the transactions. Checking any two histories for equivalence uses the *read from* and *final write* relations instead of conflicting operations used for conflict serializability. The general problem of determining if schedules are state or view serializable is NP-complete [Cla92, Pap86] so only conflict serializability permits efficient implementation.

### 2.4.1 Recovery

Since failure may leave the database in an invalid or erroneous state[8], recovery[9] mechanisms are required to bring the database into a consistent state by ensuring that no intermediate results of aborted transactions remain in the database while the effects of all committed transactions do. The aim of recovery mechanisms is to allow the resumption of database operations after the occurrence of a failure with minimum loss of data and at an economical cost. Since recovery and concurrency are interwoven, many decisions about recovery often influence concurrency and vice versa.

A non-serial history may not always be recoverable. To create a recoverable history, the relative order of some operations within a history must be defined. A transaction is

[8]An error in the system occurs when the system assumes a state that is undesirable.

[9]Recovery is the ability to recover from failures (system, media, or transaction).

*recoverable* if all other transactions it reads from commit before its commitment. Thus,

**Definition 7** *Recoverable:* A history $H$ is recoverable ($RC$) if when $T_i$ reads $x$ from $T_j$ (where $i \neq j$) in $H$ and $c_i \in H, c_j \prec c_i$. ■

This definition prevents transaction $T_i$ from reading a value produced by transaction $T_j$ and then commits while $T_j$ is still active. If $T_j$ eventually aborts then $T_i$ has modified the database based on inconsistent values produced by $T_j$.

When a transaction can only read values that have been produced by committed transactions, a stronger recoverability condition exists. A transaction that obeys this condition does not cause other transactions to abort if it aborts. Hence they *avoid cascading aborts* ($ACA$).

**Definition 8** *Avoids cascading aborts:* A history $H$ avoids cascading aborts ($ACA$) if whenever $T_i$ reads $x$ from $T_j$ (where $i \neq j$) in $H$, and $c_j \in H, c_j \prec_H r_i(x)$. ■

ACA property prevents a transaction from reading values produced by active transactions but the it can reads only values written by itself or committed transactions.

A transaction is *strict* ($ST$) if it cannot read or overwrite a data item until the commitment of the transaction that wrote the previous value. Formally:

**Definition 9** *Strict:* A history is strict ($ST$) if whenever $w_j(x) \prec_H O_i(x), (i \neq j)$, either $a_j \prec_H O_i(x)$ or $c_j \prec_H O_i(x)$. ■

Therefore, no data item may be read or overwritten until the transactions that wrote it terminates.

Bernstein *et al.* [BGH87] have shown that $RC \supset ACA \supset ST$, $FSR \supset VSR \supset CSR \supset SR$, and characterised the interrelationships between recoverable and serializable classes.

### 2.4.2 Types of Transaction Concurrency

Two types of transaction concurrency, intra- and inter- transaction concurrency are possible (and therefore two types of concurrency control) in any transaction system. *Intra-transaction concurrency* occurs within a transaction while *inter-transaction concurrency* arise between different transactions executing concurrently.

Intra-transaction concurrency can arise in a variety of scenarios such as between subtransactions alone or between at least one subtransaction and its parent transaction. In intra-transaction concurrency control, while preserving local dependencies, non-conflicting operations do not require synchronization so maximal concurrency is achievable.

However, non-local conflicts may occur because of references to shared data so a concurrency control protocol is necessary to resolve the conflicting operations. To resolve conflicting operations, serialization orders for operations must be defined. In other words, transaction's subtransactions executed concurrently must be equivalent to some serial execution. The advantage of intra-transaction concurrency control is that greater potential concurrency is achievable while providing finer granularity for recovery in cases of failures with in the transaction.

Inter-transaction concurrency control involves concurrency arising between different user transactions executing concurrently. This is required in all multi-user systems in either traditional or complex transaction models. The concurrent execution of different user transactions must be serializable to guarantee the database's correctness.

### 2.4.3 Approaches to Concurrency Control

Concurrency control schemes can be classified as: *pessimistic* and *optimistic*. Both ensure correctness in centralized or distributed environments.

Optimistic strategies permit restricted concurrent transaction execution where results validation occurs at commit. Thus, validation occurs after allowing (possibly incorrect) concurrent transaction executions. The complexity of recovery in cases of failures is tremendous because this approach allows incorrect executions. Therefore, to safely undo the effects of partial results requires a sophisticated recovery mechanism. Examples of the optimistic concurrency control approach are versioning algorithms and optimistic timestamp ordering.

Pessimistic concurrency control ensures only correct operations sequences execute. This may reduce concurrency but often simplifies recovery. Examples include two-phase locking and timestamp ordering protocols [BG89]. These are discussed in details shortly.

Concurrency control also adopts methods based on *operation semantics.* Examples of this type of concurrency control are commutativity [Wei88] and recoverability [BR90, BR91]. Two operations commute if their effects are independent of the order in which they are executed[10]. Recoverability is a weaker definition of conflicts than commutativity. An operation Q is recoverable relative to another operation P, if Q returns the same value whether or not P is executed immediately before Q. Transactions invoking operations P and Q commit in their invocation order which defines commit order. When used with locking based protocols, recoverability like commutativity, avoids cascading aborts while also avoiding the delay in processing of many non-commutative operations. Recoverability assumes a flexible recovery technique for handling the abortion of recoverable operations.

### The Two-Phase Locking Protocol

A *lock* is a variable associated with a data item that controls access. Thus, a data item is conceptually a pair $\langle value, lock \rangle$. Therefore, a reference to a data item $x$ implicitly manipulates $x(v, l)$ where $v$ is the current value (i.e., information content of $x$) and $l$ is the current lock mode.

Two locks types are used, namely: *exclusive* and *shared.* Exclusive locks (also called *write* or *update* locks) provide access to the data item for only **one** transaction. Shared locks can be held by an arbitrary number of transactions for read access. Transactions that both read and write a data item acquire an exclusive lock.

Serializability is ensured if all locking operations precede the first unlock operation in the transaction. This is called the *two-phase* locking protocol (2PL). Thus, the two-phase locking protocol is characterised by a growing phase followed by a shrinking phase. The locking and unlocking operations are monotonic increasing and decreasing, respectively. Once a lock is released, additional acquisition of locks is forbidden.

The 2PL protocol guarantees serializable schedules [Des90, EN94, BGH87]. Thus if every transaction in a schedule obeys the 2PL protocol the schedule is guaranteed to

[10]Commutativity must provide the semantics for determining if any two operations can execute concurrently.

be serializable. The order of transactions in the equivalent serial schedule depends on the order in which executing transactions lock the items they require.

Although the 2PL protocol gives the best performance for most database applications [BHG87, BK91, ACL87, CGM91, GR93] and despite guaranteeing serializable schedules, 2PL has two serious limitations — *deadlock* and *livelock*. Deadlock arises when each of two or more transactions is waiting for the other to release the lock on an item. Thus, a set of transactions $T = \{T_1, T_2 \cdots, T_n\}$ are in deadlock over a set of resources $R = \{R_1, R_2 \cdots, R_n\}$ at time $t$ if and only if $\forall\, i : 1 \leq i \leq n$, process $T_i$ wants to access resource $R_i$ and $R_i$ is held by process $T_{i+1}$ at time $t$ and $T_n$ is waiting for $R_n$ held by $T_1$. A transaction is in a state of livelock[11] if it cannot proceed for an indefinite period while other transactions in the system continue normally. Unnecessary restarts and frequent waiting can severely degrade the performance of the system [EN94, LR91, SRL88]. Several algorithms have been proposed[12] to deal with deadlock.

Variants of the basic 2PL algorithm are the *conservative* 2PL (C2-PL) and *strict* 2PL (S2-PL). The C2-PL algorithm must lock all its items before it starts, whereas S2-PL does not unlock any of its items until after it terminates by committing or aborting. C2-PL requires a transaction to lock all the items it accesses before the transaction begins execution. If any of the item is unlockable, the transaction does not lock any item; but waits until all the items are available for locking. It is a deadlock free protocol that limits concurrency. In S2-PL, a transaction $T$ does not release any of its locks until after it commits or aborts. Therefore, no other transaction can read or write an item written by T unless T has committed, leading to a strict schedule for recovery. S2-PL is not deadlock free unless used in combination with C2-PL. By combining S2-PL and C2-PL together yields recoverable and serializable schedules that depend on the lock acquisition order of the transactions but it further constrains the degree of concurrency.

[11] Livelock situations happen if the waiting scheme for locked item is unfair. Unfairness arises by giving priority to some transactions over others. The solution to this is to have a fair waiting scheme such as first-come first-serve (FCFS).

[12] Interested readers are referred to [Des90, EN94, BHG87, BL93, Cla92, HZ92, GR93] and other sources for details. Some of the algorithms are: *wait-die*, *wound-wait*, *time out*, *no-waiting*, and *cautious waiting*.

### Timestamp Ordering

Timestamp Ordering (TO) achieves concurrency without locking so it eliminates deadlock. Timestamp values are assigned to transactions when they are submitted (that is, the transaction start time). Each transaction is assigned a unique non-decreasing number. All of a transaction's operations have the same transaction timestamp. Timestamps are generated with a counter or the system clock. The timestamps are monotonically increasing.

A schedule in which the transactions participate is serializable [BGH87] and the equivalent serial schedule of the transactions corresponds to the order of the transaction timestamps. The system enforces serializability using the chronological order of the concurrent transactions' timestamps. The scheduler immediately schedules any operation that arrives for execution unless a conflicting operation with a higher timestamp has executed. A conflict occurs when an older transaction tries to read a value written by a younger transaction or when an older transaction tries to modify a value already read or written by a younger transaction. Both attempts signify that the transaction with lower timestamp was "too late" so it may see stale values [Des90]. Conflicting operations from distinct transactions get scheduled (or aborted) based on their timestamp values. The TO protocol checks conflicting operation occurrences that arrive in the wrong order and rejects those with lower timestamps by aborting them.

The basic TO protocol enforces conflict serializability but does not ensure recoverable schedules. Therefore, it does not avoid cascading aborts or produce strict schedules. Another related problem is cyclic restart; so starvation may occur if transactions abort and restart continuously.

The Strict TO ensures that schedules are both strict and conflict serializable. A transaction $T$ that issues a read or write operation on $x$ such that the timestamp of $T$ is greater than the timestamp of $x$'s last write has its read or write operation delayed until the transaction, say $P$, which wrote the value $x$ commits. This protocol is deadlock free because transaction $T$ waits for $P$ only if $T$'s timestamp is greater than $P$'s timestamp.

### 2.4.4 Recovery Techniques

There is a need to recover data in the presence of a transaction's abort. Erroneous states are resolved using exception handling mechanisms. Failure management by exception handling may involve one or more of the following alternate courses of action: (1) abandon the execution of the unit, (2) try the operation again, (3) use an alternative approach, or (4) repair the cause of the error.

Two methods generally used to recover from failures are: (a) the logging approach [HR83] which maintains versions of the data by keeping a record of the before-write value and the associated transaction, or (b) keeping a record of the inverse operations required to restore the value [Mos85]. The two approaches differs in that logging shows **what** changed while inverse operations shows the **how** of the change. Logging is the most popular recovery technique. The log is stored in non-volatile memory. When a failure occurs, the log information becomes available to undo the effects of aborted and active transactions[13] and to redo the committed transactions [Des90]. To implement the abort operation using logging, the before-write images of all writes of a transaction are restored except where the data are modified by another transaction, say $T_k$, after it was last written by the transaction.

Other recovery mechanisms adopt *contingency transactions* [BHMC90] which execute an alternate transaction when the original transaction fails, and *compensating transactions* [GS87] which amends partial executions by invoking operations to annul the committed actions of the failed transaction.

A contingency transaction executes upon failure of the transaction as an alternative to the failed transaction. A contingency transaction may abort like any other transaction since it is a transaction in its own right. Contingency transactions accomplish *forward recovery*. Compensating transactions achieve *backward recovery* from failure of a transaction. They compensate for the partial execution of the failed transaction's operations. Compensation order is the inverse of the commit order of the aborted transactions. In other words, compensating transactions provide a mechanism to undo the effects of com-

[13]An active transaction is executing transaction in progress. An active transaction either aborts or commits.

mitted partial executions thereby defining logical equivalences of rollback.

## 2.5 Extended Transactions

"Extended transactions" is a broad class of transactions models that flexibly interpret the ACID properties in order to deal with transactions in new applications. An *extended transaction* consists of a set of operations on data objects that execute atomically in a predefined order or a set of extended transactions with an explicitly given control related to the notions of visibility, consistency, recovery, and permanence [Chr91].

Extended transactions exhibit a rich and complex internal structure. Their component transactions are not necessarily atomic. The way component transactions combine to form extended transactions reflects the semantics of the applications. Some of these models assume or dictate specific recovery or commit protocols. Therefore, their effects on transaction-specific concurrency and recovery protocols are of particular interest in this research.

A transaction accesses and manipulates database's data objects by invoking operations specific to individual data objects. Depending on the semantics of the operations, aborting an operation may also force abortion of some other operations thereby leading to cascading aborts of operations. Thus, specifying concurrency uses the transaction's semantics and the data they manipulate since this approach provides a means for dealing with the application's functionalities.

Dealing with concurrency in new applications use either *ad hoc* systems or traditional concurrency control [Chr91]. Ad hoc systems allow cooperation among users based on an intuitive model of interactions. These systems lack formal definitions and as such their correctness cannot be characterised mathematically. Similarly, traditional concurrency control based on the concept of serializability are inadequate for long- duration transactions.

The following discussion provides a summary of some extended transaction models. The discussion focuses on their concurrency control and recovery properties.

### 2.5.1 Active Databases

The traditional transaction model has been extended to a flexible execution model with rule processing. Chakravarthy *et al* [Cha89] discuss the usefulness of a rule-based active capability for supporting integrity enforcement, access control, and dynamic coordination. The use of rules for incorporating active capability provides declarative specification of events, conditions, and actions for capturing object and application semantics. The model has the capability to monitor temporal events and initiate a set of actions.

An event algebra [CM91, Mis91] forms part of the model for supporting different event types specification and constructing complex events. For example, [GJS92] describes the integration of composite event specification. The event specification use a set of notations identical in expressive power to a notation based on regular expressions. The active paradigm has been incorporated into object oriented databases [Anw92, GJ91, GJS92] and used to enhance concurrency in multidatabase environments [Bak95]. Korth and Speegle [KS94], however, argue that event-condition-action rules lack the concept of correctness thereby making it impossible to detect if an execution is correct.

### 2.5.2 Nested Transaction

Nested transactions [Mos85, Pu86] are hierarchically structured transactions that support computations similar to those in procedure calls. Transactions consist of subtransactions designed to localize failure within such a transaction and to exploit concurrency between subtransactions. A subtransaction may contain read and write steps and other subtransaction invocations thereby changing a transaction from a sequence of steps into a hierarchy of subtransactions. Therefore, each node is either a subtransaction (*non-leaf transaction*), a database access (*leaf transaction*) such as read or write step, or a combination of both[14]. Thus, a nested transaction model supports two types of transactions: root transaction and subtransactions. The root of the hierarchy is used to model the execution of the transaction.

[14]This view of nested transactions differs from Moss's [Mos85] where internal nodes do not perform any database access operation.

A nested transaction can handle partial failures and subtransactions can abort independently without causing the transaction to abort. Subtransactions execute atomically with respect to their siblings and other non-related (independent) transactions and failure atomic with respect to their parents. Subtransactions can potentially access any data object currently accessed by one of its ancestor transactions or any other database's data object not currently accessible to any of its descendant subtransactions.

When a subtransaction commits, the objects it modifies become accessible to its siblings and parent transactions. However, the effects of the updates on data objects become permanent in the database when its root transaction commits. If a committed subtransaction's parent aborts, then the subtransaction aborts too. A non-leaf transaction performs only the *begin transaction* operation that starts subtransaction's execution and updates the partial order of the transaction. When a subtransaction commits, the parent inherits the subtransaction's accessed data objects; adding it to those of all its committed subtransactions. The parent also inherits the read set accumulated by the subtransaction. A committed non-leaf transaction can abort if its parent changes any element in its read set. Also, if a non-leaf transaction aborts then all its subtransactions must also abort.

A degenerate form of the NT model is the distributed transactions [Chr91, GR93]. In a distributed transaction model, a transaction is broken down into subtransactions that invoke operations on data objects distributed physically in a network. Each subtransaction may be nested. However, there are some subtle differences between the two models; for example, abortion of a subtransaction causes the abortion of the whole distributed transaction.

The preceding discourse is often called *closed nesting.* The characteristic feature of close nested transactions is the bottom-up to the root commitment approach, the semantics of which enforces atomicity at the topmost level.

Another type of nested transaction is *open nesting.* Open nested transactions have different termination characteristics by relaxing the top-level atomicity of close nested transactions. An open nested transaction makes its partial results visible outside the transaction. Therefore, logging may be inappropriate for recovery because effects of

committed transactions are durable. To undo the effects of partial results requires the use of compensating transactions.

Open nested transactions are appropriate only if it possible to define a suitable compensating transaction for each top-level transaction. A compensating transaction may never start unless the corresponding transaction associated with it previously commit [BOH+92]. Therefore, there is a begin-dependency between a top transaction and its compensating transaction.

A Saga [GS87, GGKKS90] is an example of open nested transactions. A Saga [GS87] is a two level nested transaction with traditional transactions as its components. Each component transaction is associated with an application specific compensating transaction. It can interleave in any way with other Sagas but it cannot partially execute. If a Sagas is interrupted, it either attempts to proceed by executing the missing transaction (forward recovery) or amends partial executions by invoking compensating transactions (backward recovery).

A variation of Sagas [see GGKKS90] allows the characterization of subtransactions as *vital* or *non-vital*. The abortion of a vital subtransaction causes the abort of the transaction. Thus, a vital transaction must execute successfully for its parent transaction to commit. A parent transaction may commit even if one of its non-vital transactions aborts.

### 2.5.3 Cooperative Transactions

A cooperating transaction [KKB88, KS88] is a subtransaction that does a specific task for another user. In cooperative transactions, partial changes made by one transaction may be visible to another while they are executing. Therefore, components of cooperative transaction may not produce consistent results. The effects of a transaction set are considered consistent if all the steps that express the required computational goals execute completely and the execution steps interleave correctly.

This model uses *predicatewise serializability* correctness criterion which permits non-serializable executions while satisfying the individual postconditions of the transactions. To control the interleaving of concurrent operations requires the specification of

conflict among the operations.

An application that adopts this model is an airline reservation system that involves subtransactions which require independence between reservations but depend on the entire transaction completing.

### 2.5.4 Federated Databases

A federated database system (FDBS) [SL90] is a collection of cooperating database systems that are autonomous and possibly heterogeneous. Other terms synonymously used to describe FDBS are *multidatabases*[15] (MDBS) [LMR90] and *heterogeneous databases*[16] (HDB) [EC90]. A key characteristic of a federated database system is the *cooperation* among independent systems. Components DBSs allow partial and controlled sharing of their data. A component database system can continue to execute its local operations without interference from external operations and it determines the order in which to execute external operations. It treats external operations in the same way as its local operations. This implies that a component DBMS can abort any operation that does not meet its local constraints and that its local operations are logically unaffected by its participation in an FDBS. Also, the component DBMS does not need to inform an external system of the *order* in which it executes external operations or vice versa.

Global operations involve data access using the FDBMS and may involve data managed by multiple component DBSs. Component DBSs must grant permission to access the data they manage. Local operations are submitted to a component DBS directly. A component DBS, however, does not need to distinguish between local and global operations. The classical requirements for consistency, concurrency control, and transaction management require a redefinition for federated database environment since a transaction may span several autonomous database systems.

A serious problem in the FDB model is the maintenance of global serializability, due to the difficulty of detecting conflicts between global and local transactions, since

[15]Multiple autonomous databases managed together without a global schema are called mutidatabases.

[16]Heterogenous database is a collection of heterogenous cooperating databases loosely coupled together.

each may be running under different concurrency control algorithms. Another problem is the preservation of autonomy. *Local autonomy* is the ability of each component database to control access to its data by other component databases and the ability to access and manipulate its own data independent of the other participating component databases. The former is desirable for performance and security while the latter allow local users to access their own data without external interference thereby allowing possible maximal concurrency between local operations and external operations.

## 2.6 Dependence Relationships

Besides begin dependency, other dependency relationships exist between subtransactions and parent transaction. These are:

- Abort dependency between the subtransaction and its parent. Abort dependency does not allow a subtransaction to execute if the parent transaction aborts.
- Termination dependency of the parent transaction on its subtransactions requires that the parent transaction cannot terminate until all of its subtransactions terminate.
- Force-commit-on-abort-dependency requires a transaction, say T2, to commit if another transaction, say T1, aborts. For example, compensating transactions must commit on the abortion of the transaction needing compensation.

Other types of dependencies generally found in extended transaction models are described in Chrysanthis's seminal work [Chr91].

# Chapter 3

# SPECIFYING TRANSACTIONS

This chapter begins by providing a taxonomy of any database transaction specification's desirable characteristics in Section 3.1. Section 3.2 discusses the requirements of database transaction specification formalism while Section 3.3 highlights some important issues like constraints specification, proof requirement, time property, and causality that demands particular considerations when specifying database transaction systems. Finally, Section 3.4 briefly discusses why CSP is our language of choice for presenting a sample specification and proofs in a practical example, the Electronic Shopping Mall, to demonstrate important results (See Chapter 5).

## 3.1 Taxonomy of Transactions Specification Requirements

This section presents the desirable properties of transactions specification and a taxonomy of these properties in order to explain the need for certain required language features (outlined in Section 3.2). Figure 3.1 illustrates the relationship between transaction features, desirable specification characteristics, the features of any transaction specifications formalism, and transaction correctness formal specification. The figure provides the context of the three main entities (features of transactions, desirable specification characteristics, and features of transaction specification formalism) necessary

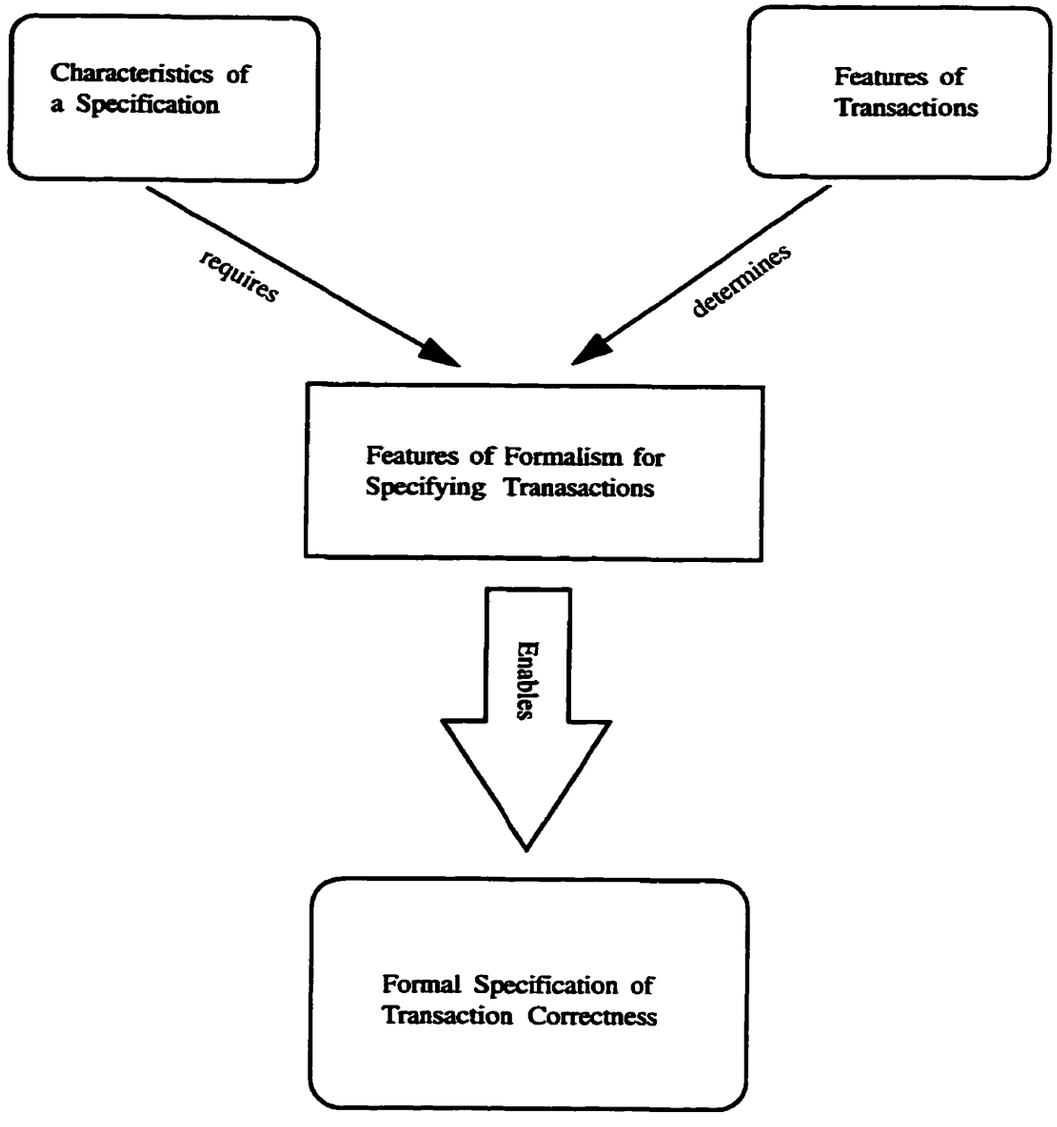


Figure 3.1: Transactions, Specification, and Formalism Features Relationship

to adequately specify transaction systems and their correctness. The main intent is to eventually espouse the requirements for *any* formalism that specifies transaction systems.

In this thesis, the characteristics of a transaction's specification are classified into three taxonomic dimensions: *correctness, confirmability,* and *completeness.* These dimensions are the *minimal*[1] requirements because any deviation impacts system dependability. The dimensions are interrelated; resolving an issue in one may transcend the other. Each dimension contributes uniquely to the specification's overall quality and utility. None of the dimensions can be ignored and hope to compensate for it in the other. Compromising any of the dimensions produces a specification that does not meet the system's desired needs. By understanding each dimension, writing specifications that exhibit these characteristics significantly increases their quality and dependability. Figure 3.2 illustrates these dimensions and their constituents.

**Completeness Dimension:**

The *completeness dimension* ensures that all essential[2] characteristics, functionalities and behaviours are modelled. Completeness dimension consists of the completeness and functional characteristics. *Completeness* is the property whereby a specification contains all essential characteristics and properties. Thus, the specification contains all significant requirement, whether relating to functionality, performance, design constraints, and interfaces. Completeness has two aspects: *conceptual* and *syntactic.* Conceptual completeness implies that the specification contains adequate functions and objects to describe the scope of the domain the specification describes. Syntactic completeness implies that the specification captures the conceptual objects/functions in the applicable specification language. A specification is *functional* if it describes only **what** the system

[1]The *conciseness* property can be ignored without any significant effect on the system's dependability.

[2]Essential requirements are those considered necessary to meet the system's operational needs, independent of any later technology advances or the specific technology used to implement the system. They may define functions that the system must support (i.e., its behaviour), nonfunctional properties of the finished system (e.g., responsiveness), or external interfaces or restrictions (e.g., communication with an external device). Changes in technological advances may later modify the system's perceived needs. This is one of the causes of a system's modification.

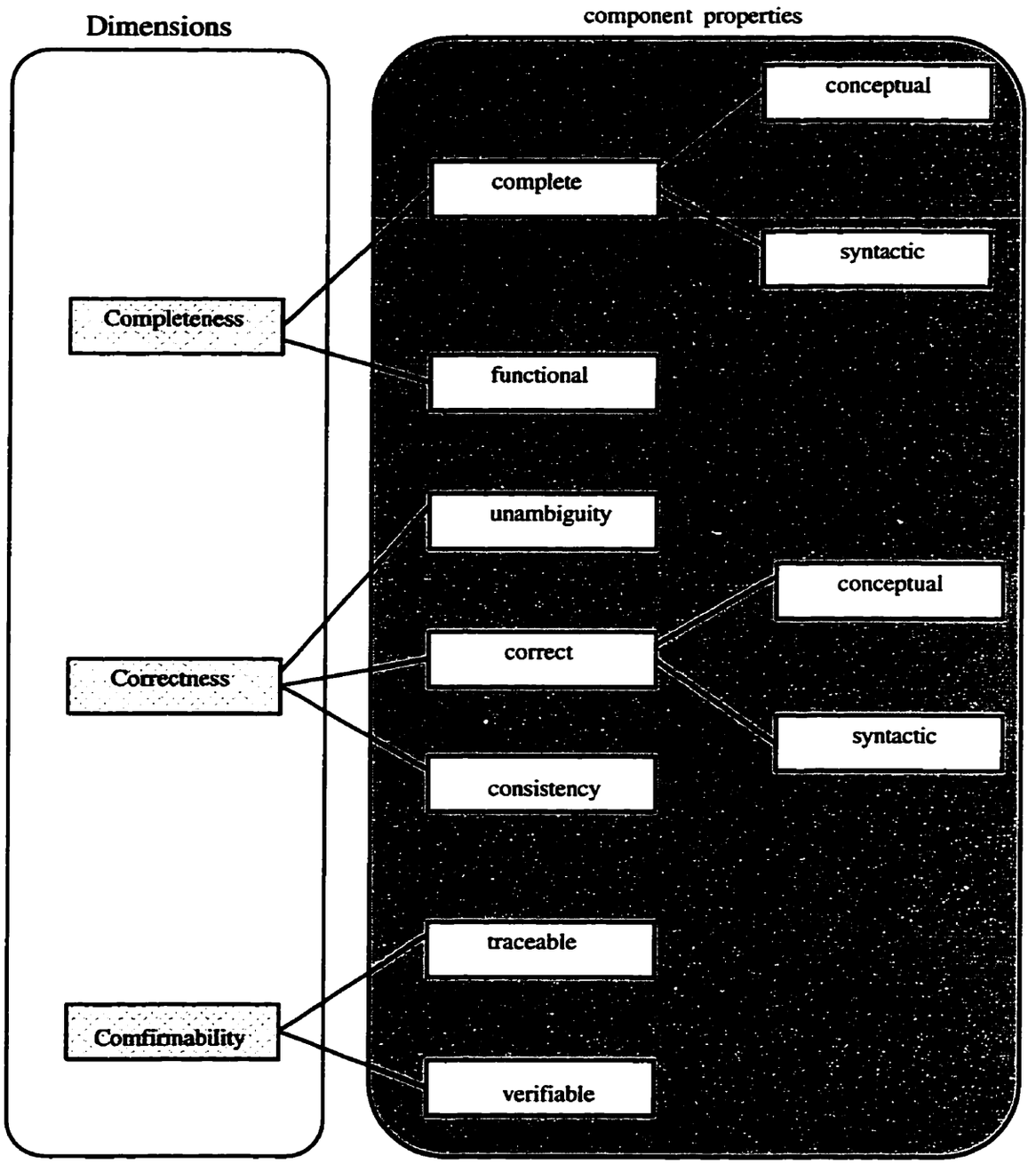


Figure 3.2: Taxonomy of Transaction Specification Characteristics

will do. The functional specification defines the operations and transformations that the system must carry out. Each functional requirement is specified in detail. In addition, functional requirements include a description of the set of legal values and ranges that the system will accept for input, the state changes and actions that the system will take on both legal and illegal input, and the output the user will see. Functional specification does not include any design or implementation features but only the system's desired functions.

The completeness dimension guarantees provision of all appropriate services, messages and operations, defines all necessary data attributes, handles and raises all appropriate exceptions, and defines what happens for all possible input[3] to the system. Every attribute must have a domain of permitted values[4]. Proper use of the completeness principle provides reusability of objects at higher level, increases productivity, and decreases configuration management. It should be emphasized that completeness as used here does not connote *absolute* completeness, which is usually difficult to attain, rather the specification should contain all functions and modules to provide the services required of the system. In other words, a specification should contain all essential requirements that completely describe the structure and behaviour of the transaction model it specifies.

Checking a specification for completeness involves: (1) componential analysis of the syntactic, semantic, and conceptual elements of the specification, (2) induction (and invariants), and (3) checking for omissions. [ABL89] provides a pragmatic checklist for checking a specification's completeness. These are:

- Are all sources of input identified?
- What is the total input space?
- Are there any timing constraints on the inputs?
- Are all types of outputs specified?
- What are all the types of runs?
- What is the input space and output space for each type of run?

[3]Software responses are defined for all realizable classes of input (including invalid inputs).

[4]The selected domain must represent attribute values in a meaningful manner.

- Is the invocation mechanism for each run type defined?
- Are all environmental constraints defined?
- Are all necessary performance requirements defined?

**Correctness Dimension:**

The *correctness dimension* ensures the specification captures accurately the desired system properties. The correctness dimension consists of the consistency, correctness, and unambiguity properties. A specification is *consistent* if and only if there are no conflicts between the requirements and it is *unambiguous* if and only if every property (requirement) has only one interpretation. Common examples of contradictions are using different terms for the same action and two requirements that contradict each other. A specification's *correctness* is the accuracy to which it describes exactly the tasks and properties and to which requirements meet their needs. In other words, a specification correctness measures its consistency with respect to its requirements (needs). Correctness has two parts: *conceptual correctness* and *syntactic correctness.* Conceptual correctness implies that the specification accurately captures/reflects the concepts of the transaction model. It depends on the specifiers' ability to translate the transaction environment into a semantic language to form a *meaningful* and *accurate* representation of transaction models. The translation from concept into formal form *must* be correct. Verifying conceptual correctness is a challenging and difficult exercise but involving *domain experts*[5] both during the specification's capture and verification helps significantly. Syntactic correctness implies that the specification adheres to the syntax of the specification language.

Correctness checks ensures there are no logical, syntactic, and semantic, and conceptual mistakes in the specification. Consistency checking checks for conflicts (same conditions but different actions), redundancies, subsumptions, inconsistencies (e.g., $A \Rightarrow B$ and $A \Rightarrow \neg B$ ), and ambiguities in a specification. Further, consistency checks ensure no input states are mapped to more than one output state and a consistent set of

[5]Domain experts are specialists in a particular application domain of interest. For example, transaction systems management specialists are domain experts when building a distributed database application software.

quantitative units are used.

With a complete and consistent formal specification (i.e., a precise definition that allows recognition of inconsistencies and preserves correctness), every operation / interpretation is verifiably correct or incorrect.

### Confirmability Dimension:

The *confirmability dimension* determines if the specification is correct. It ensures that the correct models are being built. Confirmability consists of *traceability* and *verifiability* characteristics. *Traceability* is the ability to track every property in the system's specification. Every property should be traceable to specific system's requirements. Requirements may have one of several sources, depending on the nature of the specification activity. Traceability is both *forward* and *backward.* Forward traceability is a mapping from the essential requirements entities to the resulting design entities. Thus, the requirements are "linked to the system's operational needs via specific verification from the detailed requirements to the operational needs" [KS92]. Backward traceability occurs if each reference requirement is uniquely identified so that one can map properties of the delivered product into requirements of the specification, and vice versa. Traceability may be attained through: (a) reference to the source of the need or reference to a physical data structure element or variable, and (b) mapping to another metadata object created and managed as part of the development. Every attribute is either primary or derived[6] so the specification must document the source of the data and where it will be used. A requirements traceability matrix [KS92] must be provided to ensure completeness and consistency with the customer's needs. Automated tools often help to generate much of the traceability information. Traceability in turn supports verifiability. Verification is the exercise of determining correctness. A specification must be verifiable to ensure that the specification itself is correct and that transaction systems built from it behave correctly. A specification is *verifiable* if it is possible to show that every property stated

[6]An attribute is derived if its value is generated from one or more other attributes in the model according to a specified formula. An attribute is primary if it is not derived within the scope of the model.

in it holds in the system. Thus, proving and verifying every property stated in, and tracing the properties through, the specification are the basic functions of confirmability. Therefore, confirmability provides mechanisms to check the correctness, consistency, and completeness of a specification.

**Other Properties:**

In addition to the above dimensions, a specification should be *concise*. Conciseness is the ability to express in an abstract form all essential capabilities and properties in a precise manner. A specification should be concise, clear, and annotated where necessary to assist in achieving the correctness dimension characteristics. Blum asks: "After all, of what value is an unambiguous specification when it is misinterpreted in the context of a naive theory?" [Blu92]. So a specification should contain clear and understandable commentary to assist the reader.

To write a specification that has the characteristics discussed above requires a formalism that has both expressive power and the functionality to specify and reason about the structure and behaviour of transaction models. Further, to show that the specifications are consistent, verifiable, traceable, unambiguous, and correct requires expressing the specifications using mathematical notations and then use the underlying formalism to prove these properties.

## 3.2 Transactions Specification Formalism Requirements

A transaction's behaviour is the state sequences resulting from the transaction's operations. An operation is a discrete activity that implements a definable functional abstraction. Operations may be either simple or composite, concurrent (having one or more separate threads of control) or nonconcurrent, and possibly parameterized. Therefore, operations define the behaviour of an object (or system). Thus, the behaviour of a trans-

action depends on how its components behave individually and how the components communicate with one another.

Before enumerating the desirable features of a specification language suitable for specifying transactions, it helps to reiterate the characteristics of transactions. Transactions are characterised by such features as interleaving of operations, concurrent execution (parallel executions of operations and not necessarily joint), mutual exclusion (at lower level), and synchronization of operations on shared resources to avoid deadlock. Synchronization of operations on shared resources involves temporal properties. Furthermore, transactions allow some form of nondeterminism and concealment of results, and the utilization of dependencies and causality relationships to guarantee execution correctness. These transaction's characteristics and the properties of transactions' specification (see Section 3.1) taken together help to explain the need for certain required language features.

Therefore, any formalism for specifying transaction system behaviours **must** possess the following language features:

1. Have *abstraction facilities* capable of producing a precise, consistent, and unambiguous specifications that abstracts away implementation dependent issues. It should be rigorous and mathematically based so that its designs are built from provably correct constructs.

2. Support for *proof obligation.* The framework should possess a formal specification and verification method based on a sound mathematical basis to allow formal proofs of correctness. This proof obligation can be discharged automatically by using the specification's language proof assistant or by relying on a rigorous mathematical proof analysis and logical reasoning.

3. Supports both *concurrency* and *functionality.* The formalism must provide primitive constructs to capture concurrency (such as interleaving, parallelism, nondeterminism), choice, sequence, and interrupt. In addition, the formalism should provide functional operators.

4. Have adequate data modelling facilities. The framework should appropriately capture the structured *data objects* used in database computations since these data objects provide a meaningful representation of the problem being solved. Further, the framework should provide elegant means of specifying operations on structured data in addition to capturing communications among the system's components. Thus, it should provide adequate mechanisms for defining and manipulating data objects suitable for database applications.

5. *Semantically rich* enough to capture the various semantic elements necessary for defining the consistency/correctness and reliability criteria of the different complex transaction models. The language's syntax and semantics should be clear and easy to understand. In addition, the formalism should be flexible and open-ended so that it is applicable to any transaction model.

6. Supports *temporal properties* of transaction models and provides facilities to correctly express time by selecting appropriate models and notations. The knowledge of events occurrence times provide answers to such questions as whether one event occurs before, after, or simultaneously with another. Thus, time information is necessary for events ordering that is essential for understanding transaction operations synchronization.

7. Additional features which a specification formalism should exhibit are:

   - suitable to the particular *application domain*. That is, it should be suitable for describing both the problem to be solved and the algorithms used to solve it in a common underlying framework.
   - Have support for *modularity* and *composition* of a specification's modular units into a model of the whole system. These features enable a meaningful organization of the information in the specification.

   Thus, the framework should be problem-oriented allowing system designers to structure the specification as closely to the problem as possible which reduces the chance of errors.

No single formal specification language currently has *all* the above properties. A combination of features from formalisms for specifying sequential processes and others used for specifying concurrent ones is the most promising.

## 3.3 Considerations for Transaction Specification

This section describes the fundamental issues in writing a transaction's specification. Their understanding assists us in shaping the way the specifications should be written.

### 3.3.1 Constraints Specification

A database's integrity is defined by its adherence to its integrity constraints. Assuming that the database is consistent and adheres to the integrity constraints then the formal specification is correct and verifiable if the transaction's precondition and postcondition are enforced. The precondition describes the database state which is required for the transaction to execute correctly. Postcondition describes the database state after a transaction executes if the preconditions hold.

Birell *et al.* [BGHL87] argue that any behaviour of a concurrent system is describable using a sequence of atomic action so they propose mutual exclusion to synchronize shared data access. Thus, two states define the observable effect, namely the state immediately preceding and following the action. Such mutual exclusion completely isolates the critical sections but is inappropriate for transaction systems because it is too restrictive and eliminates concurrency. Furthermore, it is impractical and incorrect to describe a non-atomic transaction's observable effects with two states since its effects may span more states and other processes actions may interleave with its actions. To overcome this problem, each non-atomic transaction's execution can be specified as a predicate that defines the allowable action sequences. Therefore, the interleaving of operations from other transactions or schedules of operations performed concurrently must be equivalent to the *effects* of executing the transactions serially [EGLT76].

Predicates that define state are written with first-order predicate logic expressions combined with process and communication primitives. A first-order predicate logic for-

mula is written in conjunctive normal form and describes the characteristics of the system. Predicate-based models define the executions allowed by a transaction set by describing when interleaving is possible. Therefore, the set of data items appearing in the predicates must be defined. Concurrency is possible by exploiting the predicates associated with the transactions. However, the predicates must satisfy the well-formedness property (see Section 2.1.1 page 23). Determining whether a given string is a well-formed formula with respect to $G$, a set of syntactic rules, is decidable[7] [WF83]. Well-formed formulae are used to build a proof system to verify the correctness of specifications. A proof is a sequence $A_1, A_2 \cdots, A_n$ of well-formed formula such that for each $i$, either $A_i$ is an axiom of the system or $A_i$ is a direct consequence of some preceding well formed formula by applying one of the inference rules. New facts are established by deducing or proving their truth from previously known facts.

### 3.3.2 Proof Requirement

Every specification determines the set of allowable actions that hold in it. These are the properties of the specified system expressible in the given syntax and semantics of the specification language. It is vital to have some effective ways of proving that an action is a consequence of a specification. A proof system shows the correctness of the specifications. Guttag and Horning [GH80] and Woodcock [Woo89] argue that proofs help to explain systems' behaviours and ensure correctness. Thus, by proving that selected properties follow from a specification we can show that the specification expresses the system's desired needs.

A proof system must be sound and complete but no sound and complete proof system exist [ST84] so we only ensure soundness. Thus, developing an inference rule for every operation allows the use of compositional inference rules to deduce facts about compound specification developed from those components.

Two proof styles [Woo89] are possible: *natural deduction* deduces one property from another; and *logical reasoning* uses an equivalences chain. Both styles employ direct

[7] A decision problem is decidable if there exist an algorithm that determines after finitely many steps whether a given string is well-formed.

proof, proof by contradiction, or proof by mathematical induction [Woo87]. Although behavioural equivalence is a fundamental programming methodology, the transaction specification systems must guarantee behavioural equivalence to the execution effects of transaction systems (programs) built from the specifications. Jahanian and Mok [JM86] argue that formal verification methods prove the consistency of safety requirements from the system specification. Thus, proving consistency and correctness is only with respect to a specification. The proofs themselves may be informal or formal, but their logical basis is formal.

### 3.3.3 Time Property

A transaction system's execution is described by the sequence of events that occur. In time sensitive applications/operations (such as operations on shared objects), it is important to reason about *what* events occur and *when*. The central concept here is *timed* execution. Often, timing requirements demand that certain communication signals and operations must execute at certain absolute times or within certain time limits. So to characterize dynamic properties and behavioural patterns of transaction systems, the temporal component of the transaction system's specifications is important. Time-based events are important for active databases and their incorporation into a general formal framework addressing events is desirable.

Therefore, specifying and verifying transaction system behaviours requires reasoning about a sequence of states and their times of occurrence. Time is associated with each state in the execution such that the sequence of times is monotonic and only finitely many states are assigned in a given time interval. The timing property is essential to fully capture and define the synchronization behaviour aspects of concurrent transaction systems. Expressing interleaving order requires specifying the relationships between events and their times of occurrences. For example, consider the concurrent execution of transactions $T_i$ and $T_j$. If transaction $T_i$ is value dependent on $T_j$ then $T_j \prec T_i$. This ordering depends on the knowledge that $T_i$ uses a data object previously modified or accessed by $T_j$. Recall, an operation precedes another at their execution time (see Section 2.4 page 41). Section 5.6 presents more details on relationship between events

and their occurrence times.

Timing properties are essential in specifying database transaction systems. They constrain the interactions between different components of the system and between the system and its environment. Minor changes in the precise timing of interactions may lead to radically different behaviours. Therefore, time is an essential synchronization mechanism for solving certain task coordination problems. However, unlike hard real-time systems that are characterized by a time-dependent utility function (that is, satisfying the time deadlines constraints determines the correctness of results), database transaction systems are soft real-time systems[8] [Son88] because satisfying hard time deadlines is not the prime consideration but rather the correct synchronization of the transactions.

The time domain in some approaches is discrete and in others is continuous. Different kinds of computation problems are best addressed using different notions of time so there is no "best" uniform approach [Sch93b]. Thus, the nature of the application dictates the timing requirements.

Generally, *TIME* is a totally ordered set of time points represented as a mapping to the number line. The type *TIME* is closed under addition and subtraction. One can identify three models of time: *discrete, clocked*, and *continuous* time. The model of time is *discrete* if the time line is isomorphic to the integers, and is *continuous* when the time line maps to the real number field. Discrete time represents the temporal evolution of the system as an enumerable sequence of snapshots; each describing the state of the system at a certain time. Continuous time represents the system evolution by a sequence of intervals of time, with a description of the system's state during each interval. In the continuous case, the behaviour of the system is represented by a mapping from an interval of nonnegative real numbers representing time intervals to system states. The intervals can overlap at endpoints only. There are infinitely many points in continuous time in any interval of non-zero length so reasoning in continuous time is problematic

[8]Soft real-time transactions have timing constraints but there may still be some justification in completing the transactions after their deadlines. Unlike hard real-time systems, catastrophic consequences do not result if soft real-time systems miss their deadlines. In hard real-time systems, a transaction's timeliness and criticalness usually determine the usefulness of the results (i.e., the results are available *in time* to influence the process being monitored or controlled).

and generally intractable. Clocked time is an approximation of continuous time. A special event is assumed to execute regularly but the relative order of events between the checkpoints remains important. In most transaction systems discrete time representation suffices because database transaction communication is essentially synchronizable at specific discrete points where interprocess communication and concurrent access to shareable data objects occur. Alfaro and Manna [AM95] show that if a temporal formula has the property of finite variability, its validity in the discrete semantics implies its validity in the continuous one.

### 3.3.4 Causality

Another important property that demands explicit representation in a concurrent transaction system specification is *causal* relationship. Causality defines the ability of one action to affect the other. Causality may be due to value dependencies [MCFP96] or the effects of the different types of dependencies (inter-transaction and intra-transaction dependencies). Causal dependencies constrain the synchronization of operations/transactions in which they exist (see Section 5.6 for details). This concept enhances correctness and recovery and thus improves system reliability.

Causal relationships are essential in cooperative problem solving environments. An investigation of causality relationships and time in complex transaction systems as it affects their dependability is required.

## 3.4 Specification Language Selection

Existing formal specification languages fall short of the language features outlined in Section 3.2. Languages such as *Z* [Spi88] and *VDM* [Jon86] are unsuitable because they lack mechanisms for capturing concurrency, parallelism, communication, interruptions, and coordination among subcomponents vital to concurrent transaction systems. Moreover, they lack the concept of time and thus lack primitives for specifying timing constraints. Similarly, functional decomposition techniques such as Structured Analysis/Design are unsuitable for specifying transaction systems because there is no support for formal ver-

ification, nondeterministic system behaviour cannot be suitably modelled, and timing properties do not integrate well with the rest of the requirements. Hierarchical Coloured Petri Nets are not well suited to the needs of advanced database transactions specification because they lack suitable data handling and manipulation mechanisms, and it is often difficult to distinguish between precedence and causality [TM91] in Petri Nets models in general.

The process algebra family of specification languages (such as CSP [Hoa85], CCS [Mil89], and ACP [Ber88, BB91]) have been found [Fid94, EBO95] most suitable for the specification and design of distributed systems. These languages support concurrency and their variants have incorporated timing properties as part of their primitive operator set. However, CSP and CCS have simple form, lack data values [Fid94], and use events as the basis for creating more complex structured behaviours. Also, data types are treated informally, typically restricted to simple types such as integers or characters.

The inability of the process algebra languages to provide structured and complex data types undermine their expressive power and usefulness for advanced database transaction applications where data fields are accessed. Clearly, the types of data objects available in a language have a profound impact on its suitability for a particular application since these data object types provide a meaningful representation of the problem being solved [Was80]. To remedy this situation, the basic CSP specification language is extended beyond the primitive atomic data types to include structured data types[9]. The formalism and and its semantics follow in Section 4.6.

[9] De Giacomo and Chen [DC96] added the concept of explicit *global store* that associates a blackboard with a process to handle data requirements in modelling a dynamic system using a variant of CCS.

# Chapter 4

# TIMED CSP FUNDAMENTALS

CSP captures communication behaviours effectively but cannot describe functional aspects of a system. Therefore, CSP is augmented with record data extensions and logic expressions to cater for the inadequate functionality. Extending CSP's expressive power makes it a suitable formal method for transaction systems description outlined in Section 3.2. Critical aspects of a transaction system environment including error handling, interrupts, concurrency, safety and liveness are capturable.

Initial key theoretical foundations are presented. The goal here is to provide an understanding of the basic language and its applicability. A brief review of the constructs, primitives, and operators applicable to transaction protocols are provided (see [DS92a, Hoa85, HJ96, Sch90, Sch93b] for a complete treatment). Finally, the necessary record data type extension to CSP are presented.

## 4.1 Elements of TCSP

Timed CSP models system behaviour that are often distributed, reactive, and subject to timing constraints. Examples of such systems include communication protocols, transaction processing systems, aircraft control systems, and space and combat missile control systems. CSP's specifications are composed of events and processes logically combined according to a set of grammatical rules. A collection of event names is called an *alphabet.* An alphabet is a set of names (or symbols) of events useful for describing a process. An

alphabet is finite and non-empty. A process only engages in events within its alphabets. If $P$ is a process, we write $\alpha P$ to denote its alphabets. These alphabets model the interface between this process and its environment.

A process is modelled in terms of the possible interactions it can have with its environment. These interactions are described in terms of events. An event is one of the following: (i) the execution of an operation by a process, (ii) the sending of a message, or (iii) the receipt of a message. An event is an instantaneous atomic action. Each event in the alphabet of a process can either be an elementary event (represented as an atom) or a composite event (such as a function) whose components consist of other events. Events can be (a) value returning, (b) self replacing (i.e., modifies at least one data variable) or (c) communication signals. Therefore, a transaction's execution is a chain of events, starting and ending on an element of the alphabet set.

Processes are abstract programs describing the behaviour of a system in terms of events.

**Definition 10** *Process:* A process is collection of syntactic objects which represents the pattern of behaviour of an object. ■

A process may represent the system or some components within the system. Processes are defined using an equational notation. The statement $A = B$ introduces a process $A$ which behaves as $B$. The behaviour of a system expressed in terms of events can only be determined when a program is executed.

**Definition 11** *Program:* A program is an ordered collection of processes. ■

Note that it is possible that a set of processes is used to implement a program. An observation of a program is a record of observable behaviour during an execution.

Timed CSP provides extensions to support timing specification in real-time systems. Timed semantics are used to analyse timed properties while untimed semantics describe other aspects of the specification. Since Timed CSP is a superset of CSP, the refinement relation ensures that untimed specifications remain valid in the timed semantics. Schneider [Sch93a] has shown that it is possible to preserve checks on safety properties when refining an untimed description to a timed one. Untimed descriptions

differ from timed ones by the addition of arbitrary delays in the timed version. This means any proofs that hold for the untimed description also hold for its timed version (see [Sch93a]).

The syntax of CSP processes includes primitive operators that enable the specification of concurrency, non-determinism, and hiding (i.e., encapsulation) elegantly and separately in transaction systems. Deadlock and recursion are definable.

## 4.2 The Language of Timed CSP

A semantic model for the language, in which each program is identified with a set of observations, is required. The different CSP's semantic models are named according to the type of observations made; for example, in the *timed traces model*, observations are sequences of a timed model. A process is an element of a semantic model; a set of observation which defines a pattern of behaviour.

Davies and Schneider [DS92a, DS9*] outlined the properties of the model of computation that influence the construction of the semantic models. These properties include: *maximal progress*, *maximal parallelism*, *finite variability*, *synchronous communication*, and instantaneous events. Maximal progress ensures a program will execute until it terminates or requires some external synchronization while maximal parallelism guarantees that each component of a parallel combination has sufficient resources required for their execution. In a finite time interval, finite variability ensures that a program may undergo only finitely many state changes. Synchronous communication ensures that each communication event requires the simultaneous participation of every program involved. Action duration is modelled by considering the beginning and the end of the action as separate events. This model of computation is consistent with that employed in [Hoa85].

Timed CSP is defined by the following *Backus-Naur Form* (BNF) grammar rule:

| | |
|---|---|
| $P ::= STOP \mid SKIP \mid WAIT\ t \mid$ | basic |
| $a \longrightarrow P \mid P; P \mid$ | sequential |
| $P \Box P \mid P \sqcap P \mid a : A \longrightarrow P_a \mid P \rhd P \mid$ | choice |
| $P \bigtriangledown Q \mid P \lightning P \mid$ | interrupt |

| | |
|---|---|
| $\| A_P\ P \mid P\ {}_A\|_B\ P \mid P \mid\mid\mid P \mid P \|_{\mathbf{A}} P \mid$ | parallel |
| $f(P) \mid P \backslash A \mid$ | abstraction |
| $\mu X \bullet F(X)$ | recursion |

This extends Hoare's syntax. A more detailed description and the various semantical models of the language may be found in [DS92a, DS92b, DS9**, Sch93a, Sch93b, Sch9**, Hoa85]. In the above BNF rule, event $a$ is drawn from the set of all alphabets $\alpha$ and event set $A$ ranges over the set of subsets of $\alpha$. In Timed CSP the time $t$ is a non-negative real number. This thesis assumes time is non-decreasing monotonic. Time is modelled by a non-negative integer without loss of generality[1] (by result of [AM95]). Time is global in that it passes at the same rate in each process. See Section 5.3 page 108 for logical time assignment rules in transaction environments. Timed CSP's operators for timeouts and timed interrupts have time variable associated with them.

For illustrative purposes (for the rest of this section) let $P$ and $Q$ be processes and $t$ be a time value.

*STOP* and *SKIP* are two special processes. *STOP* performs no events at all (deadlock). It is a broken program which will never engage in any external communication and fails to terminate, so it represents the end of a pattern of interaction. *SKIP* performs no events but ends the process it is found in or itself immediately and successfully. *SKIP* represents a successful end of a pattern of interaction. Successful termination is regarded as a special event, denoted by the symbol $\surd$. The operator *WAIT* $t$ is a delayed form of *SKIP* that terminates successfully after the specified time $t$.

**Sequence:**

The $\longrightarrow$ operator is an action transition relation. It is of the form: $\longrightarrow$: *state* $\times$ *event* $\rightarrow$ *state*. The operator allows us to add communication events to a program. The process $a \longrightarrow P$ means that when event $a$ occurs the process $P$ is immediately executed. In

[1]The thesis uses a discrete time extension of CSP with relative timing. Time is divided into slices indexed by natural numbers. These time slices represent time intervals of length that corresponds to the time unit used. Thus, time is represented as "a sequence of discrete, quantized instants $t_1, t_2, \ldots$ " [KL91].

other words, from an initial state, say $t$, the process can perform the event $a$ and move to state $t'$ where the execution of $P$ starts. This transition statewise is represented as $t \xrightarrow{a} t'$.

The sequential composition operator (;) transfers control upon termination. Thus, $P$; $Q$ represents the sequential composition of $P$ and $Q$. The program first behaves like process $P$ until $P$ performs the special event $\surd$ at which time it continues by behaving like process $Q$. If $P$ does not terminate successfully $Q$ will not start; so the program $P$; $Q$ does not terminate successfully. The $\surd$ event is invisible to the environment but occurs when $P$ terminates successfully. Sequential composition relies upon the use of *SKIP*. Note that state information does not persist across a sequential composition. That is, in the process $P$; $Q$ the initial state of $Q$ is independent of the final state of $P$.

**Choice:**

There are two choice operators — *external* and *internal* choice operators. The external choice operator ($\Box$) offers the environment a choice between the initial events of its two arguments. For example, $P \Box Q$ represents a *deterministic* external choice between processes $P$ and $Q$. If it is possible to perform the first event of $P$, but not the first event of $Q$, then $P \Box Q$ behaves as process $P$. On the other hand, if it is possible to perform the first event of $Q$, but not the first event of $P$, then $P \Box Q$ behaves as process $Q$. If both the first event of $Q$ and the first event of $P$ are possible, then the choice between $P$ and $Q$ is nondeterministic. Another form of external choice is the process $a : A \longrightarrow P_a$ called indexed external choice which offers an external choice of initial event $a$ drawn from a set $A$, which may be infinite. This is used to model process input from a channel. Note that external choice and indexed external choice represent a choice offered to the environment between a number of processes.

Internal choice is specified with the $\sqcap$ operator. For example, the program $P \sqcap Q$ represents an internal choice between $P$ and $Q$ where the outcome of this choice is *nondeterministic*. This operator is used to represent runtime nondeterminism.

**Input and Output:**

The expression $c!v$ denotes the output of value $v$ on channel $c$. The value transmitted is determined by the sending process. Similarly, the expression $c?v$ denotes the input of value $v$ on channel $c$. The input value is determined by the environment of the process. An input process is ready for any value while an output process sends just one value. Thus, $c?a \longrightarrow P$ denotes a process that is ready to engage in any event of the form $c.a$ while $c!a \longrightarrow P$ denotes a process which must perform the event $c.a$ before behaving as the process $P$. For example, if channel $c$ carries values of type $A$, the process $c?a : A \longrightarrow P_a$ accepts any value $a$ of type $A$ on channel $c$, and behaves accordingly. These processes may be used to represent input and output behaviour on a channnel.

**Timeout:**

In Timed CSP, the timeout program $P \overset{t}{\triangleright} Q$ may behave as $P$, or offers a choice between $P$ and $Q$, according to whether the timeout has occurred or not. For example, the process $P \overset{t}{\triangleright} Q$ will behave as $P$ but if no event of $P$ takes place before time $t$ the process behaves as $Q$. Also, if the first event of $P$ attempts to occur at exactly $t$, then the outcome is nondeterministic. In untimed CSP, the resulting behaviour is nondeterministic choice because it has no notion of time.

**Interrupt:**

There are two forms of interrupt — *event* and *timed* interrupts. The interrupt program $P \overset{i}{\triangledown} Q$ behaves as $P$ until the first occurrence of *interrupt event* $i$ where upon control is transferred to $Q$ and process $P$ is discarded.

The operator $\overset{t}{\lightning}$ (the lightening sign) passes control from one program to another after a predetermined time has elapsed. This is called *timed interrupt.* Without timing information, the first program may be interrupted at any point. For example, the process $P \overset{t}{\lightning} Q$ behaves as $P$ until time $t$ when it behaves as $Q$. This operator differs from the timeout operator which aborts a process only if no external activity has occurred. The timed interrupt operator is very useful in specifying certain real-time transaction

operations such as banking transactions.

**Concurrency:**

In general, the parallel composition of two processes $P$ and $Q$ is denoted $P \parallel Q$. Components of a parallel composition may progress separately on internal disjoint events. If both components can engage in a common timed event, then so can the parallel combination. The parallel composition operator has many forms. One form is a synchronised parallel combination of a set of programs parameterised by a corresponding set of interfaces $A_P$. For example, given $K = \|_{A_P}\ P$, each event $a \in A_P$ requires the participation of every subprogram $P$. Every pair of subprograms in $P$ must cooperate on each event from the intersection of their interface sets. Events outside $A$ may, however, occur independently.

Another simple form is the binary parallel combination $P\ {}_A\|_B\ Q$ in which program $P$ may perform only those events in $A$, program $Q$ may perform only those events in $B$, and the two programs must synchronize on events drawn from the intersection of $A$ and $B$.

Furthermore, if $A$ is a set of events common to processes $P$ and $Q$ the partially-interleaved combination of $P$ and $Q$ is written as $P \|_A Q$. In this combination, processes $P$ and $Q$ synchronize upon only those events in set $A$ and on termination. Other events may occur independently, even if they appear in both alphabets (i.e., they interleave on all other events). This form of parallel subprograms combination represents *hybrid* parallel combination.

In all forms of concurrency operators, all communicating events are synchronized; they can occur only if both the sender and the receiver are willing to cooperate.

**Interleaving:**

Asynchronous parallel combination ($|||$) occurs when subprograms evolve concurrently but without interacting. Generally, the $|||$ operator is a communication free merge operator. The process $P \ ||| \ Q$ represents the unsynchronised concurrent execution of processes $P$ and $Q$. Events of $P$ and $Q$ occur independently and an event can be refused only if both components independently refuse it. Interleaved processes progress independently.

They are causally independent (see Section 5.6) and may occur simultaneously. If the intersection of $P$'s and $Q$'s alphabets is nonempty then we have partially interleaved execution of the actions of both $P$ and $Q$.

**Indexes:**

The interleaving, parallel composition, and internal and external choice operators can be indexed. For example, $|||_{x:X}\ P(x)$ denotes the interleaving of each of the processes $P(x_i)$ for each $x_i \in X$ where $1 \leq i \leq \#(X)$. The indexed deterministic choice operator, $\Box_{x:X} P(x)$ denotes the choice between $P$ evaluated in an environment with the variable $x$ bound to each value in the set $X$. Similarly, the corresponding indexed nondeterministic choice operator and parallel composition is given by $\sqcap_{x:X} P(x)$ and $||_{x:X}\ P(x)$, respectively.

**Relabelling:**

The relabelled program $f(P)$ has a similar control structure to $P$, with observable events renamed according to the function $f$. Relabelling enables renaming of processes and events.

**Hiding:**

The program $P \backslash A$ behaves as $P$, except that events from set $A$ are concealed from the environment of the program. Hidden events no longer require the cooperation of the environment so they occur as soon as $P$ is ready to perform them.

**Recursion:**

Recursion is modelled either by using equations or $\mu$ operator. The recursive program $\mu X \bullet F(X)$ behaves as the process $F(X)$ where each instance of variable $X$ represents a recursive invocation. The process $\mu X \bullet F(X)$ satisfies the equation $P = F(P)$ where $P = X$. These programs have well-defined semantics if the function $F$ is *guarded*[2] [Hoa85].

[2]A function $F$ is guarded if every free occurrence of $X$ in $F(X)$ is preceded by at least one observable event.

For example, the recursive equation $P = a \longrightarrow P$ defines a process which may engage in as many events $a$ as the environment will allow. Recursive process definitions *must* start with at least an event prefixed to all occurrences of the process name on the equation's right-hand side. The following illustrates a recursive process definition of a perpetual clock [Hoa85: page 28]:

$$CLOCK = \mu X : \{tick\} \bullet (tick \longrightarrow X)$$

where $\alpha CLOCK = \{tick\}$. An equivalent equational definition of the perpetual clock is:

$$CLOCK = (tick \longrightarrow CLOCK)$$

*Summary:*

To model a system, process definitions describing the individual component's behaviour within the system are specified using the above Timed CSP operators. The component processes are composed to specify the entire system's behaviour.

## 4.3 Nature of CSP Specifications

Expressing parallelism (including concurrency) between processes is possible in CSP. This sometimes requires cooperation among the processes. Cooperation in a computation among processes makes communication[3] necessary if one process needs an intermediate result produced by another process. Also, synchronization is necessary because the former process must be suspended until the result is available. Coordination between processes is implemented by message exchange between pairs of processes using the synchronized execution of communication events in both processes. A communication sequence of process $P$ is the sequence of all communication events in which $P$ has participated. *Message passing* achieves both communication[4] and synchronization[5].

[3] If communication is expensive, the gain in computational speed as a result of concurrency may be lost in additional communication cost.

[4] Using a *shared memory* variable achieves only communication so additional care is required to synchronize processes that communicate using shared memory.

[5] *Synchronous* message passing implements the synchronization of the processes involved in that the send operation is completed only after execution of the receive operation resulting in a two-way synchro-

A CSP's description of the desired system behaviour is split into a *trace* and a *state* part. The trace part consists of a set of *trace assertions.* The trace assertions describe the relative order in which the events can occur in the sequences or traces of communications between system and environment. The state part specifies which values are communicated over channels and consist of a set of local state variables and a set of communication assertions.

By specifying the trace and state assertions, one can capture adequately the desired system's functionalities. Thus, restricting the execution relation to certain events guarantees correct allowable executions by a transaction set.

## 4.4 Semantic Models for Timed CSP

The semantic models specify the intended behaviour of a program. Each program is associated with a set of behaviour in the semantic model so a predicate on the semantic set corresponds to a requirement upon the program.

The *traces model* (the most widely understood and popular version of CSP) associates each program with a set of observable event sequences. These sequences are called *traces.* A trace defines the set of finite events sequence the process may perform. The semantics of a process in the traces model is a set containing every trace possible for that process. For example,

$$traces(P \sqcap Q) = traces(P) \cup traces(Q).$$

It is used mainly to demonstrate safety properties where it is necessary to show that every possible trace of the system is acceptable. The analysis of this set permits us to determine if the system is safe. Therefore, the traces model is a safety model because a program within the required traces model will only guarantee not to do something unexpected. For example, in the traces model of CSP, the following predicate says that program $P$ never performs a visible action[6]:

---

nization between the sender and the receiver.

[6]The definition is read as "for all $tr$ which belongs to the set of sequences of $P$'s trace such that the predicate $tr = \langle\,\rangle$ holds.

$$\forall\, tr \in traces(P) \bullet tr = \langle\,\rangle$$

Also, in the traces model, the process STOP is associated with the set $\{\langle\rangle\}$, containing only the empty trace. STOP, therefore, meets the above requirement; defined using *satisfiability* relation, denoted **sat**, as follows:

$$\text{STOP } \mathbf{sat}\ tr = \langle\,\rangle.$$

The **sat** relation is a semantic function of processes in the trace model representing the set of acceptable traces performable by that process. The traces model is sufficient for untimed safety requirements. The program, in general, does not guarantee freedom from deadlock. The traces model tells us nothing about liveness.

However, by specifying constraints that ensure event possibility will verify that the system does not deadlock or livelock. Howles [How94] shows that deadlock freedom is a feature of untimed description that extends to a timed description using refinement. Capturing liveness conditions requires *readiness* or *refusal* information in the semantic sets which leads to the *failure model.* The set of events that are refused by a process is called *refusal* set. The process is unwilling to perform any event in the set. For any given process, the refusal set of the process represents "the pathological events that must be avoided" [Low94] for the process to execute correctly. The failure model associates each program's trace with various events set, $X$, offered by the environment. The set $X$ is a *refusal set* of a program $P$, denoted $refusals(P) = X$, if it is possible for $P$ to deadlock on its first step when placed in this environment[7]. If *failures*, denoted $(tr, X)$, is present in the semantic set of a program $P$, then $P$ may perform trace $tr$ and refuse to engage in any event from $X$.

A trace may follow an infinite sequence of internal events which is called *divergence* so the third behavioural aspect included is the *failure-divergence model.*

The traces, failure, and failure-divergence semantics models have their corresponding timed semantic versions. For example, a timed trace is a finite sequence of timed events drawn from $\mathbf{R}^+ \times \alpha$ such that the times are in non-decreasing order. $(\mathbf{R}^+ \times \alpha)^*$

[7]For example, the simple process $a \longrightarrow P$ refuses every set that excludes the event $a$. Note that a process can refuse only events in its alphabet.

is the finite sequences of timed visible events. (Recall that this thesis uses integer values for time so $\mathbf{R}^+$ is replaced by $\mathbf{N}_1$) Similarly, a timed refusal set is a set of timed events consisting of a finite union of refusal tokens. The information in a trace is simply a record of events occurring at particular times; all simultaneous events may occur in any order in a trace. The operational effects have no instant causality between visible events since simultaneous events may occur in any order.

Both timed and untimed versions may be used in the analysis of the same system so that the results can be subsequently combined. A proof in the untimed traces model will hold in any of the timed models [Sch93b]. This research uses both the traces and failure semantic models where appropriate to capture both safety and liveness properties of the systems elegantly. Safety and liveness properties represent desirable facets of a process to accurately describe the intended behaviour. Safety properties that are specifiable include mutual exclusion and absence of deadlock. Liveness properties include termination and responsiveness while fairness properties ensure that every process has a chance of executing.

### 4.4.1 Reasoning with Traces

Trace projections are useful when reasoning about traces. Since a trace is a sequence, the usual primitive operators of the sequence data type are freely available in CSP. These are in for membership, $\frown$ for concatenation of sequences, $\restriction$ for the restriction of a sequence to some elements, and the functions *head*, *last*, and *tail*. In the following definitions, let $\text{seq}_1$ denotes a nonempty sequence, $A$ and $B$ be sequences, and $C$ be a set.

*head*$(A)$ is the first element of a nonempty sequence, formally defined as:

$$A \neq \langle\,\rangle \Rightarrow \mathit{head}(A) = A[1]$$

*last*$(A)$ is the last element of a nonempty sequence. So,

$$A \neq \langle\,\rangle \Rightarrow \mathit{last}(A) = A[\#A]$$

*tail*$(A)$ is the sequence with the first element of a nonempty sequence removed.

$$A \neq \langle\,\rangle \Rightarrow \mathit{tail}(A) = A - \mathit{head}(A)$$

$A$ in $B$ means that $A$ is a contiguous subsequence of $B$.

$A \restriction C$ is the restriction of $A$ to elements from the set $C$.

The following examples expound these primitives.

Let$\Phi = \langle a, b, d, f, g, l \rangle$.

Obviously $\Phi$ is a nonempty sequence so we can write $\text{seq}_1\ \Phi$. Thus,

$\Phi = \langle a, b, d, f \rangle \frown \langle g, l \rangle$,

$head(\Phi) = \langle a \rangle$,

$last(\Phi) = \langle l \rangle$,

$tail(\Phi) = \langle b, d, f, g, l \rangle$,

$\langle b, d \rangle$ in $\Phi$ but $\neg\ (\langle a, d \rangle)$ in $\Phi$, and

$\Phi \upharpoonright \{a, d, g\} = \langle a, d, g \rangle$.

Additional operators' operations are described in context as required throughout.

### 4.4.2 Proof Mechanism

CSP uses denotational semantics to validate the algebraic laws. The semantic equations for each model support the construction of a *compositional proof system.* A compositional proof system is a set of inference rules relating the properties of each program to the properties of its syntactic subcomponents. Such rules are of the form:

$$\frac{\begin{array}{l} \textit{antecedent} \\ \ldots\ldots \\ \textit{antecedent} \end{array}}{\textit{conclusion}} \quad [\ \textit{side condition}\ ]$$

The antecedents are assertions (or predicate) relations on the components of the program and the side conditions are optional assertions unrelated to the transitions. Consider the following example:

**Facts**

1. Birds can fly.

2. Birds have feathers.
3. Pigeon is a bird.

From the given facts (axioms) the following logical conclusion is immediate.

Pigeon have feathers and can fly.

A CSP's proof rule in the traces model representation of the above is:

$$\frac{P \textbf{ sat } S(tr) \qquad P \textbf{ sat } T(tr)}{\text{Pigeon} : P \textbf{ sat } S(tr) \wedge T(tr)}$$

where $P$ represents Birds, $S(tr)$ and $T(tr)$ are the specifications of the requirements *can fly* and *have feathers*, respectively.

A characteristic of the compositional proof system is that any property guaranteed by some component of the system must be true of the whole system. Similarly, any constraint imposed by a system's component is a constraint on the whole system.

To specify how a process behaves requires constraints on the set of traces associated with that process. These constraints (or behavioural specifications) are expressed as predicates on traces. To show that a process satisfies a behavioural specification requires that every trace of the process satisfies the corresponding predicate. In other words,

$$P \textbf{ sat } S(tr) \Longleftrightarrow \forall\ tr \mid tr \in traces(P) \bullet tr \Longrightarrow S(tr)$$

This means that every possible observation of $P$'s behaviour is described by $S(tr)$. Thus, the **sat** relation ties a process to its characteristics set of traces as illustrated in Figure 4.1. The **sat** relation enables proof of properties as a chain because if $P$ **sat** $S(tr)$ and $S(tr) \Rightarrow T(tr)$ then $P$ **sat** $T(tr)$. That is, if a specification $S$ logically implies another specification $T$, then every behaviour described by $S$ is also described by $T$. So every process which satisfies $S$ must also satisfy $T$.

### 4.4.3 The Implementation of a Process

A process is usually implemented as a function $F()$ that takes events as argument. Let $B$ denote the set of events in which the process $P$ is initially prepared to engage. If $x$

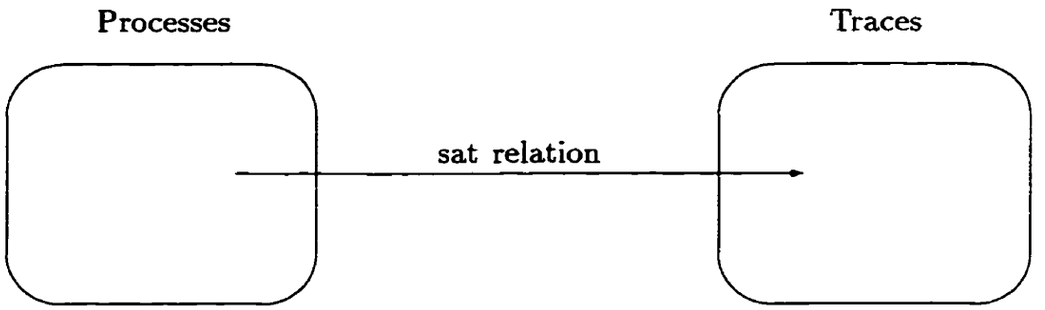


Figure 4.1: The Satisfaction Relation

is the first event of $P$, then for each $x$ in $B$, $F(x)$ defines the future behaviour of the process $P$, denoted $P'$. That is, the implementation of $P$ is given by

$$(\forall x : B \bullet F(x) \Longrightarrow P' \wedge (s : trace(P) \bullet head(s) = x))$$

For any given process ($P$) the complete set of possible traces can be predetermined using a function *traces(P)* as defined by Hoare [Hoa85].

## 4.5 Interleaving Semantics

Let $t = traces(P)$ and $u = traces(Q)$. Then the arbitrary interleaving of the actions of process $P$ and process $Q$ is defined as follows:

$$s \textbf{ interleaves}(t, u)$$

where $s$ is the interleaved actions. Suppose $t = \langle a, b, c, d, e, f \rangle$ and $u = \langle l, m, n, o, p \rangle$. One possible interleaving of $t$ and $u$ traces is:

$$s = \langle a, b, c, l, m, n, d, e, o, p, f \rangle$$

No ordering relation is imposed on $s$. Unsynchronized concurrency[8] (parallelism) occurs when there is no need to synchronize actions associated with independent entities in a

[8]Recall concurrent processes can function independent of one another or they can be asynchronous which means that they require occasional synchronization and cooperation.

system. Thus, if $\alpha P \cap \alpha Q = \varnothing$, then $P \parallel Q$ is an arbitrary interleaving of actions from the process $P$ with those from process $Q$. Formally,

$$traces(P \parallel Q) \equiv traces(P \parallel\parallel Q) = \{s \mid \exists\, t : traces(P);\ u : traces(Q) \bullet s\ \mathbf{interleaves}(t, u)\}$$

In a concurrent system, concurrent processes can invoke operations on an object so it is necessary to give meaning to interleaved operation executions. Suppose some operations of $P$ and $Q$ conflict on a data item $O$. If flow dependence of $P$ on $Q$ is required on their access to $O$, then their actions' arbitrary interleaving as defined above is unacceptable. Some ordering relations **must** be imposed on the interleaved actions of both processes. Thus, the interleaving of $P$ and $Q$ is defined as:

$$s\ \mathbf{interleaves}(t, u) \wedge \{\forall k \in t \mid k = t \upharpoonright O \wedge \forall l \in u \mid l = u \upharpoonright O \bullet l \leq k\}$$

This ordering relation permits the proper synchronization of $P$ and $Q$ with respect to their actions on data item $O$. Interleaving-based semantics captures observable temporal behaviour of processes.

Any two processes, say $P$ and $Q$ can be composed to run concurrently but requires caution because a dependency may arise from sharing data objects. If both processes have events in common, they synchronize their actions when executing such items.

We now turn to the problem of extending TCSP to capture the record structures required to describe database transaction specifications.

## 4.6 Supporting Record Data Type in CSP

Arbitrarily complex data objects must be appropriately captured by the specification technique. The components of a structured data object which may belong to a distinct type are each smaller than the whole structured data object. It is possible to access or modify the constituent parts independently in a structured data type.

Recall that, one of the limitations of CSP is its inability to define structured data elegantly. There are no primitives available to define a *record* data structure (compound

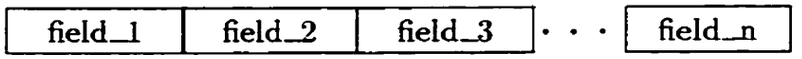


Figure 4.2: A Sample Record Data Structure

type[9]) and hence there are no means to access the attributes of such data.

**Definition 12** *Structured data:* A non-atomic variable (i.e., structured data) has at least two fields. ■

Figure 4.2 is a sample record data structure.

The ability to construct structured data types in CSP proves useful in producing more natural abstract manipulations in systems specifications. Data objects are defined by their representation and by the constructor operations used to create them in conjunction with the definitions of operations that manipulate them. The extensions are amenable to formal mathematical reasoning because they contain regular and simple structure. The specification of the data structure provide a vehicle to address some of CSP's limitations with the goal of increasing CSP's expressive power. As the field selectors are fixed at declaration time, the components of a record may be accessed as efficiently as scalar data objects.

### 4.6.1 Defining Record Data Structure

The two operations on structured data types are: (1) the construction operation which generates a new type (the data type being defined) from primitive ones and (2) the accessing operation. Therefore, we require two operators that permit the definition of and access to records in CSP. These operators define the structuring principles that show how the components of the structured data objects may be created, accessed and

[9]Compound types are built up from elementary data types. A record's component types are heterogeneous. A record represents a single data item in the problem domain. CSP is restricted to simple atomic data types only.

modified[10]. For notational convenience we shall use $\oplus$ for data structure declaration[11] (or definition) and the symbol . for data field reference binding[12]. The $\oplus$ and . operators are the basic vehicles for defining, generating, and manipulating (see Section 4.6.2) record objects. The . operator enables the application of value transforming function to objects. In other words, the . operator specifies operation which extracts components from a record structure. Therefore, it is essential that the distinction between the name of a variable (its identifier), the area it is stored (its reference or address), and the value stored must be clearly understood.

For illustrative purposes, consider the Pascal declaration of an employee record.

*TYPE*

*Employee* = **Record**

*Name* : *string*[25];

*Dependents* : *integer*;

*Rate* : *real*

**end**;

The scope of a field identifier is the record in which that identifier is declared. This means that the same identifier may not be used to specify two different fields within the same record, but the same name may be used elsewhere in the specification for some other purposes. Since the components of a record may be of any data type, in particular they may be other records, records may be nested. For example, the pascal declaration

*TYPE*

*Staff* = **Record**

---

[10]Modification of an arbitrary field is through the side effects of the assignment operation.

[11]The declaration of a record type specifies the **name** and the **type** of the various fields of the record. Implicit in the declaration of a record is a tree structure since the fields of a record can themselves be records. [In this thesis, only records whose components are scalar data objects are considered but a recursive definition is an immediate consequent. Also, variant records [LN93, Mar91] are not considered].

[12]The binding of a variable to its value involves three bindings [WC93]: (1) the binding of the variable's name to its declaration (name-declaration binding), (2) the binding of its declaration to a store location (declaration-reference binding), and (3) the binding of the storage location to a value (reference-value binding).

*Name* : *string*[25];
*Dependents* : *integer*;
*BirthDate* : *Date*;
*Rate* : *real*
**end**;

is a hierarchical record where the field variable type *Date* is a record consisting of the fields *Day*, *Month*, and *Year*. The fields within such a nested record may be accessed by simply affixing a second field identifier to the name of the enclosing record. There is one record for each employee so *Staff*[$i$], for example, refers to record of the $i$th employee.

The syntax of the structured data objects declaration is given below:

$$\oplus[data\_name] == \langle\langle field\_1, field\_2\ \{, field\_3, \ldots, field\_n\}\rangle\rangle \&\& \langle\langle type\_specifications\rangle\rangle$$

where

1. $\oplus$ is the data structure constructor operator,
2. *data_name* is the name of the data structure constructed,
3. $\langle\langle$ and $\rangle\rangle$ are used to enclose the data fields and the specifications of the data type of the individual data fields,
4. *field_i* where $i \geq 1$ is a data field (i.e., an attribute of the data structure constructed),
5. *type_specification* is the specification of the data types[13] of the fields, listed in the order in which the fields appear in the definition — this must be a 1-1 mapping so that for each *field_i* there exists $type_i$, the type of *field_i*,
6. && is used to associate the types specification with the data fields definitions,
7. $n \geq 2$, and items within the { } are optional.

Note that the symbols $\oplus$, $\langle\langle$, $\rangle\rangle$, and && are parts of the constructor syntax. Structured objects, therefore, are specified as a sequence of attributes and types[14]. The record with rank $\leq 1$ is defined nowhere in this thesis because an empty record is of no interest and

[13]The type of an object determines the values it can take and the set of operations used to manipulate these values. Data objects have a **value** and a **type**.

[14]In a nutshell, a record is a Cartesian product of domains or domain values.

a record with only one field is a degenerate case of atomic types which are available in CSP. All fields of a record must be specified at creation time. This restriction makes it possible to perform static checks on accesses to record fields. Operations involving the $\oplus$ succeeds if the identifier (its argument) is not atomic type and there is no such identifier previously declared. Otherwise, it returns the identifier and an error message.

Writing specifications is based on representing a data object by the construction operation whose evaluation yields the record data object. Exploiting their mathematical properties is dependent on having a finitely representable specification of the structure of each data object. Formally, we shall use the following definitions:

$$FIELD ::= identifier$$

$$ATOMIC ::= integer \mid real \mid char$$

$$TYPE ::= user\_defined \mid ATOMIC$$

$$FIELDTYPE == \text{seq } TYPE$$

$$DATASET == \{id : identifier \mid id \text{ is a variable already declared in the system}\}$$

Now we define the structure of a record noting that each attribute must be unique.

$$FIELDS == \text{seq } FIELD$$

$$\forall F : FIELDS \bullet$$

$$\quad \forall i, j : 1..\#F \bullet i \neq j \Rightarrow F[i] \neq F[j]$$

Further, let **type_of** be a (prefix) function that returns the type of its argument. Its signature is:

$$\mathbf{type\_of} : FIELD \longrightarrow TYPE$$

Similarly, let **Val** be a function that returns the stored value of an attribute. Its signature is:

$$\mathbf{Val} : FIELD \longrightarrow value$$

For example, the notation $\mathbf{Val}(s)$, where $s$ is an attribute, returns the current value of the attribute $s$. That is, $\mathbf{Val}(s)$ is the message or data value corresponding to the particular instance of the variable $s$.

The record type can now be defined as:

$$RECORD == \langle\langle FIELDS \rangle\rangle \&\& \langle\langle FIELDTYPE \rangle\rangle \bullet$$

$$\quad \#FIELDS = \#FIELDTYPE \wedge$$

$$\#FIELDS \geq 2 \wedge$$
$$(\forall i : 1..\#FIELDS \bullet$$
$$\textbf{type\_of}\ FIELDS[i] = FIELDTYPE[i])$$

In other words, *RECORD* is a finite set of identifiers associated with types. Its elements are associations from the set of *identifiers* to the type TYPE such that each identifier can assume an element of the corresponding type. Thus, a record is a partial function from labels to values so all labels in a given record must be distinct (a constraint in the definition of *FIELDS*).

The syntax of the structured data objects declaration is given below:

$$\oplus : identifier \longrightarrow RECORD$$
$$\forall\ data\_name : identifier \bullet$$
$$\oplus[data\_name] = data\_name \mid data\_name = RECORD \Leftrightarrow$$
$$\neg\ (\exists\ x_i : identifier \mid x_i \in DATASET \bullet x_i = data\_name) \wedge$$
$$\textbf{type\_of}\ \ x_i = \neg\ ATOMIC \Rightarrow$$
$$(\exists\ F : FIELDS;\ T : FIELDTYPE \bullet$$
$$data\_name = \langle\!\langle F \rangle\!\rangle \&\& \langle\!\langle T \rangle\!\rangle)$$

To demonstrate the application of the $\oplus$ constructor operator, consider the following. Let $x$ be a **record** type with the fields *id* and *value*. Let us further assume that *id* is of type integer and *value* is of type *TIME*. This will be defined as follows:

$$\oplus[x] = x \Rightarrow x \bullet \langle\!\langle id, value \rangle\!\rangle \&\& \langle\!\langle \mathbb{N}, TIME \rangle\!\rangle$$

The above declaration introduces the two variables $x.id$ and $x.value$ which combined create the variable $x$ of the type record. The order of access to the components of $x$ is insignificant.

To allow components (data fields of the data structure) to have different types, components must be referenced using a qualified name instead of an expression[15]. Thus, to reference any field in *data_name*, the field name is bound to the *data_name* by using the . operator. In other words, one can access each field of a record directly by using a field-designated variable of the form "record-name . field-name". Effectively, the .

[15]Components of a record are identified by their label rather than by their "ordered" position as in lists. Equality of records is componentwise.

symbol is the dereferencing[16] operator. For example, *data_name.field_2* refers to the data value in *field_2* of the record type *data_name* variable. The effect of *data_name.field_2* is to access the value stored in the location referenced by the *field_2* component of *data_name*. Thus, it is convenient to write $x.id$ to denote the *id* component of $x$ and $x.value$ to denote the *value* component of $x$. Similarly, writing the CSP statement $input?x.id$ means the input of a value for $x$'s *id* component through the channel *input*. The binding of a variable to a value occurs as a result of either an input or an assignment statement. So an assignment operation of 4 to $x.id$ written as $x.id := 4$, for example, is actually $\mathbf{Val}(x.id) = 4$ since it is the stored value that is being modified. This can be read simply as: change the *id* component of $x$ to have the value 4. Thus, the effect of $input?x.id$ is the same as the direct assignment of the value read to the *id* component of $x$.

Before formally defining the selector operator some functions are defined. The function **Attrib** returns the set of all attributes (i.e., fields) in the structure of a record data type.

$$Attrib : RECORD \longrightarrow \mathbb{F}_1\ FIELD$$

$$\forall r : RECORD \bullet$$

$$Attrib(r) = \{s : \mathbb{F}_1\ FIELD \mid (\exists a : FIELD \mid a \in s \bullet a \text{ in } r\}$$

The function **Nfields** returns the number of attributes (or fields) in a record. In other words, the application of *Nfields* yields the cardinality of the record structure. *Nfields* is formally defined as:

$$Nfields : RECORD \longrightarrow \mathbb{N}_1$$

$$\forall r : RECORD \bullet Nfields(r) = \#Attrib(r)$$

The formal definition of the selector operator follows:

$$\_.\_ : RECORD \times FIELD \longrightarrow FIELD$$

$$\forall r : RECORD \bullet r.x_1 = \mathbf{Val}(x_1) \Leftrightarrow x_1 \in Attrib(r)$$

[16]Dereferencing is the process of finding the value of a variable given the reference. For example, in the expression $x = y + 3$, $y$ is dereferenced to find its value and the constant 3 is added to it. The resulting value is then assigned to variable $x$ whose location is obtained by reference to the name $x$. Note that when a language has no assignment operator, names can be directly bound to values.

Accessing a record's field is applying the record to that field. The operation produces an error if the accessed field is undefined. Thus, $r.x_1$ extracts the value corresponding to the label $x_1$ from the record $r$, provided a field having that label is present. This condition is enforced statically.

In general, besides record access, there are many operations that can be performed on a record structure such as renaming a field, overriding the value of a field, adding a new field, and deleting an existing field. Such operations are omitted from this thesis (see [GM94] for details).

### 4.6.2 Semantic Definitions

In defining the semantics of the operators, the denotational technique is employed. The meanings of abstract programs or parts thereof are expressed in terms of semantic functions which map them into various semantic domains (the members of which are usually themselves functions). Thus, "meaning" is given simply as functions from state to state. A denotational semantics specification of the object language consists mainly of a set of semantic equations which define the semantic functions[17].

We shall define a semantic function $\mathcal{N}$ : *identifier* $\longrightarrow$ *RECORD*. Informally, we say that given an identifier, $\mathcal{N}$ defines the corresponding record. Also, we shall write $\mathcal{N}[\![x]\!]$ to represent the semantic definition of the expression or variable $x$. That is, the $[\![\ ]\!]$ in the semantic function's argument often enclose expressions (or syntactic objects) in the object language[18] variables. The left side of a semantic definition begins with an application of the semantic function to an argument representative of the syntactic domain. The argument is expressed in the same notation as the abstract syntax rules. Often, the value of any expression depends on its context or the environment that tells what values the identifiers denotes. For example, $\mathcal{N}[\![x]\!]_\rho$ denotes the value of $x$ in the environment $\rho$. Since the context of the environment of the record operators is understood, there is no need to include the environment information in the semantic definitions.

[17]Semantic functions map abstract programs (or parts of program) into semantic domains.

[18]The object language is the language being defined.

First define the syntactic and semantic domains[19] and secondly, define the semantic functions. The semantic domains describe the underlying values manipulated by the constructs of the language while the syntactic domains used to define the abstract syntactic structure of the language corresponds to the syntactic variables (or categories) defined. Each distinct field in a record is associated with a domain of values. It is important to avoid confusion between the language being defined and the notations used to define it so all syntactic objects within ⟦ ⟧ are in the defined language and have defined semantics.

For obvious reasons, we will not treat a full language but only those portions which are relevant to the record data type extension. Thus, the language with record extension, $\kappa\mathcal{CSP}$, is an extension of *TCSP* with distinguished constructs for record expressions.

The following syntactic domains and corresponding metavariable are listed.

| | | | |
|---|---|---|---|
| $I$ | $\in$ | **Ide** | Identifiers |
| $E$ | $\in$ | **Exp** | Expressions |
| $D$ | $\in$ | **Type** | Data types |
| $K$ | $\in$ | **Stmt** | Statement |

The abstract syntax rules are:

| | | | |
|---|---|---|---|
| $K$ | ::= | I | variable |
| | | $\mid \oplus[I]$ | abstraction |
| | | $\mid \oplus[I]^+$ | abstraction and composition |
| | | $\mid I.I$ | record access |
| | | $\mid E; E$ | application |
| | | $\mid E\ E$ | application |
| | | $\mid I?E$ | input |
| | | $\mid I!E$ | output |

where $[I]^+$ means at least one $I$. For example, we can construct $I, I$.

Thus, $\kappa\mathcal{CSP} = TCSP \cup K$

[19] A domain is a complete lattice. A complete lattice $D$ is a partially ordered set in which each subset $x \in D$ has a least upper bound in $D$ denoted $\top$ and a greatest lower bound in $D$ denoted $\bot$.

Following the style taken in [Pag81, Sto82, Low93, Low94, Hay84], let $S_D$ be the set of all identifiers (their associated types will be assumed). Formally,

$$S_D \triangleq \mathbb{P}(FIELD)$$

CSP's syntax includes a variable, say $X$, that can be associated with a process. **Ide** represents the domain of such a variable. In the current study, $X$ can be a record data type. For variables to have any meaning, the environment space, $E_D$, is defined as

$$E_D \triangleq Ide \longrightarrow S_D$$

The primitive semantic domain for truth values is:

$$\pi : Truth = \{true, false\}^0$$

The following defines the required semantic function

$$\mathcal{N} : Stmt \longrightarrow E_D \longrightarrow S_D$$

Thus, $\mathcal{N}$ associates each construct with its value. The domain operator '$\longrightarrow$' associates to the right.

The semantic definitions are given below.

1. $\mathcal{N}[\![\oplus[x]]\!] \triangleq x \Rightarrow x = \langle\!\langle F \rangle\!\rangle \&\& \langle\!\langle T \rangle\!\rangle \mid F \in FIELDS \wedge T \in FIELDSTYPE$
   $\wedge\ DATASET' = DATASET \cup \{x\}$
2. $\mathcal{N}[\![\oplus[x \times y \mid y \text{ is atomic data}]]\!] \triangleq \mathcal{N}[\![(\oplus[x])]\!] \times y$
3. $\mathcal{N}[\![\oplus[x, y, z]]\!] \triangleq \mathcal{N}[\![\oplus[x]]\!] \,;\, \mathcal{N}[\![\oplus[y]]\!] \,;\, \mathcal{N}[\![\oplus[z]]\!]$
4. $\mathcal{N}[\![\oplus[x \mid x \text{ is an existing record data type}]]\!] \triangleq x \wedge error \Rightarrow x \in DATASET$
5. $\mathcal{N}[\![\oplus[x \mid x \text{ is atomic data}]]\!] \triangleq x \wedge error$
6. $\mathcal{N}[\![x.y]\!] \triangleq \mathbf{Val}(y) \mid y \in Attrib(x)$
7. $\mathcal{N}[\![input?x.y]\!] \triangleq x.y := value \mid \text{dom } value \in \mathbf{type\_of}\ y$
   where *value* is the data read/captured via the input channel.
8. $\mathcal{N}[\![input!x.y]\!] \triangleq value = \mathbf{Val}(x.y) \mid value \in \mathbf{type\_of}\ y$
9. $\mathcal{N}[\![x.id]\!] \wedge \mathbf{type\_of}\ id \in \text{dom}\,\mathbf{R}$ obeys all algebraic laws of closure, associativity, distribution, and existence; e.g
   $$\mathcal{N}[\![x.id + y]\!] = \mathcal{N}[\![y + x.id]\!] \Leftrightarrow \mathbf{type\_of}\ y = \mathbf{type\_of}\ id$$

Notes:

1. Any variable created with $\oplus$ can be used within any CSP construct that requires such variable. For example, if $A$ is a record type then $a : A \longrightarrow P_a$ is a valid process that offers a choice of initial event $a$ drawn from the record type $A$ and behaves accordingly.

2. The operator $\oplus$ is CSP-expressible and non-destructive.

3. By merging definitions (4) and (5) above, we have the following:

$$\begin{aligned}
&\mathcal{N}[\![\oplus[x]]\!] = x \wedge \mathit{error} \Leftrightarrow \\
&\quad (\exists\, x_i : \mathit{identifier} \mid x_i \in \mathit{DATASET} \bullet x_i = x) \ \vee \\
&\quad\ (\neg\, (\exists\, x_i : \mathit{identifier} \mid x_i \in \mathit{DATASET} \bullet x_i = x) \wedge \\
&\qquad\quad \textbf{type_of}\ x = \mathit{ATOMIC})
\end{aligned}$$

4. $\mathcal{N}[\![\oplus[x]]\!] = x \Leftrightarrow \mathit{Nfields}(x) \geq 2$ is an important condition that must be satisfied by all definitions of the record data type.

5. Consider the following example for *Axiom 2* above. Let the variable *Person* be a record type consisting of the fields $\mathit{name}, \mathit{age}, \mathit{weight}$, and $\mathit{height}$. Also, let *Qualification* an atomic data variable. The data types of the fields are assumed. Thus,

$$\begin{aligned}
&\mathit{Person} \times \mathit{Qualification} \\
&\equiv\ [\mathit{name}, \mathit{age}, \mathit{weight}, \mathit{height}] \times \mathit{Qualification} \\
&\equiv\ \oplus[\mathit{Person}] \times \mathit{Qualification} \\
&\equiv\ \oplus[\mathit{Person} \times \mathit{Qualification}]
\end{aligned}$$

6. Note that if $A_\rho \equiv B_\rho$ for all $\rho$ then by extensionality, $A \equiv B$. That is, one may be transformed into the other by a finite sequence of permissible applicable steps. A simple structural induction suffices to show that all the occurrences of a variable denote the same value.

**Summary**

The two operators, $\oplus$ and $.$ enjoy elegant mathematical properties that make them compatible with other CSP operators. The $\oplus$ operator provides recursive type construc-

tion. Application of the operators succeeds only if they syntactically conform to their structural specifications. Application of $\oplus$ to type-incompatible argument is considered constant and the arguments are left intact (non-destructive property). The strong typing discipline is a considerable asset when writing correct specifications. Finally, the given denotational semantics is consistent with respect to record's type structure. The semantic definitions given contain the appropriate projection operations for manipulating record data type. The extension presented in this section further enriches the expressive capability of CSP thereby increasing its applicability to the specification of database transaction systems.

# Chapter 5

# PROTOCOL SPECIFICATIONS

This research assumes error free multicast communication channels exist between databases[1]. Informally, reliable multicast ensures that all or none of the recipients receive any transmitted message. Semantics for multicast include receiving, acknowledgement, and the delivering of messages. Their specifications is outside the scope of this research.

Some terms that are used in the specifications are defined in the following section. Any additional concept will be defined within context as required.

## 5.1 Basic Definitions

A data object ($x$) is an ordered pair ($x_v, \mathcal{TS}$) where $x_v$ is the object's value and $\mathcal{TS}$ is an arbitrary *time structure*. For example, if only time of reference is required $\mathcal{TS}$ is a value drawn from the domain of TIME (see Chapter 4 page 75). However, if read and write time is required, $\mathcal{TS}$ could be a structure containing more detail (see Section 5.11.1 page 161). There exists a set *Data* which is the collection of all data objects. A data object's value may be simple or a collection of values. The metafunction $VALUE[x, t]$

[1]Send and receive events enable the interaction of processes via communication channels (in CSP context) of the communication subsystem of the distributed system. Assumptions about communication channels are: (1) reliable — every sent message is received exactly only once, (2) FIFO property — messages are routed via channels in FIFO fashion, and (3) the channel capacity is unbounded. In multicasting messages are sent to a collection (not necessarily all) of the processes.

returns the value of $x$ at time $t$, where $x$ is any data object.

**Definition 13** *Read set:* The set of data read by a transaction $T$ is denoted by $RS_T$.

$$RS_T = \{a \mid a \in Data \bullet a \text{ is read by } T\}$$ ■

**Definition 14** *Write set:* The set of data written by a transcation $T$ is denoted by $WS_T$.

$$WS_T = \{a \mid a \in Data \bullet a \text{ is written by } T\}$$ ■

The union of Definitions 13 and 14 is the set of data accessed by transaction $T$.

**Definition 15** *Access set:* The set of data read or written by a transcation $T$, denoted by $Acset_T$.

$$Acset_T = RS_T \cup WS_T$$ ■

A transaction can potentially see (read or write) any data object available in the system. The set of such data objects is called the *View* set and is denoted by $Vwset_T$.

**Definition 16** *View set:* The set of objects visible to a transaction $T$.

$$Acset_T \subseteq Vwset_T$$ ■

These objects can be partial results from other transactions.

**Definition 17** *Operation :* An operation (on a data item), denoted $Op$, is a well defined action that, when applied to any permissible combination of known entities, produces a new entity. — (adapted from [Ros84]). ■

An operation identifies and performs an action on data. An operation could modify the data or access the data without affecting it. Thus, all operations on data can be represented by simple *reads* and *writes*. That is, $Op = \{r, w\}$ where $r$ and $w$ are read and write, respectively. Every operation is failure atomic so the status of every operation on termination is either success or failure. Operations preserve the domain of data to which they are applied.

The invocation and execution of an operation is atomic so it can either commit or abort. Thus, the occurrence of one precludes the occurrence of the other. Formally,

$$\forall op_k : Op \mid commit(op_k) \Rightarrow \neg abort(op_k) \vee abort(op_k) \Rightarrow \neg commit(op_k) \quad \text{[R0]}$$

Operations can be composed by using the composition operator ($\circ$) defined formally as:

$$\_ \circ \_ : Op \times Op \longrightarrow Op$$
$$\forall p, q : Op \bullet$$
$$\exists x \in Data \bullet$$
$$p \circ q = p(q(x))$$

Thus, if $p$ and $q$ are operations that operate on $x$ and $q$ acts on $x$ and result is operated on by $p$. The composed operations on $x$ is given by: $q \circ p = q(p(x))$

Transaction operations are:

- *Begin* transaction — initiates the execution of a new transaction. Returns a transaction identifier (*id*) used to identify all operations in the transaction.

  $BeginTransaction \longrightarrow id$

- *Read* or *Write* — an action which may be low-level such as read or write a data item or record. The read or write operations are usually atomic.

- *Task invocation* — requests the services of a procedure which itself is a transaction. The request results in the execution of the transaction (procedure) often concurrent with the invoking transaction. An example of a task invocation operation is a request to transfer money to a bank account which can be implemented as two nested subtransactions consisting of deposit and withdrawal.

- *Precommit* — the identified transaction has completed its operations and is ready to commit.

- *Commit* — the transaction has terminated normally and all of its effect are made permanent. The committed transaction is removed from the system.

  $Commit(id) \longrightarrow Boolean$

- *Abort* — the transaction has terminated abnormally and all of its effect should be removed.

  $$Abort(id) \longrightarrow Boolean$$

- *End* transaction — indicates the completion of the identified transaction. The transaction may be committed or aborted.

  $$EndTransaction(id) \longrightarrow (Commit \mid Abort)$$

The operations commit, abort, and precommit are transaction terminating operations; begin and end transaction are transactional service operations; read and write are transaction access operations; and finally task invocation is a transaction service request.

Each transaction access operation designates an access to some particular object. The *effect* of an access operation is the changes it makes to its operand and the value it returns [BL93]. Every operation returns a value after execution. So the effect of an operation refers to the value of a data item set by a *Write* operation and the result returned by a *Read* operation.

**Definition 18** *Commute:* Two operations commute if they return the same values and leave the data base in the same final state when executed in either order. ■

Thus, a pair of operations commute if their execution have the same effect on a database independent of the relative ordering of the operations' execution .

**Definition 19** *State:* The state of an object is its value at a point in time. ■

In timed specifications, a state is augmented with time parameter, $t$. Thus, $state_t$ denotes the value of the object at time $t$.

Generally, an instance of a data item often has associated with it some other items such as currency indicators, that together determines its state. The state of a data "reflects the dynamic aspects of the data as it changes as the result of an operation" [TL82].

Content may not change when an operation is performed but the state always does[2]. Not all operations cause a change in the value of the data but they cause a change in state of the data. Therefore, operations can change the state of a data item by either changing the content or related control mechanisms like the currency indicators. A write operation alters both the data value and the state. A read operation does not alter the content but the state. In other words, operations transform a data state ($state_i$) to another data state ($state_j$). For example, if $state_i$ is the present state of a data item $x$ and after executing the operation ($op$) on $state_i$ we get a new state $state_j$ if and only if $i < j$.

To determine an operation's effect on a data item requires the functions $s$ and $r$. The function $s$ returns the final state produced by an operation. Its signature is:

$$s : STATE_t \times Op \longrightarrow STATE_k$$
$$\forall t, k : TIME \mid t, k \in STATE \bullet t < k$$

The function $r$ gives the return value of an operation. Its signature is:

$$r : STATE_t \times Op \longrightarrow VALUE[x, t]$$

For a read operation, there exists a "transient" state change but the content of the state is unaffected. When the operation commits, the resulting state is equivalent to the state before the read operation. State changes are observed through return values. The return value of a read operation depends on the value set by a previous write operation on the data object while a write operation is return value independent of another write or read operation. In summary, $s$ is equivalent in effects to a write operation while $r$ is a read operation.

**Definition 20** *Conflict:* An operation $p_i$ of transaction $T_i$ conflicts with another operation $p_j$ of a different transaction $T_j$, $T_i \neq T_j$, if they both access the same data and if

[2]Note that an operation is an event. An event is anything that can be identified with a specific point in time [KL91]. It causes a transition. A state transition is a change in state [Rum91, Boo94] which is caused by an event. When an event is processed, the process's clock is automatically advanced to the next instant.

there exists a state of the data such that the sequences $p_i p_j$ and $p_j p_i$ operating on that state either return different values or leave the data in different final states. ■

In other words, two operations conflict if their effects on the state of an object or their return values differ depending on their execution order. To formally define conflict using $s$ and $r$ defined above, let $x_o$ be the present state of $x$.

$$\begin{aligned}
&\mathit{Conflict} : Op \leftrightarrow Op \\
&\forall\, p_i, p_j : Op \bullet \\
&\quad \mathit{Conflict}(p_i, p_j) \Leftrightarrow [s(x_o, (p_i \circ p_j)) \neq s(x_o, (p_j \circ p_i)) \vee \\
&\qquad\qquad r(x_o, (p_i \circ p_j)) \neq r(x_o, (p_j \circ p_i))] \qquad \text{[R1]}
\end{aligned}$$

Note that $s(x_o, (p_j \circ p_i))$ is equivalent to $s(s(x_o, p_i), p_j)$ in programming language parlance. Consider the following example. Let the initial value of $x = 10$. Let $p_i$ and $p_j$ be the operations $\mathit{read}(x)$ and $\mathit{write}(x = 5)$ respectively. The following represents the steps that determine if the two operations conflict.

$$s : (x_o(10), \mathit{read}(10)) = x_1(10) \Rightarrow (x_1(10), \mathit{write}(x = 5)) = x_2(5)$$

$$r = 10$$

Now interchange the relative order of both operations and observe both the final value set and the return value.

$$s : (x_o(10), \mathit{write}(x = 5)) = x_1(5) \Rightarrow (x_1(5), \mathit{read}(5)) = x_2(5)$$

$$r = 5$$

Since $(r = 5) \neq (r = 10)$, then $p_i$ and $p_j$ conflicts.

Two database operations in a schedule conflict if they return different values or leave the database in a different state when their order is reversed. It is desirable that the operations return the same values and leave the database in the same state after reversing their order.

In summary, conflict between any two operations can occur only when *all* the following conditions are satisfied: (*1*). They access the same data item, (*2*). At least one of the operations is a write operation (a state modifying operation), and (*3*). A change

in the relative execution order of the operations give different data states or returns different values. That is, two operations conflict if their effect on the state of an object are dependent on their order of execution. Conditions (*1*) and (*2*) are captured by:

$$RS_{T_i} \cap WS_{T_j} \neq \emptyset \vee RS_{T_j} \cap WS_{T_i} \neq \emptyset \vee WS_{T_i} \cap WS_{T_j} \neq \emptyset$$

where $T_i$ and $T_j$ are two distinct transactions. Using these (implicitly), [R1] captures condition (*3*).

Therefore, for any two conflicting operations, say $p_i$ and $p_j$, belonging to transactions $T_i$ and $T_j$, respectively, the corresponding transactions in which they participate conflict. So two transactions $T_i$ and $T_j$ conflict if:

$$\begin{aligned} Conflict(T_i, T_j) \Leftrightarrow (Acset(T_i) \cap Acset(T_j) = \{B \mid B \neq \emptyset\}) \wedge \\ (\exists\, o_i, o_j : Op \mid o_i \in T_i, o_j \in T_j; \\ \exists\, x : Data \mid x \in B \bullet Conflict(o_i, o_j)) \end{aligned}$$

**Definition 21** *Active:* An operation (event) is active if its execution has been initiated but has not terminated. ■

Similarly, a transaction $T$ is active if and only if it has started but has yet to perform abort or commit operation. Formally,

$$\begin{aligned} &\forall\, T : Transaction \bullet \\ &\quad Active(T) \Leftrightarrow trace(T) \neq \langle\,\rangle \wedge \\ &\qquad \neg(a \text{ in } trace(T) \mid a \in \{abort, commit\}) \end{aligned}$$

**Definition 22** *Conflict set:* The conflict set of a transaction $T$, denoted $ConflictS_T$, contains the operations of active transactions that may conflict with T. ■

Two transactions can access the same object, say $O_x$, if the active operations of one excludes the conflict set of the other. For example, $T_j$ can access $O_x$ without conflicting with another transaction $T_i$ if the conflict set of $T_j$ excludes active operations of $T_i$ on $O_x$.

## 5.2 The Model

The processes interact with one another in a coordinated fashion using messages to cooperatively process transactions. Each transaction is assigned a unique identification[3] number *id*. The system uses the transaction *id* to bind all the transaction's operations together as a logical unit to maintain its atomicity.

The problem domain can be represented as a graph, $G = (node, link)$, where the *node* represents participating databases and the *link* represents the interconnections between the databases. The nodes are partitioned into two equivalence classes, $cnode_1$ and $cnode_2$ such that $\#cnode_1 = 1$ and $\#cnode_2 = n$ where $n$ is the number of participating LDBs in the MDB. Thus, $cnode_1$ represents the MDB's interface and $cnode_2$ represents the set of LDBs. So the *degree* of a node is the number of edges entering into and leaving from it. The function *degree*() calculates the degree of a node:

$$degree(cnode_1) = n \mid n = \#LDBs$$

$$\forall\, m \in cnode_2 \bullet degree(m) = 1$$

To represent this with our formalism requires definition of the basic types:

$$[NODE, LINK]$$

Communication between connected nodes is possible. The signature of the function, *links* defines the connection between any two connected nodes.

$$links : LINK \rightarrowtail\!\!\!\!\!\!+\;\; (NODE \times NODE)$$

Since the set of nodes available at any time is finite, communication can only take place between nodes in the network. Thus,

$$available\_nodes : \mathbb{F}\, NODE$$

Similarly, the set of available links (that is, the connection between any two adjacent nodes) is finite. Communication between nodes is bidirectional along the links. Thus,

[3]The generation of transactions *ids* is outside the scope of this research but the research assumes that a mechanism for generating the *ids* exist.

the available links are:

$$available\_links : \mathbb{F}\ LINK$$

It would be useful to be able to project out the first or the second element of an ordered pair of nodes. So we define *first* and *second* respectively to achieve this.

$$first, second : NODE \times NODE \longrightarrow NODE$$

$$\forall x, y : NODE \bullet$$
$$first(x, y) = x \ \wedge$$
$$second(x, y) = y$$

The function *snodes* transforms an ordered pair into a set.

$$snodes : NODE \times NODE \longrightarrow \mathbb{F}\ NODE$$

$$\forall x, y : NODE \bullet$$
$$snodes(x, y) = \{n : NODE \mid n \in first(x, y) \vee n \in second(x, y)\}$$

The network of participating databases is:

$$available\_nodes = \bigcup\{l : links \bullet snodes(\text{ran } l)\}$$

$$available\_links = \text{dom } links$$

$$\forall i : \text{ran } links \bullet$$
$$\#(snodes(i)) = 2 \wedge (snodes(i) \in available\_nodes)$$

In the above definition, it is possible to add local databases or delete connections between any two local databases. Thus, the network can grow or shrink in size. Also, it might be necessary to determine whether any signal transmitted from one local database can be received at the MDB's interface and vice versa. Thus we need a function *reachable*:

$$\_reachable\_ : NODE \longleftrightarrow NODE$$

$$\forall n_1, n_2 : NODE;\ e_1 : LINK \bullet$$
$$n_1\ reachable\ n_2 \Rightarrow (n_1, n_2 \in snodes(\text{ran } links(e_1)))$$

Figure 5.1 shows a basic graphical representation of the problem domain. Communication along the edges is bidirectional.

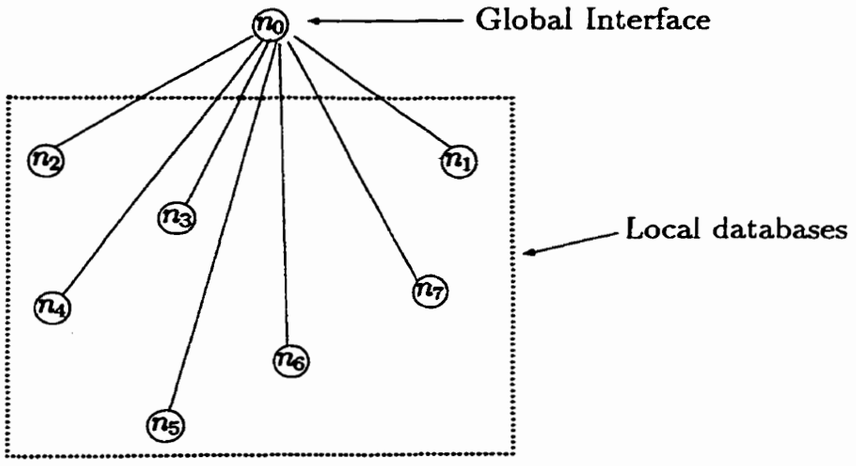


Figure 5.1: A Graphical Model of Problem Domain

## 5.3 Logical Time Assignment

An assignment of time value to every event in a trace that preserves the consistency of all possible dependencies among the events in the trace is crucial. The assignment mechanism must uphold the following:

1. No two events of the same transaction are assigned the same logical time.
2. The logical times at each transaction are monotonically increasing based on their occurrence order.
3. The logical time of any send event is less than the corresponding receive event's logical time.
4. The number of events assigned logical times smaller than $t$ is finite for any time $t$.

Properties (2) and (3) imply that the order of logical times must be consistent with the ordering relation $<$.

## 5.4 Definitions

Some initial definitions are required but others will be added as required since their understanding requires context. The maximum value any time variable can assume is an integer constant represented by $N$. The primitives of sequence data type (see Section 4.4.1) are applicable.

### 5.4.1 Preamble

The following are the basic types:

$$[CHANNELS, EVENTS]$$

$$\mid N : \mathbb{N}_1$$

$$BOOLEAN ::= true \mid false$$

A collection of events which are acted upon by a process (the definition of a process follows shortly) is denoted by $ALPHABETS$, formally

$$ALPHABETS == \mathbb{F}\ EVENTS$$

It is important to reason not only about what events can occur in a system but also the time of their occurrences. Time is necessary for the ordering of events. In this domain time is discrete and non-negative integer so time is represented by

$$TIME == \mathbb{N}_1$$

Each event occurs at an instance of time so event occurrences are timed events[4].

$$TIMEDEVENT == TIME \times EVENTS$$

[4]Events happen in both time and space so we can have sequential events in which two events occur on the same data item one after the other in time or concurrently in which two events occur at the same time but on different data items.

The function *time* returns the time of occurrence of the event performed in a timed event.

$$time : TIMEDEVENT \longrightarrow TIME$$

$$\forall s : TIMEDEVENT \bullet$$
$$time(s) = (t \mid t : TIME) \Leftrightarrow (\exists a : EVENTS \bullet s = (t, a))$$

The function *event* returns the event performed in a timed event.

$$event : TIMEDEVENT \longrightarrow EVENTS$$

$$\forall s : TIMEDEVENT \bullet$$
$$event(s) = (a \mid a : EVENTS) \Leftrightarrow (\exists a : EVENTS; t : TIME \bullet$$
$$s = (t, a))$$

Events are executed by a process[5] that describes system's behaviour. Events may be channel events whereby communication, input or output occurs through a named channel. Every channel has an associated type. For example, input commands such as $input?x$ means that $x$ is a variable ($x \in ALPHABET$) of the process under consideration of type *input*. With the $\oplus$ operator introduced, variable (data) elements may have structure, for example, records. The variables are typed variables.

$$PROCUNIT ::= nochan\langle\langle TIMEDEVENT\rangle\rangle$$
$$\mid chan\langle\langle CHANNELS \times TIMEDEVENT\rangle\rangle$$

The functions *message* and *channel* respectively extracts the message and channel name components of a channel event.

$$\forall s : PROCUNIT \mid s \in chan \wedge s = c.x \bullet$$
$$channel(c.x) = c \ \wedge$$
$$message(c.x) = event(x)$$

The set of all messages which a process $P$ can communicate on a channel, say $c$, is defined by:

$$\alpha c(P) = \{v \mid message(c.v) \in ALPHABETS \ P\}$$

[5] A process starts, performs a finite number of events (also called actions), and then either stops or terminates successfully.

The effect of an executed communication event (I/O in this case) $ch?v$ and $ch!expr$ by a pair of processes $M$ and $M'$ respectively is the instantaneous assignment $v := expr$ in $M$ where $v$ is a local variable of $M$.

Channel names are removed from timed events with *Dropchan*.

$$Dropchan : PROCUNIT \longrightarrow PROCUNIT$$
$$\forall s : PROCUNIT \mid s \in chan \bullet$$
$$Dropchan(s) = s \backslash channel(s)$$

The function *ptime*() returns the time of a process's event occurrence. The signature of *ptime* is:

$$ptime : PROCUNIT \longrightarrow TIME$$

So a process is defined formally as:

$$PROCESS == \text{seq}\, PROCUNIT$$
$$\forall p : PROCESS \bullet$$
$$\forall i, j : 1 \,..\, \#p \bullet i < j \Rightarrow$$
$$ptime(p[i]) < ptime(p[j])$$

A process progresses only upon execution of an event. A process can be in one of three stages (commit, abort, or active) so

$$STATUS ::= Abort \mid Commit \mid Active$$

The function *GetPstatus* reports a process's status. This permits a (sub)process to enquire whether its parent or another process has committed or aborted. Its signature is given as

$$GetPstatus : Transaction\_id \longrightarrow STATUS$$

A process's behaviour is recorded in a timed trace which is a finite monotonic sequence of timed events. Formally,

$$TIMEDTRACE == \text{seq}_1\, TIMEDEVENT$$

$$\forall s : TIMEDTRACE \bullet$$
$$\forall i, j : 1 \mathinner{..} \#s \bullet i < j \Rightarrow$$
$$time(s[i]) < time(s[j])$$

For example, given the timed trace:

$$s = \langle (t_1, a_1), (t_2, a_2) \rangle \Rightarrow t_1 \leq t_2$$

Thus, a trace is given by:

$$trace : PROCESS \longrightarrow TIMEDTRACE$$

Each process's trace uniquely specifies a path leading from the start of the process's execution to that particular state. However, only one path leads to an acceptable state. Note that the trace of process before it engages in its first event (of course an empty trace) is irrelevant in this study.

The *prefix* of a trace is a continguous subsequence of the trace such that the head of the subsequence and the trace are the same. The prefix operator[6] is **prefix**, defined formally as:

$$\_ \textbf{ prefix } \_ : \text{seq} \leftrightarrow \text{seq}$$
$$\forall s, t : TIMEDTRACE \bullet$$
$$s \textbf{ prefix } t \Leftrightarrow (\exists u : TIMEDTRACE \bullet s \frown u = t)$$

An example of a trace's prefix is:

$$\langle a, d, f \rangle \textbf{ prefix } \langle a, d, f, h, k \rangle = true$$

A trace $s$ is a prefix of itself so $trace(P)$ **prefix** $trace(P)$. Thus,

$$\langle a, d, f, h, k \rangle \textbf{ prefix } \langle a, d, f, h, k \rangle = true$$

The **prefix** relation is reflexive, transitive, and antisymmetric.

[6]The operator **prefix** differs from in because in may hold for **any** continguous subsequence of any given sequence whereas **prefix** holds only when the head of **both** the subsequence and given sequence is the same.

The set of *all possible*[7] traces of a process is called *traces*:

$$traces : PROCESS \longrightarrow \mathbf{P}\ TIMEDTRACE$$

Elements of *traces* are prefixes of a *trace*, so $trace \in traces$.

The relational operator that establishes an ordering relationship between sequence elements is:

$$\_ <_S \_ : EVENTS \leftrightarrow EVENTS$$
$$\forall a, b : EVENTS \bullet$$
$$a <_S b \Leftrightarrow (\exists S : \mathrm{seq}\, EVENTS;\ i, j : \mathbb{N}_1 \mid$$
$$i \in \mathrm{dom}\, S \wedge j \in \mathrm{dom}\, S \wedge S(i) = a \wedge S(j) = b \bullet i < j)$$

This operator is necessary because the mathematical $<$ operator is undefined for the sequence data type. The relation $<_S$ is reflexive, asymmetric, and transitive. Sequence can be a history, trace or transaction. When the context of the sequence is clear one can write $<_H$ or $<_T$ to refer to the ordering in the sequence $H$ (a history) and $T$ (a transaction), respectively.

A process executes within an *interval* defined by a beginning and end time. An interval consists of all the enclosed time points. Intervals can be *closed* and *open*. An interval is open when it excludes the two endpoints; that is, their endpoints are not contained within the interval. The interval is closed when the two endpoints are contained in the interval. Thus, we can model execution overlaps between any two processes. An interval is defined as:

$$INTERVAL == \mathrm{seq}\ TIME$$

$$\forall t : INTERVAL \bullet \#t = \#(\mathrm{ran}\, t) \ \wedge \ head(t) < last(t)$$

[7] All possible traces of a process are defined by the powerset of its timed events. This allows for unforseen occurrences such as a system failure. The record of the timed events up to that point is still a timed trace.

The following defines open interval.

$$
\begin{aligned}
&\mathit{openint} : \mathit{TIME} \times \mathit{TIME} \pfun \text{seq } \mathit{TIME} \\
&\forall a, b : \mathit{TIME} \mid a < b; \mathit{int} : \text{seq } \mathit{TIME} \bullet \\
&\quad \mathit{openint}(a, b) = \mathit{int} \Leftrightarrow (\forall t_1, t_2 : \mathit{TIME} \mid \\
&\qquad\qquad (a < t_1 \land t_1 < b \land a < t_2 \land t_2 < b \land t_1 \neq t_2) \bullet \\
&\qquad\qquad\quad t_1 \leq t_2 \Rightarrow t_1 <_{\mathit{int}} t_2)
\end{aligned}
$$

Similarly, close interval is defined as follows

$$
\begin{aligned}
&\mathit{closeint} : \mathit{TIME} \times \mathit{TIME} \pfun \text{seq } \mathit{TIME} \\
&\forall a, b : \mathit{TIME}; \mathit{int} : \text{seq } \mathit{TIME} \bullet \\
&\quad \mathit{closeint}(a, b) = \mathit{int} \Leftrightarrow (\forall t_1, t_2 : \mathit{TIME} \mid \\
&\qquad\qquad (a \leq t_1 \land t_1 < b \land a < t_2 \land t_2 \leq b \land t_1 \neq t_2) \bullet \\
&\qquad\qquad\quad t_1 \leq t_2 \Rightarrow t_1 <_{\mathit{int}} t_2)
\end{aligned}
$$

The difference between *openint* and *closeint* is the inclusion of the end points in *closeint*. Partial cases where one end is closed and the other is open are intuitively derivable.

### 5.4.2 Temporal Operators

Transaction specification requires several temporal operators. This section provides those definitions and corresponding semantics.

The function **times** returns a time sequence for the events in a timed trace.

$$
\begin{aligned}
&\mathit{times} : \mathit{TIMEDTRACE} \longrightarrow \text{seq } \mathit{TIME} \\
&\forall s : \mathit{TIMEDTRACE} \bullet \\
&\quad \mathit{times}(s) = (t \mid t : \text{seq } \mathit{TIME}) \Leftrightarrow \\
&\qquad\qquad (\exists a : \mathit{EVENTS}; t_1 : \mathit{TIME} \mid t_1 \text{ in } t \bullet \langle (t_1, a) \rangle \text{ in } s)
\end{aligned}
$$

The function **events** returns an event sequence in a timed trace.

$$
\mathit{events} : \mathit{TIMEDTRACE} \longrightarrow \text{seq } \mathit{EVENTS}
$$

$$\forall s : TIMEDTRACE \bullet$$
$$events(s) = (x \mid x : \text{seq}\, EVENTS) \Leftrightarrow$$
$$(\forall i : 2 .. \#s \bullet x = event(s[1]) \frown event(s[i]) \mid$$
$$\forall i, j : 2 .. \#s \mid i \neq j \bullet i < j \Rightarrow x[i] <_x x[j]$$

The function **first** returns the first event in a sequence.

$$first(s) \mathrel{\widehat{=}} event(s[1])$$

Similarly, **last** returns the last event in a sequence.

$$last(s) \mathrel{\widehat{=}} event(s[\#s])$$

The function **begin** returns a trace's first event time. If the trace is empty a time value of **infinity** (i.e., $\infty$) is returned. The function **end** returns the last event's time or $\infty$ if the trace is empty. Formally, the above is given by:

$$\forall s : TIMEDTRACE \bullet$$
$$begin(s) \mathrel{\widehat{=}} (t \mid t : TIME \bullet (s = \langle\rangle \Rightarrow t = \infty)$$
$$\vee\ (t = time(head(s))))$$
$$end(s) \mathrel{\widehat{=}} (t \mid t : TIME \bullet (s = \langle\rangle \Rightarrow t = \infty)$$
$$\vee\ (t = time(s[\#s])))$$

If we constrain the timed trace to be nonempty, the begin and end functions can simply be defined as $begin(s) \mathrel{\widehat{=}} time(s[1])$ and $end(s) \mathrel{\widehat{=}} time(s[\#s])$, respectively.

The following additional relational operators are useful.

1. **At**

The **at** operator denotes the occurrence of an event at a particular time point. For example, $b$ **at** $t$ means event $b$ occurs at exactly time point $t$.

$$\_at\_ : EVENT \times TIME \leftrightarrow TIMEDEVENT$$
$$\forall b : EVENTS;\ t : TIME \bullet$$
$$b \textbf{ at } t \Leftrightarrow time(b) = t \wedge$$
$$(\exists s : TIMEDTRACE \bullet \langle (t, b) \rangle \text{ in } s)$$

2. **Before**

The **before** operator returns the part of a trace that describes all events occurrences prior to the referenced time point. For example, $s$ **before** $t$ means the sequence of all events in $s$ that occur at any time less than $t$. The formal definition follows:

$$\_before\_ : TIMEDTRACE \times TIME \leftrightarrow TIMEDTRACE$$

$$\forall s : TIMEDTRACE; t : TIME \bullet$$

$$s \textbf{ before } t \Leftrightarrow \forall i : 1 \,..\, \#s \bullet time(s[i]) < t$$

3. **During**

The operator **during** returns the part of a trace that occurs within some time interval $I$. The definition of **during** is:

$$\_during\_ : TIMEDTRACE \times INTERVAL \leftrightarrow TIMEDTRACE$$

$$\forall s : TIMEDTRACE; t : INTERVAL \bullet$$

$$s \textbf{ during } t \Leftrightarrow (\exists s_1 \mid s_1 \textsf{ in } s \bullet$$

$$(\forall i : 1 \,..\, \#s_1 \bullet time(s_1[i]) \textsf{ in } t)$$

4. **Occur**

The operator **Occur** returns all events occurring at a given time.

$$\_occur\_ : TIMEDTRACE \times TIME \leftrightarrow TIMEDTRACE$$

$$\forall s : TIMEDTRACE; t : TIME \bullet$$

$$s \textbf{ occur } t \Leftrightarrow (\exists s_1 \mid s_1 \textsf{ in } s \bullet$$

$$(\forall i : 1 \,..\, \#s_1 \bullet time(s_1[i]) = t)$$

5. **After**

The **after** operator returns the part of the trace that occurred immediately following the referenced time point. This operator is defined as follows:

$$\_after\_ : TIMEDTRACE \times TIME \leftrightarrow TIMEDTRACE$$

$$\forall s : TIMEDTRACE; t : TIME \bullet$$

$$s \textbf{ after } t \Leftrightarrow \forall i : 1 \,..\, \#s \bullet time(s[i]) > t$$

For example, $s$ **after** $t$ means the occurrence of all events in $s$ at any time greater than $t$.

The **before**, **after**, and **during** operators are applicable to the time interval semantics. In this case, the following intervals are naturally associated with:

$$\text{before} = (1, t]$$
$$\text{after} = [t, \infty)$$
$$\text{during} = (t_1, t_2)$$

where [ or ] denotes open and ( or ) denotes closed ends of the interval.

For example, the **at** operator can be applied to an interval $I$ as follows:

$$S \textbf{ at } I \triangleq (\exists\, a : EVENTS \mid a \textsf{ in } events(S) \bullet$$
$$\exists\, t : TIME \mid t \textsf{ in } times(S) \bullet a \textbf{ at } t \wedge t \textsf{ in } I)$$

$S$ **at** $I$ holds if and only if some elements of S occur at some time during $I$.

Similarly, we can also specify that an event or events do not occur at a particular time or a time interval. For example, $\neg$ ($a$ **at** $t$) and $\neg$ ($S$ **at** $I$) mean that no $a$ occurred at time $t$ and no element of the trace $S$ occurred within the interval $I$, respectively.

6. **Overlap**

Two intervals *overlap* when they have some points in common. Formally the **overlap** operator is defined as:

$$\_overlap\_ : INTERVAL \leftrightarrow INTERVAL$$

$$\forall\, I_1, I_2 : INTERVAL \bullet$$
$$I_1 \textbf{ overlap } I_2 \Leftrightarrow I_1 \neq I_2 \wedge (\exists\, t : TIME \bullet$$
$$t \textsf{ in } I_1 \wedge t \textsf{ in } I_2)$$

The event times of processes that simultaneously use an unshareable resource must not overlap with respect to the resource's use times. That is, if $P$ and $Q$ are processes that use a nonshareable resource, say *Res*, to adequately model the concurrent utilization of *Res*, the following constraint must hold:

$$\forall\, p : trace(P) \upharpoonright Res;\ q : trace(Q) \upharpoonright Res \mid P \neq Q \bullet \neg\, (times(p) \textbf{ overlap } times(q))$$

7. **Precedes**

**Precedes** states that one timed event occurs before another. It is defined as

$$\_precedes\_ : TIMEDEVENT \leftrightarrow TIMEDEVENT$$

$$\forall t_1, t_2 : TIME; a, b : EVENTS \bullet$$
$$(t_1, a) \textbf{ precedes } (t_2, b) \Leftrightarrow (\exists s : TIMEDTRACE \mid$$
$$\langle (t_1, a) \rangle \text{ in } s \wedge \langle (t_2, b) \rangle \text{ in } s) \bullet t_1 \leq t_2)$$

The relational temporal operator **precedes** is reflexive, asymmetric, and transitive.

8. **Duration** and **Availability**

The function **eduration** returns the duration of an event which is calculated by subtracting the start time from the end time. The signature is:

$$eduration : EVENTS \longrightarrow TIME$$

Let $\rho = \textbf{eduration}(x)$ represents the duration of the event $x$. If the event $x_i$ occurs at time $t$ then the next event $x_{i+1}$ must occur no earlier than time $t + \rho$. Thus,

$$time(x_{i+1}) = t + \rho$$

The events transition will be:

$$(x_i, t) \longrightarrow (x_{i+1}, t + \rho)$$

Thus, if $\rho$ is the minimum execution time for all events then each process will spend at least $\rho$ time in each state. The advantages of this are:

- assures that time progresses forward in an infinite trace, and
- a finite number of events are executed in a finite amount of time.

The function **available** returns true if the first event of its argument can occur and return false otherwise; e.g., $available(P) = true$ means the first event of process $P$ can occur in the present environment of $P$. The signature is:

$$available : PROCESS \longrightarrow BOOLEAN$$

A process's **duration** is the time difference between its last event and the first. Thus, the function *pduration*() returns a process's duration. Formally,

$$pduration : PROCESS \longrightarrow TIME$$

$$\forall t : TIME; P : PROCESS \bullet$$
$$\textbf{if } trace(P) = \langle\rangle \textbf{ then } pduration(P) = t \Rightarrow (t = \infty)$$
$$\textbf{else } pduration(P) = t \Rightarrow$$
$$t = time(last(trace(P))) - time(first(trace(P)))$$

## 5.5 Interleaving

The function **interleaves** defines the interleaving of two or more event sequences. In the timed trace semantics (see Section 4.4, Chapter 4), the chronological order of event times must be maintained. For example, consider the following two sequences:

$$x = \langle(1, a), (3, d), (3, g), (5, e), (6, c)\rangle$$
$$y = \langle(1, k), (2, h), (3, f), (8, l)\rangle$$

Then, $\textbf{interleaves}(x, y) = \langle(1, a), (1, k), (2, h), (3, d), (3, f), (3, g), (5, e), (6, c), (8, l)\rangle$ is one of the possible interleavings. However, the resulting interleaved sequence must uphold the order of event times. Whenever $x_i = y_j$, $x_i$ and $y_j$ appears in the interleaved sequence as a permutation[8] of $x_i$ and $y_j$. This semantics is different from Hoare's [Hoa85] definition which is arbitrary interleaving of the elements of the two sequences (see Section 4.5) because this definition incorporates time semantics. The formal definition follows:

$$interleaves : TIMEDTRACE \times TIMEDTRACE \longrightarrow TIMEDTRACE$$

[8]Specifically, let $p : S \longrightarrow S$ be a permutation of a set $S \equiv \{s_1, s_2, \ldots, s_n\}$. That is $p$ is a rearrangement of the elements $s_1, s_2, \ldots, s_n$. The interleave function is $\mathbf{O}(m + k)$ where $m + k$ is the total number of elements. This linear function is achieved by using merge sort since the two sequences are sorted.

$$
\begin{array}{l}
\forall x, y, z : TIMEDTRACE \bullet \\
\textbf{if } x = \langle\rangle \textbf{ then } interleaves(x, y) = y \\
\textbf{else if } y = \langle\rangle \textbf{ then } interleaves(x, y) = x \\
\quad \textbf{else } interleaves(x, y) = z \Leftrightarrow \#z = \#x + \#y \wedge \\
\quad\quad (\forall t_x, t_y : TIME \mid t_x \in \mathrm{ran}\ times(x) \wedge t_y \in \mathrm{ran}\ times(y) \bullet \\
\quad\quad\quad t_x \leq t_y \Rightarrow (\exists a, b : EVENTS \mid \\
\quad\quad\quad\quad \langle (t_x, a) \rangle \textbf{ in } x \wedge \langle (t_y, b) \rangle \textbf{ in } y \bullet \\
\quad\quad\quad\quad\quad (\langle (t_x, a) \rangle \textbf{ in } z \wedge \langle (t_y, b) \rangle \textbf{ in } z \wedge \\
\quad\quad\quad\quad\quad\quad (t_x, a) \textbf{ precedes } (t_y, b))))
\end{array}
$$

The elements of the interleaving function forms a total partial order and is order preserving. The relation on the elements of a partial order is reflexive, transitive, and antisymmetric.

## 5.6 Causality

Concurrency, a key feature of cooperative concurrent process problem solving algorithms, depends upon fine control over process communication and synchronization. Processes interact using messages during transactions execution. True concurrency is possible only when the concurrent processes share no variables. However, when accessing shareable data objects, concurrent processes can proceed independently until they reference variables in their common environment as illustrated in Figure 5.2. Such accesses require synchronization to guarantee correctness of the transactions.

A transaction can be *causally dependent* or *independent* of another transaction. Causality among events is the ability of one event to directly or indirectly (by transitivity) affect another.

**Definition 23** A transaction $A$ is *causally dependent* on transaction $B$, denoted as $A \xrightarrow{CD} B$ if and only if $A$ reads or uses an object written by $B$ while $B$ is still active or an event of $B$ triggers the occurrence of an event of $A$. ■

In the above definition, $A$ is called the *dependent* transaction while $B$ is called the

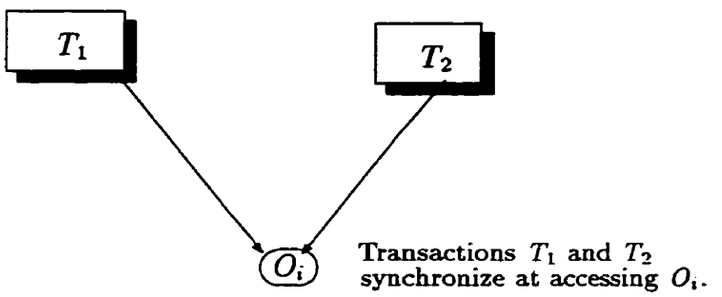


Figure 5.2: Synchronization Point of two Transactions

*depended-on* transaction[9]. This dependence would results in the abortion of the dependent transaction if the depended-on transaction fails. However, the failure of any causally independent transaction does not affect the progress of other transactions in the system. The use of an event's partial ordering as defined by their causal relationships enables efficient recovery from failures[10]. It should also be noted that if $A$ happening at time $\lambda$ causes $B$ to occur at time $\mu$ then $\lambda$ must come before $\mu$, so causality respects time.

Formally, causal dependency is defined as:

$$\_ \xrightarrow{CD} \_ : Transaction \leftrightarrow Transaction$$
$$\forall A, B : Transaction \bullet$$
$$A \xrightarrow{CD} B \Leftrightarrow (\exists x_i : Data \mid x_i \in Acset_{T_A} \wedge x_i \in Acset_{T_B} \wedge$$
$$\exists p, q : Op \mid p \in OT_A \wedge p = r(x_i) \wedge$$
$$q \in OT_B \wedge Active(T_B) \wedge q = w(x_i) \bullet$$
$$time(q) < time(p))) \qquad \text{[R2]}$$

That is, the write of object $x_i$ by transaction $B$ precedes the read operation on object $x_i$ by transaction $A$. In other words, $A$ is causally dependent on $B$ if and only if:

$$\exists r(x_i) \in OT_A \ \wedge \ \exists w(x_i) \in OT_B \wedge Active(T_B) \bullet w(x_i) \prec r(x_i)$$

[9]There are three instances of conflict : $READ \longrightarrow WRITE$, $WRITE \longrightarrow WRITE$, and $WRITE \longrightarrow READ$. Only the last one results in causal dependence.

[10]A partial ordering of a process's interactions permits mathematical analysis (induction) to prove some system's properties such as the absence of deadlock.

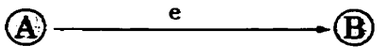


Figure 5.3: Sample Dependency Relationship Graph

There is an analogy between read/write operations and receive/send operations. Since processes interact via message passing, the send and receive message events indicate the flow of information (and some control/dependency) between the processes. This induces a causal dependency from the receiver process on the sender process. In this communication situation, the send message (signal) is treated as a write operation of the sender process while the receive message is considered as a read operation by the receiver process thereby defining causal dependencies between the pairs of corresponding send and receive events.

The causal dependency relation is transitive so:

$$A \xrightarrow{\mathcal{CD}_*} B \Leftrightarrow A \xrightarrow{\mathcal{CD}} B \ \vee$$
$$(\exists_1 B_i : \mathit{Transaction} \bullet A \xrightarrow{\mathcal{CD}} B_i \wedge B_i \xrightarrow{\mathcal{CD}_*} B) \qquad \text{[R2a]}$$

A graph showing the dependencies in a transaction's execution history is called a *dependency graph.* The transactions are the nodes and the edges are labelled by the object on which the dependency is induced. For example, there is a directed edge from $A$ to $B$ labelled $e$ if $B$ depends on $e$ generated by $A$, as illustrated in Figure 5.3. Several causal relationships are possible such as those shown in Figure 5.4 where the dashed arrows ($- - \rightarrow$) indicate causal dependencies and the solid arrows indicate a nested transaction invocation hierarchy.

To enforce correct serialization in such causally dependent situations requires the isolation of each transaction and that the transactions' execution follow a predefined partial order. Therefore, a set of tasks or operations can execute concurrently if they are causally independent of each other. To illustrate this, Figure 5.4 shows:

- $T_{23}$ depends on some values produced by $T_1$ and $T_{21}$.

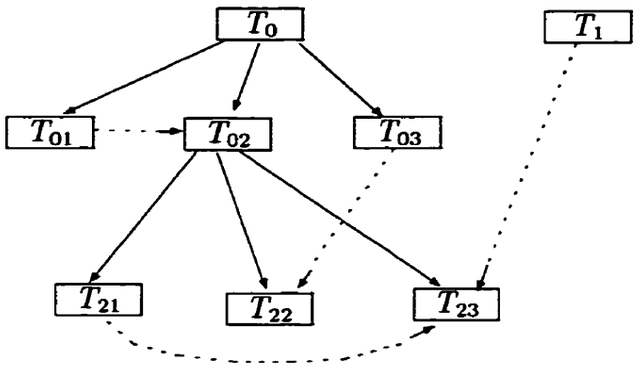


Figure 5.4: Sample Causal Dependencies

- $T_{02}$ depends on some produced by $T_{01}$.
- $T_{22}$ depends on some values produced by $T_{03}$.

In the temporal order of events, $T_0$ and $T_1$ can run concurrently but are synchronized in the temporal partial order according to the causal dependency induced at $T_{23}$. The commitment order of the transactions **must** obey the temporal order of event occurrences in both transactions to guarantee the correctness of results. Therefore, the transactions should obey the following commitment order $T_1$, $T_0$ for the correct execution of the transactions.

*Causal dependency constraints* dictate the order in which computations of causally dependent transactions must be executed relative to one another. Therefore, two transactions that are causally dependent obey the causal dependency constraint if each transaction executes events in nondecreasing timestamp order and preserves the cause-and-effect relations. The cause must always precede the effect thereby defining the sequencing constraints. Consider the concurrent execution of two events, $A$ and $B$, by transactions $T_i$ and $T_j$ respectively. If $B$ reads a variable updated by $A$, the execution of $A$ must precede $B$ to uphold causality. So, $T_i < T_j$ ensures the computation's correctness. After executing the causally dependent events, the two transactions may be independent of each other so the execution can progress out of timestamp order without violating causality

constraints. Thus, obeying the causal dependency constraint suffices to guarantee correct execution.

Fujimoto [Fuj90] states that deciding whether $A$ can execute concurrently with $B$ requires an operational simulator. This implies without actually executing the simulator, determining whether $A$ affects $B$ or not may be impossible or at least infeasible. However, this conclusion is no longer valid because by analysing the transaction's specifications statically (see [Gra94] for detail about static analysis and database operations relationships) one can determine an operation's effect on another, if any.

Determining causal dependency constraints in general is complex and highly data dependent. For example, the scenario in which $A$ affects $B$ can be a complex sequence of events depending on event timestamps. Using the causality concept to determine acceptable schedules is strictly a conservative approach to concurrency control since the approach first determines when it is safe to process an event before executing the event.

## 5.7 Specification of Transaction

A transaction $T_i$ is a sequence of read ($r_i$), write ($w_i$), and task invocation ($t_i$) operations terminated by a transaction terminating operation, $n_i \mid n_i \in \{a_i, c_i\}$ where $c_i$ is a commit and $a_i$ is an abort operation. A more rigorous definition is given as:

**Definition 24** *Transaction :* A transaction $T_i$ is a sequence of transaction operations such that:

1. the first element is a *Begin* transaction operation,
2. $\forall\, Op_i \in T_i \bullet Op_i < n_i$
3. $n_i$ occurs only once in the sequence,
4. all other elements in the sequence can be read ($r_i$), write ($w_i$), or task invocation ($t_i$) operation, and
5. a partial order ($<_{T_i}$) orders any conflicting pair of operations. ■

Formally,

$$
\begin{aligned}
&\mathit{Transaction} \mathrel{\widehat{=}} T \mid T = \mathrm{seq}_1\ \mathit{Op} \Rightarrow \\
&\quad \mathit{head}(T) = \mathbf{begin} \wedge \mathit{last}(T) = n_i \wedge \\
&\qquad (\#a = 1 \wedge \#a = \#c \wedge (a \text{ in } T \Rightarrow \neg\,(c \text{ in } T) \wedge c \text{ in } T \Rightarrow \neg\,(a \text{ in } T)) \wedge \\
&\qquad\quad (\forall p, q : \mathit{Op} \mid p, q = \mathit{Op} \setminus n_i \wedge p \text{ in } T \wedge q \text{ in } T \bullet p < n_i \wedge \\
&\qquad\qquad \mathit{Conflict}(p, q) \Rightarrow p <_T q \vee q <_T p))
\end{aligned}
$$

A permutation of non-conflicting operations in $T$ gives an instance of a transaction, say $T_i$ that has equivalent effects as $T$. This definition is essentially an operational view of a transaction[11].

A task is a granule of computation treated by the system as a unit of work to be scheduled and executed. Tasks are selected one by one to have their operations (methods) scheduled. The dependencies between the tasks in $T$ are specified by their precedence constraints; they are given by a partial order relation $<_T$ defined over $T$. Thus, $T_i <_T T_j$ if the execution of $T_j$ cannot begin until the task $T_i$ starts or terminates.

The events *begin*, *commit*, and *abort* are special events in the alphabets of every transaction and they are always present in the alphabet set of every transaction. In a multi-user transaction system, a transaction set, *TSet*, is defined as:

$TSet = \{T_1, T_2, T_3, \ldots, T_n\}$ where $n \geq 1$. Thus, $TSet = \bigcup_{i=1}^{n} T_i$

The components of a transaction can be partitioned into two mutually exclusive classes called the *vital* set and *non-vital* sets. The set of vital components, denoted *Vtset*, consists of those subtransactions or tasks that are critical to the correctness (and completion) of the transaction. Similarly, the set of non-vital components of a transaction, is denoted *Nvset*. Either set may be empty but if both are at the same time, the case is

[11]Although a transaction is defined as a sequence, it is important to note that this definition only constrains the user-visible behaviour of *an* implementation. Thus, any acceptable configuration of the transaction's events simply has to present the same behaviour to the users.

of no interest. Thus, for any transaction $i$ the following definition holds:

$$\begin{aligned} Transaction_i \mathrel{\widehat=} Vtset_i \cup Nvset_i \mid \\ Vtset_i \cap Nvset_i = \varnothing \wedge \\ (Vtset_i = \varnothing \Rightarrow Nvset_i \neq \varnothing \vee \\ Nvtset_i = \varnothing \Rightarrow Vtset_i \neq \varnothing) \end{aligned}$$

Let *Cmset* denotes the set of committed transactions, called *Commit* set. Let *compensate* denotes the function that calls a compensating transaction to annul the effects of the committed transaction for which it is defined. The semantics of the function *compensate* notes the compensation order required for the committed transactions whose effects need annulling.

## 5.8 History

A history ($H$) of the concurrent execution of a transaction set *Tset* is a sequence of the operations or events from *Tset*. The operations ordering in $H$ contains at least all orderings in *Tset*. That is, the sequence is a partial order of events that is consistent with the partial order $<_T$ of the events associated with each transaction $T$ in *Tset*. A history for a transaction set $Tset = \{T_1, T_2, T_3 \cdots T_n\}$ is:

$$\begin{aligned} HISTORY_{Tset} \mathrel{\widehat=} \mathrm{seq}\, S \Rightarrow (\forall T : Transaction \mid T \in Tset \bullet \\ (\forall i, j : \mathbb{N} \mid i, j \leq \#Tset, \\ \forall p, q : Op \mid p, q \in \bigcup_i \mathrm{ran}\, T_i \bullet \\ (p, q \ \mathrm{in}\ S \wedge p <_{T_i} q \Rightarrow p <_S q) \wedge \\ (\forall p \in T_i, q \in T_j \mid T_i \neq T_j \bullet \\ Conflict(p, q) \Rightarrow p <_S q \vee q <_S p))) \end{aligned}$$

The order of elements in $H$ is based on the order of respective elements in each $T_i$ in the set of currently executed transactions *Tset* and the order enforced by the actual execution.

Let *Tset* be a set of transactions and $H_1$ be a sequence of the transactions' concurrent execution. For an illustration, let

$$Tset = \{T_1, T_2, T_3\}$$

$$T_1 = w_1(x), w_1(y), c_1$$

$$T_2 = r_2(x), w_2(x), w_2(y), c_2$$

$$T_3 = w_3(x), w_3(y), c_3$$

$$H_1 = \langle w_1(x), r_2(x), w_2(x), w_3(x), w_2(y), w_1(y), c_1, c_2, w_3(y), c_3 \rangle$$

The relational operator $\in_H$ establishes the membership relationship between a history and a transaction. A transaction is in history if all its operations are in the history. Formally,

$$\_ \in_H \_ : Transaction \leftrightarrow HISTORY$$
$$\forall\, T : Transaction;\ H : HISTORY \bullet$$
$$T \in_H H \Leftrightarrow (\forall\, p : Op \mid p \in T \bullet p \text{ in } H)$$

All operations of $T$ are included in a history because only committed transactions exist under the prefix commit closed property. To illustrate the application of the above definition, consider the following example: $T_1 \in_H H_1$ because the operations $w_1(x)$, $w_1(y)$, and $c_1$ which belong to $T_1$ are elements of the history $H_1$. Similarly, operations of $T_2$ and $T_3$ are in $H_1$, so $T_2 \in_H H_1$ and $T_3 \in_H H_1$ hold.

The occurrences of events in a history are related by $<_H$.

$$\_ <_H \_ : EVENTS \leftrightarrow EVENTS$$
$$\forall\, a, b : EVENTS \bullet$$
$$(\exists\, H : HISTORY \mid a \text{ in } H \wedge b \text{ in } H \Rightarrow$$
$$a <_H b \Leftrightarrow time(a) \leq time(b)) \qquad \text{[R3]}$$

For example, $r_2(x) <_H w_1(y)$ in $H_1$ because the occurrence of $r_2(x)$ precedes $w_1(y)$. To make explicit the schedule instance in which the $<_H$ relationship is considered, the schedule is indicated as the subscript in the $<$ relation. For example, the above would be written as $r_2(x) <_{H_1} w_1(y)$

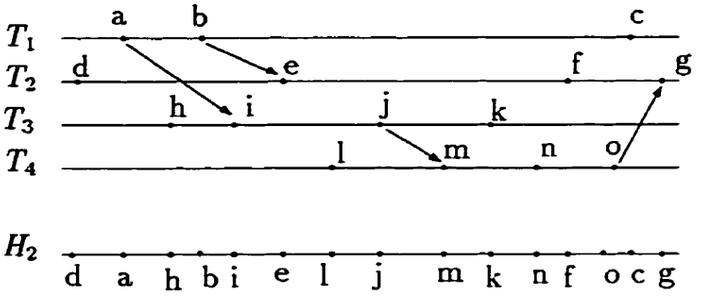


Figure 5.5: Space-Time View of Execution

A history is *complete* if it contains only completed transactions otherwise it is incomplete. An incomplete history does not represent a consistency preserving execution. For determination of correctness, however, only a complete history projected over committed transactions is useful. A projection[12] of a history $H$ on a set of transactions *Tset* is a history that contains only operations of transactions from *Tset*. A *committed projection* of a history $H$ contains only operations of committed transactions. Committed projection, denoted *ProjectCMT*, is formally defined as:

$$
\begin{array}{l}
\mathit{ProjectCMT} : \mathit{HISTORY} \times \mathit{Op} \longrightarrow \mathit{HISTORY} \\
\forall\, H : \mathit{HISTORY};\ p : \mathit{Op} \mid p = \mathit{commit} \wedge p \text{ in } H \bullet \\
\quad \mathit{ProjectCMT}(H, p) = (\exists\, H' : \mathit{HISTORY} \bullet \#H' \leq \#H \ \wedge \\
\qquad (\exists\, T : \mathit{Transaction} \mid T \in_H H; \\
\qquad\quad j : \mathbb{N}_1 \mid j \in \operatorname{dom} H \wedge H[j] = p \ \bullet \ T \in_H H' \ \wedge \\
\qquad\qquad (\forall\, i : 1..j \bullet H[i] = \mathit{commit} \wedge \mathit{commit} \in T \ \Rightarrow \ T \in_H H'))) \quad \text{[R4]}
\end{array}
$$

Since transaction execution sequences implies timing, a history is a timed trace. Each execution defines a unique sequence of events as indicated in the space-time diagram shown in Figure 5.5. In the diagram, each horizontal line is a transaction, each dot on a

[12]The projection of a history over some events $k$ is the restriction of the history to the events, written as $\mathit{trace}(T_i \restriction k)$ where $T_i$ is the transaction whose history is to be projected and $k$ is the set of events of interest. Thus, a projection is obtained by deleting all other events other than the specified event(s).

line is an event of that transaction, and an arrow indicates interaction between transactions. This indicates a causal dependency. Since events of a concurrent execution may be reordered if consistent with the causal order among the transactions and with the partial order of each transaction's execution, their permutation gives rise to an equivalent though different sequence. Tel [Tel94] establishes the equivalence of executions under reordering of non-conflicting events. For example, $H_3 = \langle h, d, a, i, l, e, b, j, m, n, k, f, e, o, g \rangle$ is a sequence different from that shown in Figure 5.5. The executions $H_3$ and $H_2$ have the same collection of events and causal order. By results established in [Tel94], $H_3$ and $H_2$ are equivalent executions.

To keep track of events and enforce the desired correctness rule, a dependency (e.g., abort or commit dependency) relationship is established whenever an invoked operation that conflicts with an active operation is allowed to execute. Therefore, the system must maintain[13] data structures to record these dependencies.

**Definition 25** *Dependency set:* The dependency set of a transaction $T$, denoted $DependS_T$, contains those transactions that developed inter-transaction dependencies with $T$ during their concurrent execution. ■

A transaction's dependency set includes all other transactions upon which it depends. The dependency relationships are used to determine the serialization order of the transactions. The dependency set of a transaction is relative to a history. Generally, inter-transaction dependencies are established during the concurrent execution of a transaction set. Such dependencies could result from the behaviour of transactions over shared data, the structural nature of the transactions, or a combination of both.

## 5.9 Specification of Correctness

The transaction models for the new application domains requires some correctness criteria to derive an appropriate concurrency control algorithm. A correctness criterion is a

[13]One way to avoid doing this is by forcing the invoking transaction to either abort or wait until the conflicting active operation terminates.

standard for judging transaction histories correctness in order to achieve a certain degree of concurrency transparency in the system. In other words, a correctness criterion is a specification of the properties which guarantees database integrity. Thus, the correctness criterion employed by a transaction system determines the acceptable transaction histories. The two most common correctness criteria based on the serializability concept, *Conflict Serializability (CSR)* and *View Serializability (VSR)*, are formally specified in this section.

To adequately model transactions' concurrency we need to guarantee both temporal and functional correctness by using mechanisms that ensure timing constraints always hold or that the transaction aborts. The temporal constraints are used to synchronize concurrent accesses to data object by multiple transactions and provide an acceptable degree of functional correctness whenever the system must abort the transaction.

Before addressing correctness formally, the following definitions are necessary.

### 5.9.1 Failure Atomicity

Failure atomicity is the all or nothing execution property of transactions. A transaction is failure atomic if and only if *all* operations invoked by the transaction either commit or abort. Formally, failure atomicity is defined as:

$$
\begin{aligned}
&\exists\, op_i : Op \mid op_i \in Transaction_i \bullet \\
&\quad commit(op_i) \text{ in } trace(Transaction_i) \Rightarrow \\
&\qquad (\forall\, op_j : Op \mid op_j \in Transaction_i \bullet \\
&\qquad\quad commit(op_j) \text{ in } trace(Transaction_i)) \vee \\
&\quad abort(op_i) \text{ in } trace(Transaction_i) \Rightarrow \\
&\qquad (\forall\, op_j : Op \mid op_j \in Transaction_i \bullet \\
&\qquad\quad abort(op_j) \text{ in } trace(Transaction_i))
\end{aligned}
\qquad \text{[R5]}
$$

When a transaction obeys the atomicity principle, the presence of a commit operation it invoked in its trace means all operations of the transaction commits. Similarly, the presence of an abort operation invoked by the transaction in its trace means that all the transaction's operations abort. Thus, the semantics of a transaction's commit and abort is based on failure atomicity.

**Lemma 1** Every atomic transaction is failure atomic.

**Proof:**

Given that $T$ is a transaction and atomic. $T$ is failure atomic if and only if $T$ satisfies specification [R5]. By [R0] and [R4] the invocation and execution of every operation is atomic. So, if

$$\exists\, p : Op \mid p \in T \bullet commit(p) \Rightarrow$$
$$\forall\, q : Op \mid q \in T \bullet commit(q) \Rightarrow \text{[R5] holds.}$$

Similarly, the same reasoning holds for an abort operation.
However, if

$$\exists\, p, q : Op \mid p, q \in T \bullet commit(p) \wedge abort(q) \Rightarrow$$
$$\text{contradiction of specification [R0]}$$
$$\text{So specification [R5] does not hold.}$$

So $T$ is failure atomic. □

In summary, for any transaction $T$,

$$commit(T) \Rightarrow trace(T) \neq \langle\,\rangle \wedge \neg abort(T) \text{ in } trace(T) \wedge \neg Active(T)$$
$$abort(T) \Rightarrow \neg commit(T) \text{ in } trace(T) \wedge \neg Active(T) \qquad \text{[R6]}$$

Conflict serializability ensures there are no cycles in the serialization graph of the transactions in the schedule. Thus, a binary relation that defines any schedule's serialization order is of the form $T_i \rightsquigarrow T_j$, where $T_i \neq T_j$. Formally,

$$\_ \rightsquigarrow \_ : Transaction \leftrightarrow Transaction$$

$$\forall\, T_i, T_j : Transaction \bullet$$
$$T_i \rightsquigarrow T_j \Leftrightarrow T_i \neq T_j \wedge$$
$$(\exists\, p, q : Op \mid p \in T_i \wedge q \in T_j \bullet$$
$$Conflict(p, q) \wedge time(p) < time(q) \wedge$$
$$(\forall\, p_i, q_j : Op \mid p_i \neq p \wedge q_j \neq q \wedge p_i \in T_i \wedge q_i \in T_j \bullet$$
$$Conflict(p_i, q_j) \Rightarrow time(p_i) < time(q_j))) \qquad \text{[R7]}$$

The transitive closure[14], denoted $\rightsquigarrow^*$, is given as:

$$T_i \rightsquigarrow^* T_j \Rightarrow T_i \rightsquigarrow T_j \vee$$
$$(\exists\, T_k : \mathit{Transaction} \bullet T_i \rightsquigarrow T_k \wedge T_k \rightsquigarrow^* T_j)$$

A schedule, $H$, consisting of a transaction set is conflict order preserving if and only if:

$$\forall\, T_i, T_j : \mathit{Transaction} \mid T_i, T_j \in_H H \bullet$$
$$T_i \rightsquigarrow^* T_j$$

### 5.9.2 Serial History

The definition of conflict serializability requires a specification of serial history and the equivalence[15] of two histories. A history $H$ is *serial* if for every two transactions $T_i$ and $T_j$ that appear in $H$, either all operations of $T_i$ appears before all operations of $T_j$ or vice versa. In a serial history there is no interleaving of the transactions' operations involved.

A set of correct transactions executed serially always produces correct results and transforms the database from one correct state to another on successful completion. Thus, most correctness criteria often use serial equivalence as a criterion[16] for the derivation of concurrency control protocols[17]. So, the concept of serial history is fundamental in defining most correctness criteria. Formally, a serial history is:

$$\mathit{Serial} : \mathit{HISTORY} \longrightarrow \mathit{BOOLEAN}$$
$$\forall\, H : \mathit{HISTORY},$$
$$\forall\, T_i, T_j : \mathit{Transaction} \mid T_i, T_j \in_H H \bullet$$
$$\mathit{Serial}(H) = \mathit{true} \Leftrightarrow (\exists\, p, q : \mathit{Op} \mid p \in T_i \wedge q \in T_j \bullet$$
$$p <_H q \Rightarrow (\forall\, r, s : \mathit{Op} \mid r \in T_i \wedge s \in T_j \bullet r <_H s)) \qquad \text{[R8]}$$

The following universal axiom holds for all serial histories.

**Axiom 1**: A set of correct transactions serially executed preserves database correctness

[14]Permissible schedules of a transaction set concurrent execution forbids cyclic transitive closures of the form $T_i \rightsquigarrow^* T_j$, where $T_i = T_j$.

[15]In general, two histories are equivalent if they have the same effects. The effects of a history are the values written by the write operations of unaborted transactions.

[16]The use of serial equivalence as a criterion for correct concurrent execution prevents the occurrence of lost updates and inconsistent retrievals.

[17]These protocols attempt to serialize transactions in their access to data items.

on successful completion.

**Proof:** Vacuous □

Serial equivalence requires all of a transaction's accesses to a particular data item be serialized with respect to accesses by other transactions. All pairs of conflicting operations of two transactions should be executed in the same order and could be said to have the same effect.

Two different transactions have the same effect as one another if the read operations return the same values and the data items have the same values at the end. The effects of transactions concurrent execution depend only on the conflicting operations relative ordering because the computational effects of executing conflicting operations depends on their relative order.

### Conflict Equivalence

Two histories are *conflict equivalent* if they are defined over the same set of transactions and the ordering of identical conflicting operations of unaborted transactions is the same.

**Definition 26** *Conflict equivalence:* Two histories $H$ and $H'$ are equivalent if:

1. $H_T = H'_T = \bigcup_{i=1}^{n} T_i$ and
2. for any conflicting operations $O_i \in T_i$ and $O_j \in T_j$ where $a_i, a_j \notin H$, if $O_i \prec_H O_j$ then $O_i \prec_{H'} O_j$ ■

A formal definition of conflict equivalence, $CE$, follows.

$$
\begin{aligned}
&CE : HISTORY \leftrightarrow HISTORY \\
&\forall H, H' : HISTORY \bullet \\
&\quad CE(H, H') \Leftrightarrow H \neq H' \wedge (\forall T_k : Transaction \bullet T_k \in_H H \Rightarrow \\
&\qquad T_k \in_{H'} H' \wedge \neg(abort(T_k) \text{ in } H) \wedge \neg(abort(T_k) \text{ in } H') \wedge \\
&\qquad\quad (\forall T_i, T_j \in_H H \mid T_i \neq T_j \bullet T_i \rightsquigarrow^* T_j \wedge \\
&\qquad\qquad (\forall p, q : Op \mid p \in T_i \wedge q \in T_j \wedge Conflict(p, q) \bullet \\
&\qquad\qquad\quad p <_H q \Rightarrow p <_{H'} q))) \qquad \text{[R9]}
\end{aligned}
$$

Since the aim of any system that supports transactions is to maximise concurrency [CDK94], transactions must synchronize their operations to avoid interference between conflicting operations. One method uses the synchronization protocol called *conflict*

*serializability* [BGH87]. Therefore, the goal of concurrency control protocols is to avoid interference/conflicts between operations in different transactions on the same data item and thereby avoid errors.

### 5.9.3 Conflict Serializability

Serial execution negates the benefits of concurrent execution but arbitrary interleaving of transactions during execution can lead to interference problems. Therefore, it is necessary to control the interleaving of transactions' operations to generate the same effects as a serial execution.

**Definition 27** A history is conflict serializable if and only if it is conflict equivalent to a serial history. ■

Using the preceding definitions ([R8] and [R9]), conflict serializability, *CSR*, is formally defined as:

$$
\begin{aligned}
&CSR : HISTORY \longrightarrow BOOLEAN \\
&\forall H : HISTORY \bullet \\
&\quad CSR(H) = true \Leftrightarrow (\exists H' : HISTORY \mid Serial(H') \bullet CE(H, H'))
\end{aligned}
\qquad \text{[R10]}
$$

By taking the committed projection of the transactions' schedule, the consistency of conflicting operations ordering in the schedule can be checked. The schedule is serializable if the ordering is consistent and equivalent to a serial schedule. Thus, given a sequence of transaction executions, if $T_i$ conflicts with $T_j$ and $T_i \rightsquigarrow T_j$ then $T_i$ will not occur after $T_j$ in the sequence. That is, a sequence of transactions of the form $T_i \rightsquigarrow T_j \rightsquigarrow T_k \rightsquigarrow T_l \rightsquigarrow \cdots \rightsquigarrow T_n$ where $T_i \neq T_n$ are the only acceptable schedules. Note that due to the upward inheritance of locks in the closed nested transactions model, it is impossible for $T_i = T_n$.

**Theorem 1** Each conflict serializable execution of a transaction set has the same effect as some serial execution of the transactions in the set. So serializable executions are correct.

**Proof:**

Let $Tset = \{T_1, T_2, \cdots, T_n\}$ be a transaction set consisting of $n$ transactions. The concurrent execution of transactions in $Tset$ is the history given by:

$$HISTORY_{Tset} = \text{seq}_1\, S \mid \text{ran}\, S \in T_i \wedge T_i \in Tset$$

Also, let

$$\exists\, p \in T_i, q \in T_j \mid T_i, T_j \in_H HISTORY_{Tset} \bullet$$
$$conflict(p, q) \Rightarrow p <_H q$$

So $T_i \rightsquigarrow T_j$ (by definition [R7])

$$\equiv T_i \longrightarrow T_j \;=\; T_i;\, T_j$$

Thus, all conflicting operations of $T_i$ must precede those of $T_j$.

Suppose $T_i \rightsquigarrow^* T_j \mid T_i = T_j$. This means that

$$T_i \rightsquigarrow T_k \rightsquigarrow T_l \rightsquigarrow \cdots \rightsquigarrow T_j \rightsquigarrow T_i \Rightarrow \neg\, T_i \rightsquigarrow^* T_j$$

This violates transitive closure property of $\rightsquigarrow$ and thus not conflict order preserving.

So only consistent conflict order preserving schedules are admitted.

Since $T_i \rightsquigarrow T_j \;\equiv\; T_i \longrightarrow T_j \;=\; T_i;\, T_j$ then each conflict serializable execution has the same effect as some serial execution. [*i*]

By (*i*) and [R9], (a) conflicting operations in any two equivalence schedules are similarly ordered and (b) both schedules contain the same operations. [*ii*]

Also, by [R10] CSR execution is conflict equivalent to a serial execution. So behaviourally,

$$CSR \text{ execution} \equiv \text{Serial execution} \qquad [iii]$$

From Axiom 1 we know that: (a) each transaction executes completely before the next, (also inferred from [*i*] above) and (b) each transaction is correct. So for (a)

$$\forall\, T_i;\, T_j : Transaction \mid T_i;\, T_j \in Tset \bullet$$
$$i \neq j \Rightarrow \forall\, p, q : Op \mid p \in T_i \wedge q \in T_j \bullet$$
$$p\ ;\ q \Rightarrow time(p) < time(q) \ \wedge$$
$$q\ ;\ p \Rightarrow time(q) < time(p)$$

So the sequential history is correct. Thus, the history is a permutation of the transactions in the history. To elucidate this further, let $s_0$ be the initial correct database state and $T_1$; $T_2$; $\cdots$; $T_n$ be the sequential execution order of the $n$ transactions in *Tset*. So $T_1$ executes against the initial state $s_0$ to generate the state $s_1$. Using the state transformation function $(s)$ (see page 103), $s_0(T_1) \longrightarrow s_1$. Since all transactions are correct and $T_1$ runs atomically, $s_1$ is consistent and correct. Thus,

$$\forall i : \mathbb{N} \mid i > 1 \bullet s_{i-1}(T_i)$$

where $s_{i-1}$ is the resulting state after the execution $T_1$; $T_2$; $\cdots$; $T_{i-1}$

Since serial execution is consistency preserving, the database maintains a consistent and correct state $s_n$ on successful completion of transaction $T_n$. So

Serial execution $\Rightarrow$ database correctness preserving

Combining [*i*] - [*iii*] with the preceding,

*CSR* execution $\equiv$ Serial execution
$\Rightarrow$ database correctness preserving
$\Rightarrow$ *CSR* execution = database correctness preserving. □

*CSR* is equivalent to any serial history that is a topological sort of the serialization orders. Since the serialization orders can have more than one topological sort, so a serializable history may be equivalent to more than one history.

### 5.9.4 View Serializability

View serializability uses the basic idea that each transaction sees the same data in a history as in an equivalent serial history. So, the functions *ReadsFrom* for the "reads from" concept, and *FinalW* for the "final write" of a data concept are required.

**Reads From**

$$ReadsFrom : Transaction \leftrightarrow Transaction$$
$$\forall T_i, T_j : Transaction \mid (\exists H : HISTORY \bullet T_i \in_H H \wedge T_j \in_H H \wedge T_i \neq T_j \Rightarrow$$

$$
\begin{aligned}
&(\mathit{ReadsFrom}(T_i, T_j)_H \Leftrightarrow \neg(\mathit{abort}(T_j) \text{ in } H) \wedge \\
&\quad (\exists p, q : Op \mid p = r(x) \wedge p \in T_i \wedge q = w(x) \wedge q \in T_j \bullet \\
&\qquad q <_H p \wedge p <_H \mathit{abort}(T_j) \wedge (\forall m : Op;\ T_m : \mathit{Transaction} \mid \\
&\qquad\quad m = w(x) \wedge m \in T_m \wedge T_m \in_H H \bullet \\
&\qquad\qquad q <_H m \wedge m <_H p \Rightarrow \mathit{abort}(T_m) <_H p))))
\end{aligned}
\qquad \text{[R11]}
$$

The above definition states that the write operation on $x$ by $T_j$ must precede the read operation by $T_i$, $T_i$ reads $x$ from $T_j$ only if $T_j$ has committed or is still active, and no unaborted transactions have modified $x$ between when $T_i$ reads $x$ and its update by $T_j$. For example, $T_2$ reads $x$ from $T_1$ in $H_1$. So, the relation $\mathit{ReadsFrom}(T_2, T_1)_{H_1}$ holds. Thus, $T_1 \rightsquigarrow T_2$.

### Final Write

The *final write* of a data item $x$ in a history $H$ is the operation $w_i(x) \in H$ such that $a_i$ is not in $H$ and for any $w_j(x) \in H$ where $i \neq j$ either $w_j(x) \prec w_i(x)$ or $a_j \in H$. Formally,

$$
\begin{aligned}
&\mathit{FinalW} : \mathit{HISTORY} \times \mathit{Data} \longrightarrow \mathit{Transaction} \\
&\forall H : \mathit{HISTORY}, x : \mathit{Data} \bullet \\
&\quad \exists T_i : \mathit{Transaction} \mid T_i \in_H H \bullet \\
&\qquad \mathit{FinalW}(H, x) = T_i \Leftrightarrow w_i(x) \in T_i \wedge \neg(\mathit{abort}(T_i) \text{ in } H) \wedge \\
&\qquad\quad (\exists T_j : \mathit{Transaction} \mid T_j \in_H H \wedge T_i \neq T_j \wedge w_j(x) \in T_j \bullet \\
&\qquad\qquad w_j(x) <_H w_i(x) \vee \mathit{abort}(T_j) \text{ in } H))
\end{aligned}
\qquad \text{[R12]}
$$

As an illustration of $\mathit{FinalW}$, consider $H_1$ (see page 127). The final write of $y$ in $H_1$ is the write operation of transaction $T_3$. So $\mathit{FinalW}(H_1, y)$ is by transaction $T_3$.

### View Equivalence

Recall that in a view serializable history each transaction sees the same data as it would have in some serial execution so the two histories are view equivalent.

**Definition 28** Two histories $H_m$ and $H_n$ are view equivalent if

1. they are defined over the same set of transactions and have the same operations,
2. for any unaborted $T_i$ and $T_j$ in $H_m$ and $H_n$ and for any $x$, if $T_i$ reads $x$ from $T_j$ in $H_m$ then $T_i$ reads $x$ from $T_j$ in $H_n$ , and

3. for each $x$, if $w_i(x)$ is the final write of $x$ in $H_m$ then it is also the final write in $H_n$. ■

The formal definition of view equivalence ( $VE$) using [R11] and [R12] is:

$$
\begin{array}{l}
VE : HISTORY \leftrightarrow HISTORY \\
\forall\, H, H' : HISTORY \bullet \\
\quad VE(H, H') \Leftrightarrow \#H = \#H' \wedge (\forall\, T_k : Transaction \bullet\ T_k \in_H H \Rightarrow T_k \in_H H' \wedge \\
\qquad (\forall\, T_i, T_j : Transaction \mid T_i \neq T_j \wedge \neg(abort(T_i)) \wedge \\
\qquad\quad \neg(abort(T_j)) \wedge T_i \in_H H \wedge T_j \in_H H \bullet \\
\qquad\qquad ReadsFrom(T_i, T_j)_H \Rightarrow ReadsFrom(T_i, T_j)_{H'}) \wedge \\
\qquad\qquad (\forall\, x : Data \mid x \text{ in } H \Rightarrow x \text{ in } H' \bullet \\
\qquad\qquad\quad FinalW(H, x) \Rightarrow FinalW(H', x)))) \qquad \text{[R13]}
\end{array}
$$

The following formally defines *View Serializability* (VSR) using [R4], [R8], and [R13]:

$$
\begin{array}{l}
VSR : HISTORY \longrightarrow BOOLEAN \\
\forall\, H : HISTORY \bullet \\
\quad VSR(H) = true \Leftrightarrow (\forall\, \psi : Op \mid \psi = commit \wedge \psi \text{ in } H; \\
\qquad\qquad H' : HISTORY \mid H' = ProjectCMT(H, \psi) \bullet \\
\qquad\qquad\quad (\exists\, H'' : HISTORY \mid \forall\, T_i : Transaction \bullet \\
\qquad\qquad\qquad T_i \in_H H' \Rightarrow T_i \in_H H'') \bullet \\
\qquad\qquad\qquad\quad Serial(H'') \wedge VE(H', H''))
\end{array}
$$

$\psi$ is used to determine the committed projection prefixes of transactions. Therefore, a history is *view serializable* if for any committed projection prefix of $H$ it is view equivalent to some serial history[18].

To illustrate how $VSR$ defined above works, consider any committed prefix $H_{1'}$ of $H_1$ (see page 127 for $H_1$).

1. If $H_{1'}$ includes $c_3$ then $H_{1'} = H_1$.

   So, $H_{1'}$ is view equivalent to the serial execution $T_1$; $T_2$; $T_3$.

[18]Two histories can be tested for state equivalences in **O**(n) time where n is the length of the histories [Cla92]. This may suggest that $VSR$ is efficient. However, if $H$ has $m$ transactions, then there are $m!$ possible serial histories. Further, if each history is of length $n$ the time will be of $\mathbf{O}(m!n^2)$. So $VSR$ is NP-complete.

2. However, if $H_{1'}$ includes $c_2$ but not $c_3$ then

$$H_{1'} = \langle w_1(x), r_2(x), w_2(x), w_2(y), w_1(y), c_1, c_2 \rangle$$

But $H_{1'}$ is **not** view equivalent to either $T_1$; $T_2$ or $T_2$; $T_1$ because

- $T_2$ reads $x$ from $T_1$ (so $T_1 \rightsquigarrow T_2$) **and**
- the final write of $y$ in $H_{1'}$ is the write operation by $T_1$ which is different in either $T_1$; $T_2$ or $T_2$; $T_1$ serial history.
  That is, $FinalW(H_{1'}, y) \neq (FinalW(H, y) \mid Serial(H))$.

Thus, the prefix commit-closed property does not hold. Hence $H_1$ is not view serializable.

## 5.10 Specification of Transaction Models

A transaction model is characterized by:

- the structure of the individual transactions allowed in the model, its *transaction structure*, (e.g., simple, flat, nested) and
- the structure of objects on which the transactions can operate, its *object structure* (e.g., simple or complex).

The combination of the objects and transaction structures determines the richness of the transaction. Running a transaction against an (complex) object may actually spawn additional transactions on component objects. This forces an implicit nesting of the transaction itself.

**Definition 29** *Spawn:* is the creation of a child transaction (process) $t$ by an executing transaction $T$ that runs concurrently with the transaction. $T$ is the parent while $t$ is the child transaction. ■

$$SPAWN(T, t) \Rightarrow \exists\, t \mid t \in child(T) \bullet Active(t) \wedge Active(T)$$

SPAWN establishes a parent-child relationship at time of execution. These family relationships are discussed in Section 5.10.2.

Identifying transaction relationships and exploiting them is key to deriving correct and reliable schedules for real-life applications. The structure of transactions delineates the models of transactions discussed below. The objects structure could be simple or complex.

### 5.10.1 Single Level Transactions

Single level transactions often adopt a synchronous and sequential execution pattern whereby one transaction is executed after the preceding transaction has completed. The transactions are assumed correct. The main goal is to achieve a serialized transaction execution while maintaining data consistency.

The flat transaction model is a degenerate case of the nested transaction model. Emphasis is on closed nested transaction model (and open nested model where necessary to illustrate the relaxation of some model properties).

### 5.10.2 Nested Transactions

Internal nodes of a nested transaction[19] can be subtransactions, database access operations, or a combination of both. So an operation that is part of transaction may itself be implemented as a transaction. Thus, a parent transaction can sequentially or concurrently execute any number of subtransactions and can perform access operations and other computations while its subtransactions are active. Although the execution of a parent's subtransactions can be concurrent, their execution should preserve the partial order defined within the transaction. Also, they should be equivalent to a sequential execution.

[19]Recall, there are two nesting types distinguished by the nature of accessibility to their modified data by other transactions. These are *closed nesting* which delays its commit and hides results, and *open nesting* which immediately commits on completion of its operations and make its results visible to all other transactions.

The localization of transaction failures means that when a subtransaction aborts, the parent receives the abort event information and then decides what to do next. For example, the parent may (1) trigger the execution of a contingency transaction that implements the same effects as the failed transaction, (2) execute a compensating transaction to recover from the failure, or (3) choose to abort completely. The children cannot successfully commit until their parent have. When a child aborts, it releases all the data objects in its access set (i.e., newly acquired locks) excluding those it inherited from its descendants.

The following functions are defined to simplify our specifications. The basic tree terminologies and definitions (see [Sch94] pages 403 - 407, 483 - 490 and [LMWF94] page 470) are used (in this thesis) in their usual way to refer to relationships between transactions.

- $child(T)$ yields the children (subtransactions) of transaction $T$.
- $Parent(T)$ returns the parent of transaction $T$.
- $Ancestors(T)$ returns the set of ancestors of $T$.
- $Descendants(T)$ returns the set of all descendants of $T$.

  Every node in the hierarchy is both an ancestor and a descendant of itself. The functions $AncestorsP$ and $DescendantsP$ return their proper ancestors and descendants sets respectively.

- $Siblings(T_i, T_j)$ returns true if $T_i$ and $T_j$ have the same immediate parent and false otherwise.

A formal definition of the function *Siblings* would be:

$$Siblings(a, b) = true \Leftrightarrow a \neq b \wedge Parent(a) = Parent(b)$$

Additional functions are defined within context as needed.

Two commit types exist in the nested transaction model. These are: (1) *internal* and (2) *external* commit. A subtransaction commits internal if its root transaction is

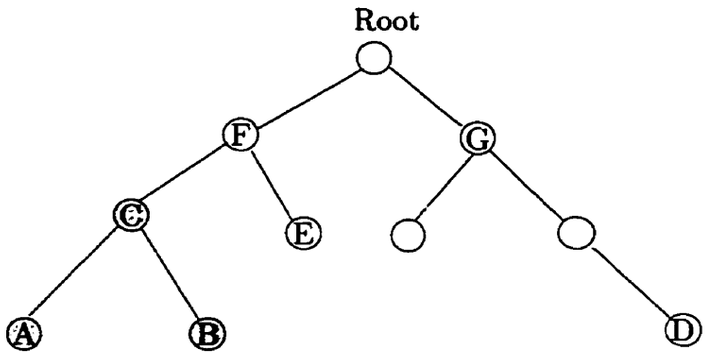


Figure 5.6: Visibility of Commit (a)

active when it commits. If a root transaction commits, it is called external commit. In Figure 5.6, the shaded and unshaded nodes represent committed transactions and active transactions, respectively. Subtransactions $A$, $B$, and $C$ internally commit. If the root transaction commits, the *whole* transaction commits externally.

Recall, each node is either a subtransaction, database access operation (read/write), or a combination of both. When a subtransaction is spawned, the subtransaction and its parent may execute concurrently. Events in the system can trigger the execution of other independent but associated transactions in the system[20]. For example, in a sales transaction the user transaction should really not wait for ledger posting operations or inventory report generation before committing successfully if all other required operations are correctly executed (or satisfied). Thus, these events are begin-dependent on certain states of the system.

The hierarchical nature of nested transactions makes visibility issues complex. Nested transactions concurrency control protocols must enforce internal commits and make them visible when appropriate. The following rules determine commit's visibility:

- External commit visibility:

  External commit is visible to every operation so every other transaction (operation) in the system sees any committed top-level transaction's operation. External

[20]However, these transactions develop some form of dependencies with the triggering transaction.

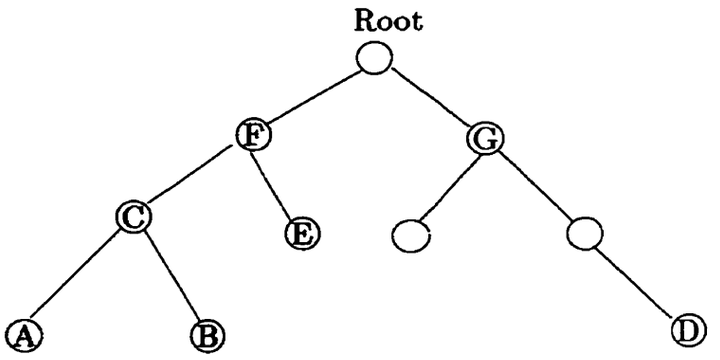


Figure 5.7: Visibility of Commit (b)

commit's visibility, denoted *visibleto*, is formally defined as:

$$
\begin{array}{l}
\_visibleto\_ : Transaction \longleftrightarrow Transaction \\
\forall\, T_i, T_k : Transaction \bullet \\
\quad T_i \textbf{ visibleto } T_k \Leftrightarrow T_i \neq T_k \wedge \\
\quad\quad AncestorsP(T_i) = \varnothing \wedge AncestorsP(T_k) = \varnothing) \wedge \\
\quad\quad\quad commit(T_i) \Rightarrow \forall\, p : OP \mid (p \in T_i \vee \\
\quad\quad\quad\quad (\exists\, T_j : Transaction \mid T_j \in DescendantsP(T_i) \bullet p \in T_j)) \bullet \\
\quad\quad\quad\quad\quad Active(T_k) \Rightarrow Vwset_{T_k'} = \{p\} \cup Vwset_{T_k})
\end{array}
\qquad \text{[R14]}
$$

- Internal commit visibility:

  Internal commit of an operation $A$ is visible to operation $B$ if and only if:

  1. $A$ and $B$ are from the same top-level transaction, and
  2. either (a) there is an ancestor of $B$ which is a sibling of some internally committed ancestor of $A$ or (b) $B$ is the parent of $A$.

  Part (a) of constraint (2) captures the case when $A$ and $B$ are siblings.

The formal definition of a transaction's committed operations *visibility* to another within a nested transaction, denoted $\blacktriangleright$, is given by:

$$
\begin{array}{l}
\_ \blacktriangleright \_ : Op \times Op \longrightarrow BOOLEAN \\
\forall\, p, q : Op \mid (\exists\, T_i, T_j, T_k : Transaction \mid T_i \neq T_j \wedge T_i \neq T_k \wedge T_j \neq T_k \wedge
\end{array}
$$

$$T_i, T_j \in DescendantsP(T_k) \wedge AncestorsP(T_k) = \emptyset \wedge$$
$$p \in T_i \wedge q \in T_j \wedge commit(T_i) \wedge Active(T_j)) \bullet$$
$$p \blacktriangleright q = true \Leftrightarrow Siblings(T_i, T_j) \vee T_j = Parent(T_i) \vee$$
$$(\exists\, T_l, T_m : Transaction \mid$$
$$T_l \neq T_m \wedge commit(T_m) \wedge T_l \in AncestorsP(T_j) \wedge$$
$$T_m \in (AncestorsP(T_i) - AncestorsP(T_j)) \bullet$$
$$Siblings(T_l, T_m) \vee Siblings(T_m, T_j)) \qquad \text{[R15]}$$

Thus,

$$p \blacktriangleright q = true \Rightarrow Vwset_{T_j'} = Vwset_{T_j} \cup \{p\}$$

The internal commit visibility ($\blacktriangleright$) is an asymmetric function but it is transitive.

A more appropriate term for subtransactions commit is *pre-commit.* When a subtransaction pre-commits, it does not mean the committment of its operations or all other operations inherited from its descendants, if any. A subtransaction's commit operation releases its access set to its parent. The parent assumes responsibility for committing all the subtransaction's operations and the subtransaction's previously inherited operations, if any. Thus, the semantics of a subtransaction's commit event indicates only the completion of all its tasks but the effects of the subtransaction's operations are reflected in the database when the root transaction to which it belongs commits.

So when a subtransaction commits, the parent inherits its access set and its effects are made visible to its parent transaction and siblings. Thus,

$$\forall c : Transaction \mid (\exists\, T : Transaction \wedge c \in child(T)) \bullet$$
$$commit(c) \Rightarrow Acset_{T'} = Acset_T \cup Acset_c \wedge$$
$$\forall n, p, q : Op \mid q \in T \wedge p \in c \wedge$$
$$(\exists\, d : Transaction \mid n \in d \wedge Siblings(d, c)) \bullet$$
$$p \blacktriangleright q \wedge p \blacktriangleright n$$

So the notion of visibility is defined with respect of one transaction to another since only a visible transaction can affect the behaviour of the active transaction that sees the modifications made. A transaction can affect another transaction through the transaction's invocations of operations down the transaction hierarchy, inheritance of access set of its

descendants, or the commit actions of the transaction and all of its ancestors up to the least common ancestors [LMWF94].

Figures 5.6 and 5.7 illustrate commit visibility. In Figure 5.6, the objects modified by the committed subtransactions $A$, $B$, and $C$ are *invisible* to subtransaction $D$ but *visible* to $E$ and $F$. However, in Figure 5.7, **all** the objects modified by the subtransactions $A$, $B$, $C$, $E$, and $F$ are *visible* to subtransaction $D$ since subtransaction $F$ is a sibling of $G$ which is an ancestor of $D$.

The semantics of nested transactions requires the effects of a committed subtransaction be visible within its root transaction and invisible to other root transactions (see specification [R15]). So two distinguishable levels of serialization exist; top-level transactions serialization with respect to each other and subtransactions serialization within a top-level transaction.

When operations from different transactions, say $T_i$ and $T_j$, concurrently access an object their serialized execution is equivalent to $T_i \longrightarrow T_j$ or $T_j \longrightarrow T_i$. The serialization order chosen must be *strictly* consistent whenever the two transactions are accessing all other objects. Thus, consistent serialization ordering of any two conflicting transactions is required to guarantee correctness.

When a root transaction aborts, all the effects of its previously committed subtransactions are annulled while all its active subtransactions are aborted. Thus,

$$\begin{aligned}
&\forall\, T : \mathit{Transaction} \mid \mathit{child}(T) \neq \varnothing \bullet \\
&\quad \mathit{abort}(T) \Rightarrow (\forall\, c : \mathit{Transaction} \mid c \in \mathit{child}(T) \bullet \\
&\qquad \mathit{commit}(c) \Rightarrow \mathit{Compensate}(c) \wedge \\
&\qquad\quad \mathit{Active}(c) \Rightarrow \mathit{abort}(c))
\end{aligned}$$

So all provisionally committed descendants of an aborted (sub)transaction are aborted.

If a vital child aborts the parent must abort[21]. So formally

$$\begin{aligned}
&\exists\, c : \mathit{Transaction} \mid (\exists\, T : \mathit{Transaction} \bullet c \in \mathit{descendant}(T) \wedge c \in \mathit{Vset}_T) \bullet \\
&\qquad \mathit{abort}(c) \Rightarrow \mathit{abort}(T)
\end{aligned}$$

[21]The abort of a vital child (even if it has provisionally committed) can come about due to the abortion of the (sub)transaction for which it is a child

Also, orphan[22] transactions are not allowed to commit. The no orphan commit rule for closed nested transactions states that every child transaction terminates if its parent terminates. Formally,

$$\forall T_i, T_p : Transaction \mid Parent(T_i) = T_p \bullet$$
$$(commit(T_p) \vee abort(T_p)) \wedge Active(T_i) \Rightarrow abort(T_i)$$

That is, any active child aborts whenever the parent aborts or commits. The parent does not commit before its child terminates. In Figure 5.8, $T_j$ must abort if it is still active whenever its parent, $T_p$, terminates before it does.

This strict rule, however, can be relaxed in some applications where some operations may continue to run beyond the termination (commit, in this case) of the parent. For example, a transaction may triggers the execution of another transaction (e.g., the production of a sales report might be initiated while the purchase transaction that initiates it completes successfully). This case can be implemented as an independent transaction with a commit/begin dependency established.

Another property of nested transactions is that it localizes failures by allowing a subtransaction to abort independently without causing the whole transaction to abort. Further, if $T_c$ is a child of $T$ such that $p$ and $q$ are two conflicting operations, if $T_c$ invokes $p$ and $Active(p)$ then $T$ cannot invoke $q$ until $p$ commits. That is, formally

$$\forall T_c, T : Transaction \mid T_c = child(T);$$
$$x : Data \mid x \in Acset_T \cap Acset_{T_c} \bullet$$
$$\forall p, q : Op \mid p \in T_c \wedge q \in T \wedge Conflict(p, q) \bullet$$
$$p <_T q \Leftrightarrow commit(p) <_T q$$

In other words, if a child's operation on an object is active, the parent cannot invoke a conflicting operation on that object; the parent must wait until the child commits.

The conflict set of a nested subtransaction excludes all the operations performed by its ancestors. It includes only active operations of its descendants (and other independent active transaction's operations with some common data items in their access sets). The

[22]An orphan transaction is an active subtransaction whose parent has terminated.

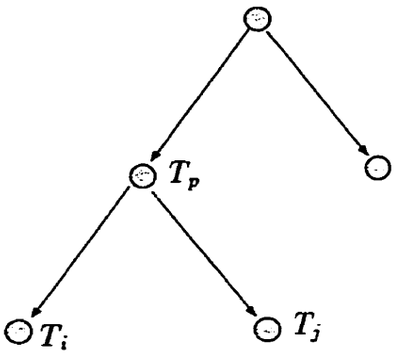


Figure 5.8: Conflict Set of Subtransaction

conflict set of $T_p$ (see Figure 5.8) is given by:

$$ConflictS_{T_p} = (\bigcup_{i=1} Acset_{T_i} \mid T_i \in Descendants(T_p) \wedge Active(T_i))$$
$$\cup \{T_j : Transaction \mid T_j \notin Ancestors(T_p) \wedge T_j \notin DescendantsP(T_p) \bullet$$
$$Active(T_j) \wedge Acset_{T_p} \cap Acset_{T_j} \neq \emptyset\}$$

A short hand form is to exclude the complement of the above expression from the conflict set. This gives:

$$ConflictS_T = \{T_i : Transaction \mid T_i \notin Ancestor(T) \bullet$$
$$Active(T_i) \wedge Acset_T \cap Acset_{T_i} \neq \emptyset\}$$

Thus, a subtransaction can access any data object currently accessed by one of its ancestors without causing any conflicts. Also, a subtransaction can see the changes made by the parent to those data objects.

If a transaction is causally dependent on its parent, the subtransaction cannot commit unless its parent does. Such a subtransaction commits if and only if the parent transaction commits.

$$\forall T_i, T_j : Transaction \mid T_i \in child(T_j) \wedge T_i \xrightarrow{CD} T_j \bullet$$
$$commit(T_i) \Leftrightarrow commit(T_j)$$

Recall too that a nested transaction $T$ cannot commit unless all its vital children commit. Formally,

$$\forall c : child(T) \mid c \in Vtset_T \bullet commit(T) \Rightarrow \bigcup_{i=1}^{n} commit(c_i)$$

All the changes made in the database by $T$'s committed descendants are made durable in the database only when the root transaction commits.

In general, a nested transaction $T$ cannot commit unless all its children commit or abort. Formally,

$$\forall T : Transaction \mid child(T) \neq \varnothing \bullet$$
$$commit(T) \Rightarrow \forall c : Transaction \mid c \in child(T) \bullet$$
$$commit(c) \vee abort(c)$$

The set of objects accessible to $T$ and its subtransactions are given by:

$$\forall T \in (Vtset_T \cup Nvset_T) \bullet Vwset_T = \bigcup_{i=1}^{m} Local\_Db_i$$

$$\forall c : child(T) \bullet Vwset_c = Acset_T \cup \{\bigcup_{i=1}^{n} Local\_Db_i\}$$

The view set of a subtransaction is the most recent state of objects in the database.

### Summary

The properties of *closed* nested transaction are summarized below:

- A set of committed nested transactions is serializable due to the semantics of atomic objects. That is, the effects of **all** root transactions in the set are committed in a serializable fashion in the database.
- Operations are committed only by root transactions because of
  - the semantics of a subtransaction commit, and
  - the no orphan commit principle.
- If a root transaction $t_0$ aborts, all operations performed by $t_0$ and its descendants abort because

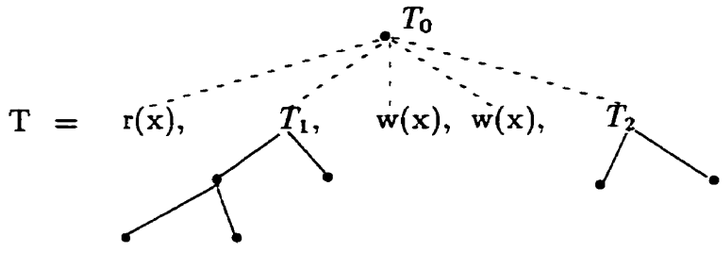


Figure 5.9: Root Transaction Construction

- no orphan commit principle,
- failure atomicity property of the root transaction, and
- quasi-failure atomicity property of subtransactions.

It is possible to transform a given transaction into an explicit nested transaction.

**Lemma 2** Given a transaction $T = \text{seq}_1\, OP \mid OP \in \{Op, T_i\}$ where $T_i$ is a task invocation operation, construct a transaction $T_0$ such that the children of $T_0$ are the elements of $T$ (i.e., $child(T_0) = \{T_c \mid T_c \text{ in } T\}$). Then

1. $T_0$ is a properly formed transaction.
2. $T_0 \equiv T$.

**Proof**

See Figure 5.9 for an illustration of a root transaction's construction. Since a parent can concurrently run any number of subtransactions $T_1$ and $T_2$ can execute concurrently synchronizing only on accesses to common data objects while maintaining the consistency of the partial order. So,

$$T = |||_{c=1}^{\#OP} T_c \wedge (\forall k, j \in \text{dom } OP \bullet Conflict(T_j, T_k) \Rightarrow T_j <_T T_k \vee T_j <_T T_k).$$

Therefore, $T$ is a nested transaction if

$$\exists\, T_0 : \mathit{Transaction} \mid \mathit{AncestorsP}(T_0) = \varnothing \wedge$$
$$\mathit{child}(T_0) = \{T_i \mid T_i \textbf{ in } T\} \bullet$$
$$\mathit{commit}(T) \Leftrightarrow \mathit{commit}(T_0) \wedge$$
$$\mathit{commit}(T_0) \Rightarrow (\forall\, c : \mathit{Transaction} \mid c \in \mathit{DescendantsP}(T_0) \bullet$$
$$\mathit{commit}(c) \vee \mathit{abort}(c) \wedge (c \in \mathit{Vset}_T \Rightarrow \neg\, \mathit{abort}(c))) \qquad \square$$

In the open nested transaction environment a subtransaction can access partial results of their ancestors, partial results of committed siblings, as well as any committed independent transaction's output[23]. The effects of committed subtransactions are visible to the parents and other transactions in the system. Thus, at commit point, the child releases all the data items in its access set. Formally,

$$\forall\, c : \mathit{child}(T) \bullet$$
$$\mathit{commit}(c) \Rightarrow \mathit{WS}_c \cup \{\bigcup_{i=1}^{n} \mathit{Local\_Db}_i\} \;\wedge\; \mathit{Acset}_c = \{\,\} \;\wedge$$
$$\forall\, T_i : \mathit{Transaction} \mid T_i \in \{\bigcup_{i=1}^{n} \mathit{Local\_Db}_i\} \bullet$$
$$\mathit{Vwset}'_{T_i} = \mathit{Vwset}_{T_i} \cup \mathit{WS}_c$$

Since other (sub)transactions can view partial results, cascading aborts may occur when the committed subtransaction's parent aborts[24]. Recall, when a transaction fails, it may try an alternative plan, the contingency transaction, made for it or it may decide to abort. If there are committed components of the transaction, they are all rolled back on abort of the root transaction. Rollback action may use compensating transaction or restore changed values using their before images. A contingency transaction does not execute if its associated original transaction commits. Thus, the begin operation of the contingency transaction is causally dependent on the abort operation of the original transaction (depended-on transaction). The abort of a contingency transaction is independent of its

[23]Recall, in open nesting once a subtransaction commits, its results are recorded in the database. Thus, the results become visible to other transactions.

[24]The effects of the committed subtransaction must be annulled from the database.

original transaction's abort. Also, a contingency transaction have the same view set as the original transaction. Formally,

$$\forall\, T_i : \mathit{Transaction} \bullet \mathit{abort}(T_i) \Rightarrow$$
$$(\exists\, Cp : \mathit{Transaction} \mid Cp \in \mathit{conTp}(T_i) \bullet$$
$$\mathit{begin}(Cp) \wedge \mathit{Vwset}_{Cp} = \mathit{Vwset}_{T_i})$$

where the function $\mathit{conTp}(T_i)$ yields the set of contingency transactions of $T_i$. Also, the following definition always hold.

$$\mathit{commit}(T_i) \Rightarrow \neg\ \mathit{begin}(Cp) \mid Cp \in \mathit{conTp}(T_i).$$

In other words, the following relation holds for contingency transactions.

$$\mathit{begin}(Cp) \xrightarrow{\mathcal{CD}} \mathit{abort}(T)$$

Similarly, compensating transactions may never start unless the corresponding transaction associated with it previously commits, so

$$\forall\, T_i : \mathit{Transaction} \bullet$$
$$\exists\, Cp : \mathit{Transaction} \mid Cp \in \mathit{comp}(T_i) \bullet \mathit{begin}(Cp) \Leftrightarrow \mathit{commit}(T_i)$$

where the function $\mathit{comp}(T_i)$ yields the set of compensating transactions of $T_i$. Thus,

$$\mathit{begin}(Cp) \xrightarrow{\mathcal{CD}} \mathit{commit}(T_i) \text{ and}$$
$$\mathit{abort}(T_i) \Rightarrow \neg\ \mathit{begin}(Cp) \mid Cp \in \mathit{comp}(T_i).$$

always hold.

### 5.10.3 Distributed Transaction

In the distributed transaction model, a transaction is decomposed into a set of components that are executed at different nodes in the network. Thus, decomposing a transaction into subtransactions that can execute at different sites, access different parts of the database, or provide information independent of other operations but logically required for the transaction's correct completion enhances intra-transaction concurrency.

Distributed transactions exhibit a correctness behaviour called *setwise* failure atomicity in that an element of the set commits if and only if every element commits. Distributed transactions are characterized by:

- Intolerance of any subtransaction failure. Abortion of a subtransaction causes the transaction to abort.
- Subtransactions are failure atomic.
- Subtransactions can commit only if the distributed transaction to which they belong commits.

Thus, if $T = \{T_1, T_2 \cdots T_n\}$ is a user's global transaction, then $T$'s definition that captures the above characteristics is:

$$T = |||_{i=1}^{n} T_i \ \wedge$$
$$commit(T) \Leftrightarrow \forall\, T_i \mid T_i \in child(T) \bullet commit(T_i) \ \wedge$$
$$abort(T) \Leftrightarrow \exists\, T_i : Transaction \mid T_i \in child(T) \vee T_i \in DescendantsP(T) \bullet$$
$$abort(T_i) \Rightarrow abort(Parent(T_i))$$

A variant of distributed transaction that is more flexible than the previous definition exist. In this variant, the transaction aborts only when any of its vital component aborts. This model has more practical utility. It is formally defined as:

$$T = |||_{i=1}^{n} T_i \ \wedge$$
$$\exists\, T_k \in Vtset_T \bullet abort(T_k) \Rightarrow abort(T)$$

where $T_i$ and $T_k$ are component transactions. Thus, the transaction has a commit dependency on all its active vital component transactions. This notion of commit dependency applies recursively to nested transactions. Therefore, the parent or main transaction aborts if any of the vital component transactions aborts. So,

$$commit(T) \Leftrightarrow \forall\, T_i \mid T_i \in child(T) \wedge T_i \in Vtset_T \bullet commit(T_i)$$
$$abort(T) \Leftrightarrow \exists\, T_i : Transaction \mid T_i \in Vtset_T \wedge$$
$$T_i \in child(T) \vee T_i \in DescendantsP(T) \bullet$$
$$abort(T_i) \Rightarrow abort(Parent(T_i))$$

Recall, when a global transaction aborts, all active component transactions are aborted and all committed components are compensated. Thus,

$$abort(T) \Rightarrow \forall\, T_i \in (Vtset_T \cup Nvset_T) \bullet |||_{i=1}^{n}\ abort(T_i) \wedge (\forall\, T_k \in Cmset \mid T_k \in Child(T) \bullet compensate(T_k))$$

For any two transactions $T_i$ and $T_k$, let their common data objects accessed be: $A = Acset(T_i) \cap Acset(T_k)$. If $A \neq \varnothing$ then $T_i$ and $T_k$ can only engage in synchronizable concurrent executions. Formally,

$$\forall\, T_i, T_k : Transaction \bullet T_i \,||\, T_k \Rightarrow (A = \varnothing \Rightarrow T_i \,|||\, T_k) \vee (A \neq \varnothing \Rightarrow T_i \,\|_A\, T_k)$$

In other words, the concurrent execution of $T_i$ and $T_k$ is captured by:

$$T_i \,||\, T_k \Rightarrow T_i \,|||\, T_k \ \lessdot A = \varnothing \gtrdot\ T_i \|_A T_k$$

Suppose $T_{23}$ is a vital component of $T_0$ in Figure 5.4 and if $T_1$ aborts, $T_{23}$ must abort because of the causal dependency relationship between $T_1$ and $T_{23}$. Therefore, $T_0$ cannot commit because of the cascading effects of the causal dependence relation between $T_1$ and $T_{23}$ via $T_{21}$. Generally,

$$abort(T_i) \text{ in } trace(T_i) \Leftrightarrow \neg\ (commit(T_i) \text{ in } trace(T_i))$$

### 5.10.4 Multidatabase

#### Assumptions

1. The MDBMS may contain intelligent agents that maintain a knowledge base of the available services at the sites.

2. Each transaction (local or global) can execute fully, either aborting or committing at termination of its operations.

3. Heterogeneity of data models and concurrency control protocols of the different LDBs is possible.

4. Global transaction are decomposable into subtransactions.

5. A transaction can be composed of read and write operations as well as other transactions. Thus, global subtransaction can be nested.

6. Two or more subtransactions of a global transaction can execute at a single site.

7. Data objects can be fragmented and distributed across multiple LDBs. Thus, replication of data is prohibited.

8. Assumes a reliable system of hardware and software components.

9. Reliable communication between the GTM and the LDBs. Multicast communication method from the GTM to the LDBs.

10. Value dependencies are possible so the action of a global subtransaction at one site can affect the behaviour of another executing at a different site.

Communication between LDBs is passes through the MDBMS's communication layer. The following are possible scenarios.

- Case 1
  - One subtransaction of a GT per LDB.
  - The subtransactions are not nested.
- Case 2
  - Multiple subtransactions of a GT can execute on a single LDB.
  - The subtransactions are not nested.
- Case 3
  - Multiple subtransactions of a GT can execute on a single LDB.
  - Nested subtransactions are permissible.

An example of case 3 above can manifest in the electronic shopping mall case study. In the purchase example, the process that accepts payment from a customer may have to access Visa and/or Mastercard account LDB as well as other financial information that the customer may provide. A possible SQL-like statement[25] for the purchase transaction

[25]This is not "pure" SQL statement. The semantics of the CONSTRAINT part of the statement simply list additional constraints that must be satisfied before the statement can successfully complete.

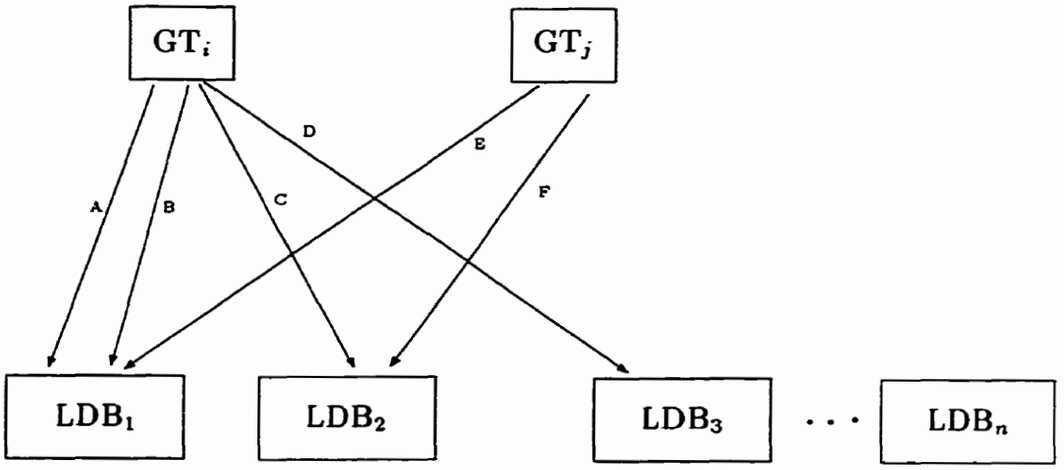


Figure 5.10: Multi-database Transactions

might be of the form:

SELECT **bicycle** FROM products

WHERE colour = green AND model = 1996 AND price $\leq$ \$60.00

CONSTRAINT 1. payment_method = Visa AND Mastercard

2. $Visa.amount <$ \$32.00

3. delivery_time $\leq$ 5 days.

It is noteworthy that the MDB is a collection of LDBs located at different sites $i$ where $i \geq 2$. Thus,

$$MDB = \bigcup_{i=1} LDB_i \quad \text{and} \quad Vwset_{MDB} = \bigcup_{i=1} Vwset_{LDB_i}$$

The *Acset* of a global transaction $GT$ consisting of $GT_i$ subtransactions is given by

$$Acset_{GT} = \bigcup_{i=1} Acset_{GT_i}$$

If multiple subtransactions of a GT can be executed at a site, the subtransactions must be serialized at that site with respect to the GT and other independent transactions at that site. Indirect conflicts may occur among global transactions because of the existence of local transactions. Finally, all the subtransactions of the GT are serialized according

the defined partial order of the subtransactions. In other words, determining correctness of MDB transactions follows two basic steps, namely: (*i*). serialize the execution of the subtransactions with other independent transactions at each LBD site, and (*ii*). serialize the global transactions.

**Notes**

- Multidatabase correctness protocol is called quasi-serializability (QSR) [DE89] or multidatabase serializability [BO90, Bar90]. These correctness criteria are equivalent. A set of local and global transactions is *quasi-serializability* (QSR) if :

  1. All executions at a site are conflict serializable with respect to that site **and**
  2. All the global transactions are conflict serializable with respect to one another.

  That is, the execution of the set of transactions at each site is serializable and the transactions have the same serialization orders at all the sites.

- All transactions executing at $SITE_i$ belong to that site, so if $T_i$ executes at $SITE_i$, then $T_i \in SITE_i$.

- Let $G$ represent the set of all global transactions. Each global transaction in $G$ is subscripted to uniquely identify it, so $G_i$ is a global transaction in the set $G$. Thus, $G_i \in G$.

- Usually, a transaction is either local or a subtransaction of a particular $G_i$. If $GST_{ij}$ is a subtransaction of $G_i$ we can write $GST_{ij} \in G_i$.

- Similarly, let $L_i$ be the set of transactions executing at $SITE_i$. Thus, $L_i$ includes global subtransactions executing at that site and the local transactions initiated by local users at $SITE_i$.

$$L_i = \{T : \textit{Transaction} \mid T \in (\bigcup_i GST_{ij} \mid GST_{ij} \in SITE_i \ \vee \ \bigcup_i LT_i \mid LT_i \in SITE_i)\}$$

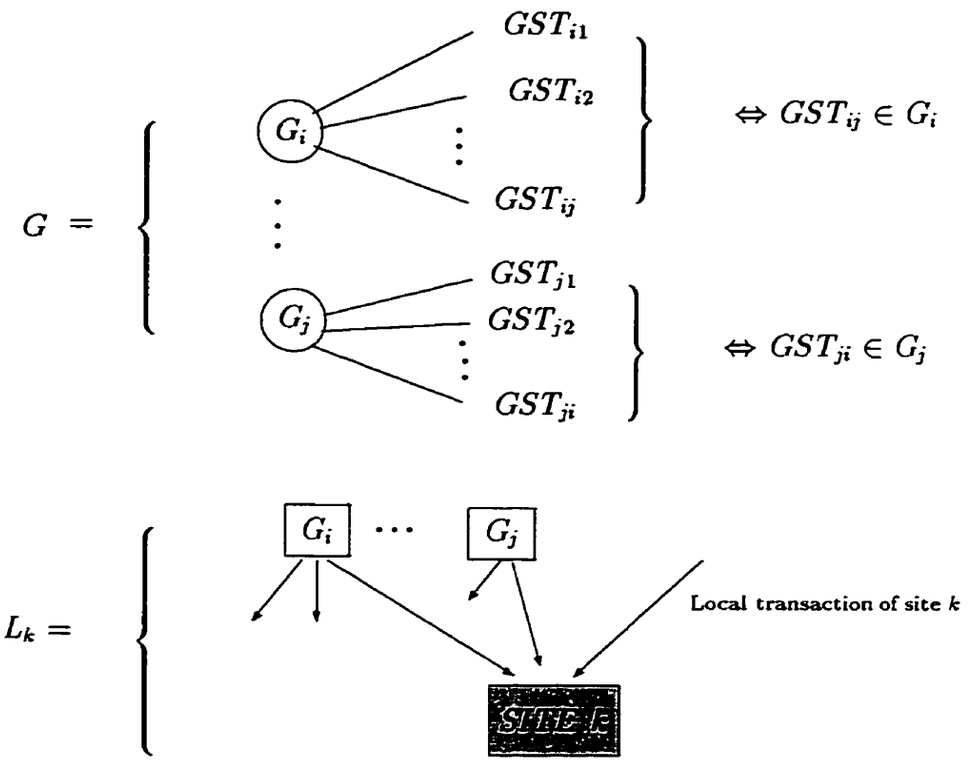


Figure 5.11: Relationship between transactions and a site

Therefore, $SITE_k$'s schedule is a sequence of local and global transactions operations executing at site $k$. The serialization order at $SITE_i$ is defined as:

$$\forall i, : \mathbb{N};\ T : Transaction \mid T \in L_i \bullet$$
$$\exists H : HISTORY \mid T \in_H H \bullet commit(T) \text{ in } H \Rightarrow \neg(T \rightsquigarrow^* T) \qquad \text{[R16]}$$

The above descriptions are sketched in Figure 5.11.

- Define the *serialization order* (**Gso**) between any two global transactions as follows:

$$\_ \textbf{Gso} \_ : Transaction \leftrightarrow Transaction$$
$$\forall T_m, T_n : Transaction \mid T_m, T_n \in G \bullet$$
$$T_m \textbf{ Gso } T_n \Leftrightarrow T_m \neq T_n \wedge T_m \rightsquigarrow T_n$$

- Global serialization order consistency must be maintained whenever subtransactions from different global transactions invoke conflicting operations on a common

data object. Thus, **Gso** must have the transitive closure property. Transitive closure, **Gso**$^*$, is defined as:

$$T_m \ \mathbf{Gso}^* \ T_n \Rightarrow T_m \ \mathbf{Gso} \ T_n \vee$$
$$(\exists \ T_o : \mathit{Transaction} \mid T_o \in G \bullet T_m \ \mathbf{Gso} \ T_o \wedge T_o \ \mathbf{Gso}^* \ T_n)$$

To guarantee the consistency of the global transactions serialization orders, if $T_i$ precedes $T_j$ at *some* site, $T_i$ must precede $T_j$ at *all* other sites and in the global commit order[26]. $T_i$ precedes $T_j$ at $SITE_i$ if a task of $T_j$ invokes some conflicting operations after the site has received the commit message from $T_i$.

Note that if an operation of $G_i$ precedes $G_j$ at some site, say $SITE_k$, and the operations conflict then for all other sites every conflicting operation of $G_i$ **must** precede $G_j$ to maintain consistent ordering of the global transactions. Formally,

$$\exists \mathit{SITE}_i \mid \mathit{SITE}_i \in \mathit{LDBs} \bullet$$
$$G_i \ \mathbf{Gso} \ G_j \text{ at } \mathit{SITE}_i \Rightarrow$$
$$\forall \mathit{SITE}_n \mid n = 1 \cdots \#\mathit{SITES} \bullet G_i \ \mathbf{Gso}^* \ G_j \qquad \text{[R17]}$$

- The serialization of global transactions is given by:

$$\forall \ T : \mathit{Transactions} \mid T \in G \bullet$$
$$\exists \ H : \mathit{HISTORY} \mid T \in_H H \bullet \mathit{commit}(T) \text{ in } H \Rightarrow$$
$$\neg(T \ \mathbf{Gso}^* \ T) \ \wedge \text{ specification [R17]} \qquad \text{[R18]}$$

Therefore, a set of local and global transactions is *quasi-serializability* (QSR) if and only if specifications [R16] and [R18] hold.

## 5.11 Specification of Concurrency

This section presents the formal specification of the basic *timestamp ordering* (TO) and the 2PL locking concurrency control protocols. The main objective here is to demonstrate the application and usefulness of the $\oplus$ and . extensions to CSP. The specifications

[26]This holds for conflicting sets that must provide identical local serialization orders. See definition of $\rightsquigarrow$. It requires that all subtransactions appear in the same order in the equivalent serial schedule at all sites. This constraint is necessary because indirect conflicts may occur among the *GSTs* at a site due to their conflicts with local transactions.

**Algorithm 1**: *Timestamp Ordering*

```
Begin BASICTO;
   CASE of operation:
        WriteItem(T,x) operation:
        IF readTS(x) > ts(T) or writeTS(x) > ts(T)
             THEN abort and rollback T
        ELSE writeItem(T,x); writeTS(x) := ts(T)
   ReadItem(T,x) operation:
        IF writeTS(x) > ts(T)
             THEN abort and rollback T
        ELSE begin
                  readItem(x);
                  readTS(x) := max(ts(T), current readTS(x))
             end{begin}
        end{ELSE}
   end{CASE}
end{BASICTO}
```

Figure 5.12: Timestamp Ordering protocol

also illustrate the benefits of using a formal methodology by enabling the comparison (or possible derivation of equivalence) between schedules the protocols admit. These protocols were chosen because they are understandably the most popular synchronization protocols and often found in most experimental and commercial transaction system implementations.

### 5.11.1 Time Stamp Ordering Protocol

The basic TO algorithm is first represented in pseudocode to expound its structure. The pseudocode is then transformed to an equivalent CSP specifications.

Figure 5.12 represents the general Timestamp protocol in pseudocode notation. Each operation is bound to a specific transaction so $writeItem(T, x)$ means the write of data item $x$ operation performed by transaction $T$. Recall, conflicting operations from distinct transactions get scheduled (or aborted) based on their timestamp values. The TO protocol checks conflicting operation occurrences that arrive in the wrong order and rejects those with lower timestamps by aborting them.

The following definitions are necessary to specify the protocol.

**Timestamp:**

Each transaction is associated with a unique time value indicating when the transaction was submitted to the system. The timestamps are ordered based on the transactions' submission order. Hence, if transaction $T_1$ starts before transaction $T_2$, then $ts(T_1) < ts(T_2)$. The older transaction has the smaller timestamp value.

Let the function $now()$ return the next available system time, and $time_T$ represents transaction $T$'s timestamp.

The function $submit()$ assigns a timestamp value to the transaction in its argument.

$$submit : Transaction \longrightarrow Transaction$$

$$\forall t : Transaction \bullet submit(t) = time_t \Rightarrow time_t = now()$$

The function $ts()$ returns the timestamp of a transaction.

$$ts : Transaction \longrightarrow TIME$$

$$\forall t : Transaction \bullet ts(t) = time_t$$

The function $sameT$ checks the operations and timestamps of any two transactions to see if the transactions are the same. Its definition is:

$$sameT : Transaction \times Transaction \longrightarrow BOOLEAN$$

$$\forall T_i, T_j : Transaction \bullet$$

$$sameT(T_i, T_j) = true \Leftrightarrow ts(T_i) = ts(T_j) \wedge$$

$$\forall p, q : Op \bullet p \in T_i \Rightarrow p \in T_j \wedge p <_{T_i} q \Rightarrow p <_{T_j} q$$

Therefore, the definition

$$\forall t_1, t_2 : Transaction \bullet$$

$$t_1 \neq t_2 \Leftrightarrow sameT(t_1, t_2) = true$$

provides a global constraints for differentiating transactions that may have different timestamps but the same operations. This situation arises when a transaction cannot progress and is resubmitted for processing.

All operations of a transaction are assigned the same timestamp as the transaction. The following axioms define timestamp's increasing monotonicity property.

1. $\forall T_i, T_j, T_k : Transaction \bullet$

   (a). $submit(T_i) < submit(T_j) \Rightarrow ts(T_i) < ts(T_j)$

(b). $ts(T_i) \leq ts(T_j) \wedge ts(T_j) \leq ts(T_k) \Rightarrow ts(T_i) \leq ts(T_k)$

2. For any given scheduler, no two transactions have the same timestamp. Formally,
$\forall T_i : T \bullet ts(T_i) = time_{T_i} \Rightarrow \neg (\exists T_j : T \bullet ts(T_j) = time_{T_j} \Rightarrow time_{T_i})$

**Data item:**

A data item $X$ is a 4-tuple, $X = (id, val, tsr, tsw)$ where $id$ is a unique identifier of the data item, $val \in$ dom $TYPE$[27] is the value (the information content) of the data item, $tsr \in TIME$ is the timestamp value of $X$'s last read operation, and $tsw \in TIME$ is the timestamp value of $X$'s last write operation. The data record is defined using the $\oplus$ operator as:

$$\oplus[DATA] == \langle\langle id, val, tsr, twr \rangle\rangle \&\& \langle\langle ID, TYPE, TIME, TIME \rangle\rangle$$

Note that $id$ is a unique system managed immutable identifier for the data item.

The following functions are useful for manipulating the components of a variable of data type *DATA*:

The function *readTS* returns a value $tsread \in TIME$ that is the stored value of the $tsr$ component of the data item $x$.

$$readTS : DATA \longrightarrow TIME$$

$$\forall x : DATA \bullet$$

$$readTS(x) = tsread : TIME \mid tsread = Val(x.tsr)$$

Similarly, the function *writeTS* returns a value $tswrite \in TIME$ that is the value of the $tsw$ component of $x$. Formally, *writeTS* is defined as:

$$writeTS : DATA \longrightarrow TIME$$

$$\forall x : DATA \bullet$$

$$writeTS(x) = tswrite : TIME \mid tswrite = Val(x.tsw)$$

The function $readItem(T, x)$ perform a read of data item $x$ by transaction $T$.

$$readItem : Transaction \times DATA \longrightarrow TYPE$$

$$\forall x : DATA; t : Transaction \bullet$$

$$readItem(t, x) = valc : TYPE \mid valc = Val(x.val)$$

Clearly, *readItem* returns the $val$ component of $x$. Similarly, an operation of transaction

[27] $TYPE$ is user defined depending on the application.

$T$ to write an item $x$ is given by:

$$writeItem : Transaction \times DATA \longrightarrow DATA$$

$$\forall x : DATA; t : Transaction \bullet$$

$$writeItem(t, x) = x \mid \exists valc' : TYPE \bullet$$

$$x.val := valc' \ \wedge \ x.tsw := ts(t)$$

where $valc'$ is the new value of *value* component of $x$ attained during computation. The function *writeItem* modifies the *val* component of the data item $x$.

The function $maxtime()$ returns the larger of two timestamp values. The definition of $maxtime()$ follows.

$$maxtime : TIME \times TIME \longrightarrow TIME$$

$$\forall x, y : TIME \bullet$$

$$maxtime(x, y) = x \Leftrightarrow (x \geq y) \ \vee$$

$$maxtime(x, y) = y \Leftrightarrow (x \leq y)$$

The CSP representation of the timestamp ordering protocol follows.

An operation is either a write or a read operation, so

$$operation ::= readItem \mid writeItem$$

Let $c$ and $d$ be channels[28]. The sets of permissible events through channels $c$ and $d$, respectively are:

$$\alpha c(T) = \{readTS, ts, writeTS\} \quad \text{and}$$

$$\alpha d(T) = \{readItem, writeItem\}$$

The permissible events of *BASICTO* are:

$$\alpha(BASICTO) = \alpha c(T) \ \cup \ \alpha d(T) \ \cup \ \{\lightning\}$$

The process $RVALUES(T, x)$ reads the timestamp of $T$ and the last read and write timestamps of the data item $x$ that $T$ wishes to access. It is specified as:

$$RVALUES(T, x) = c?readTS(x) \longrightarrow c?ts(T) \longrightarrow c?writeTS(x) \longrightarrow SKIP$$

[28]Recall, a channel is used for either I/O or communication operations.

Let $s$ be the trace of process $RVALUES(T, x)$ on completion. That is,

$$s = trace(RVALUES(T, x))$$

The process $Q(T)$ removes all actions of transaction $T$ from the database and resubmits $T$ after some time delay for re-execution from the beginning with a new timestamp[29]. Formally,

$$Q(T) = DELETE(T);\ PAUSE(k \mid k \in \mathbb{N}_1);\ submit(T) \longrightarrow BASICTO$$

where $DELETE(T)$ undoes all actions of $T$ using its transaction logs[30] and the process $PAUSE()$ pauses the progress of a transaction's operations for the specified time. $PAUSE()$ is specified using the CSP primitive processes $STOP$ and $SKIP$, as:

$$\forall x : TIME \bullet$$
$$PAUSE(x) \mathrel{\widehat{=}} STOP \stackrel{x}{\triangleright} SKIP$$

The basic timestamp protocol is finally given by:

$$BASICTO = d?operation \longrightarrow (WRITES \,\Box\, READS)$$

where

$$WRITES = (RVALUES(T, x);\ WRITEX(T, x)) \,\not\langle operation = writeItem \not\rangle\, SKIP$$

$$\begin{aligned} WRITEX(T, x) \;=\; & (\textbf{if } (readTS(x) > ts(T)) \vee (writeTS(x) > ts(T)) \\ & \quad \textbf{then } WRITES/s\!\not\downarrow Q(T) \\ & \quad \textbf{else } (writeItem(T, x) \longrightarrow BASICTO)) \end{aligned}$$

$$\begin{aligned} READS \;=\; & (RVALUES(T, x); \\ & (\textbf{if } (writeTS(x) > ts(T)) \\ & \quad \textbf{then } (READS/s\!\not\downarrow Q(T)) \\ & \textbf{else } (readItem(T, x) \longrightarrow x.tsr := maxtime(ts(T), readTS(x)) \\ & \quad \longrightarrow BASICTO)) \\ & \quad \not\langle operation = readItem \not\rangle\, SKIP \end{aligned}$$

---

[29]This is a dynamic form of timestamping.

[30]For update operations, the original values of the data are written first into the transaction logs to support transaction recovery in case of failure. Undo operations are performed on the transaction logs to restore the database state to a consistent state.

The process $Q(T)$ rolls back $T$ after the occurrence of the catastrophic event *interrupt* and resubmits it with a higher timestamp. In other words, $Q(T)$ rolls back $T$ to the previous state in the execution sequence.

Suppose $\alpha T_i$ represents a set of some operations on object set $o$ by transaction $T_i$ and $\alpha T_j$ represents some operations on objects $o$ by transaction $T_j$. Further, suppose $B = \alpha T_i \cap \alpha T_j \neq \varnothing$. The concurrent execution of both transactions using the timestamping protocol is given by:

$$TO = T_i \Big\|_{B} T_j$$

Using the *BASICTO* process defined previously, the process $TO$ can be executed safely without violating the correctness of the database. The $trace(TO)$ satisfies the ordering relation $T_{i(o_k)} \leq T_{j(o_k)}$ for all $i, j, k \in \mathbb{N}$ and for all $o \in B$ and $i \neq j$ whenever $ts(T_i) < ts(T_j)$. Formally,

$$\begin{aligned} trace(TO) \quad \textbf{sat} \quad & [(\forall\, i, j, k \in \mathbb{N};\ o \in B \mid i \neq j \bullet \\ & \exists\, ts(T) \in TIME \bullet ts(T_i) < ts(T_j) \\ & \Longrightarrow trace(T_i \upharpoonright o_k) \leq trace(T_j \upharpoonright o_k)) \ \wedge \\ & \quad \textbf{interleaves}(trace(T_i), trace(T_j))] \end{aligned} \qquad \text{[R19]}$$

Recall, timestamp values are assigned according to the transactions' start time order. So each transaction's timestamp is a unique non-decreasing number thereby creating a serial order among concurrent transactions.

A schedule in which the transactions participate is conflict serializable as seen from the specification $trace(TO)$. The system enforces serializability that is the chronological order of the timestamps of the concurrent transactions. This protocol does not cause deadlock since $T$ waits for $P$ only if $ts(T) > ts(P)$. However, it does not produce recoverable schedules[31] and starvation may occur if a transaction gets aborted continuously.

[31] Algorithms based on timestamping for producing recoverable schedules, called *strict timestamp ordering* do exist (see [EN94] for details) but are not considered in this thesis.

### 5.11.2 The 2-Phase Locking Protocol

Recall, a *lock* is a variable associated with a data item used for controlling concurrent access to the data item. So a reference to a data item $d$ implicitly manipulates $d(v, l)$ where $v$ is the current value and $l$ is the current lock mode.

Using formal notation, data amenable to lock manipulation is defined below. First the basic types of locks[32] are given as:

$$LOCKTYPES ::= X \mid S \mid unlock$$

A data item is represented in the $\oplus$ notation as:

$$\oplus[DATA] = \langle\langle val, lock \rangle\rangle \&\& \langle\langle TYPE, LOCKTYPES \rangle\rangle$$

The following rules govern lock acquisition and release in the general locking protocol.

1. A transaction cannot request a lock on a data item for which it already holds a lock.
2. A transaction cannot unlock a data item if it does not hold a lock for it.
3. A transaction must acquire a lock on data item before it performs any read or write operations.
4. A transaction must release all locks after it completes all read or write operations.
5. Any access (i.e., read or write) request for a data item can only be granted if it is compatible with the data item's current lock mode (i.e., $S$ or $X$, respectively).
6. If the request is incompatible, the requesting transaction must **wait** until the requested lock mode becomes available.
7. The release of a shared lock on a data item by a transaction **may not** result in unlocking the data item because multiple transactions can concurrently hold a

[32]In this thesis, only the basic lock modes are considered. The intent here is to demonstrate the application of formal methodology in protocol design and analysis. The application easily scales up to more locking modes without loss of analytic power.

**Algorithm 2:** *General Locking Scheme*

```
loop:  WHILE more locks required DO
       get lock request(lock)
       IF lock = X THEN
            IF (current lock = X) or (current lock = S)
                THEN refuse request and wait until ready
            ELSE grant request and branch loop
       IF lock = S THEN
            IF current lock = S THEN
                grant request
                share-count = share-count + 1
                branch loop
            IF current lock = X THEN
                refuse request
                wait until ready
            ELSE grant request
                share-count = share-count + 1
                branch loop
       IF lock = unlock THEN
            IF current lock = X THEN
                release lock
                resume one of the processes waiting, if any
                branch loop
            IF current lock = S THEN
                share-count = share-count - 1
                IF share-count = 0 THEN
                     release lock
                     resume one of the processes waiting, if any
                     branch loop
                ELSE branch loop
       end{WHILE}
```

Figure 5.13: General Locking Scheme

shared lock on the data. The shared data item is unlocked when the lock count is zero.

Figure 5.13 represents the general locking protocol.

Two locks always conflict if they are on the same data item[33], they are issued by distinct transactions, and at least one of the locks is a write lock. When a lock request cannot be granted, the lock manager uses a fair[34] algorithm to avoid livelock. A blocked request must wait for all previously blocked lock requests to be granted.

To guarantee serializability, the locking and unlocking operations in every trans-

[33]A data item granule is the unit of data to be locked — either *coarse* granularity (large object units e.g., file) or *fine* granularity (small units of an object e.g., records of file or even tuples of a record).

[34]A fair scheduling algorithm is one that gives a fair chance to the satisfaction of every request. This is often achieved by using a queue and places new requests at the tail of the queue.

**Algorithm 3:** *Lock Conflict Resolution*

```
1       IF Transaction T lock request blocked THEN
2           IF wait-queue is empty THEN
3               add T to wait-queue
4           ELSE IF there is a transaction S awaiting same lock THEN
5                   abort T
6                   resubmit T later after some time delay
7               ELSE add T to wait-queue
8           end{ELSE IF }
9           end{IF }
10      end{IF}
```

Figure 5.14: Combined cautious waiting and immediate rescheduling

action obeys the simple positional principle that all locking operations precede the first unlock operation in the transaction. Thus, no further locking can occur subsequent to the first unlock operation. This positional principle is called the *two-phase locking* (2PL) protocol.

This thesis presents an integrated approach for dealing with situations when lock requests are refused. The approach combines *cautious waiting*[35], to prevent long chains of blocked transactions, with *immediate rescheduling* whenever they can proceed irrespective of their position in the queue. Thus, no deadlock can occur since a transaction can only wait for a transaction in progress to release its locks. Starvation is minimized too. Further, by not processing the waiting list sequentially, increases in CPU throughput is achieved by reducing overhead. Figure 5.14 shows the algorithm that handles blocked request.

Before giving a formal definition of the 2-PL protocol, the following functions are necessary. The function *increment* increases the value of its argument by 1 while the function *decrement* decreases the value of its arguement by 1.

$$increment, decrement : \mathbf{N} \longrightarrow \mathbf{N}$$

$$\forall i : \mathbf{N} \bullet increment(i) = i + 1 \ \wedge \ decrement(i) = i - 1$$

[35]In a cautious waiting scheme the length of a blocking chain is restricted with the aim of reducing the time a transaction has to wait for compatible lock modes while ensuring the prevention of possible deadlock.

To handle the refusal of lock request operation (which results in no lock acquisition) a data structure that holds waiting transactions is required.

$$WAITQ == \text{seq}(Transaction \times \text{seq}\, DATA)$$

Thus, WAITQ is a sequence of transactions waiting to get locks. Transactions are added to the tail of the list. The process *SUSPEND* defines the suspension of a transaction using *WAITQ*.

$$\begin{aligned} SUSPEND(T_i) = \textbf{if } & WAITQ = \langle\,\rangle \textbf{ then } WAITQ' = WAITQ \frown T_i \\ & \textbf{else if } x : DATA \in T_i \wedge (\exists\, T_j \textbf{ in } WAITQ \bullet x : DATA \in T_j) \\ & \qquad \textbf{then } CANCELL(T_i) \\ & \qquad \textbf{else } WAITQ' = WAITQ \frown T_i \end{aligned}$$

$$\begin{aligned} CANCELL(T_i) \triangleq\ & DELETE(T_i);\ RELEASELOCKS(T_i); \\ & PAUSE(t);\ SUBMIT(T_i) \end{aligned}$$

where the process $DELETE(T_i)$ undoes all actions of $T_i$ based on $T_i$ execution logs and $SUBMIT(T_i)$ puts the transaction in the ready queue (for processing from its beginning), the scheduler thereby selects the next transaction for scheduling. The details of $DELETE(T_i)$ and $SUBMIT(T_i)$ processes are omitted in this thesis (because they are implementation issues). However, their omission does not affect the completeness of the specifications nor our ability to understand them.

The formal definition of the 2-PL protocol follows:

$$\begin{aligned} & TWOPHASE \triangleq GROWPHASE;\ SHRINKPHASE \\ & GROWPHASE = (ACQUIRELOCKS \mid\mid\mid OPERATIONS) \ \overset{\nabla}{_{\{\text{unlock}\}}} SKIP \\ & SHRINKPHASE = RELEASELOCKS \mid\mid\mid OPERATIONS \end{aligned} \qquad \text{[R20]}$$

Both of these phases are monotonic. The number of locks increases in the *GROWPHASE* and decreases in the *SHRINKPHASE*. The processes for the two-phase protocol must be bound to a transaction. In that way, we are able to analyse the acquisition and release of locks on the data items it manipulates.

To define the subprocesses, let $C$ denotes the allowable operation set and $B$ represents the set of lock modes. So,

$B = \{X, S, unlock\}$, and
$B_1 = B \backslash \{unlock\}$

$$
\begin{aligned}
ACQUIRELOCKS \;=\; & (k : B\, \mu\, k \bullet \\
& (test?flag \longrightarrow \textbf{if } (flag = false) \textbf{ then } SKIP \\
& \textbf{else } (request?k \longrightarrow c?DATA.lock \longrightarrow \\
& \quad \textbf{if } (k = X) \textbf{ then} \\
& \quad ((refuse(k) \longrightarrow (SUSPEND(T_i) \;|||\; ACQUIRELOCKS)) \\
& \quad \not<((Val(DATA.lock) = X) \vee (Val(DATA.lock) = S)) \not> \\
& \quad (DATA.lock = X \longrightarrow ACQUIRELOCKS)) \\
& \quad \textbf{else } (\textbf{if } (k = S) \textbf{ then } ( \\
& \quad (DATA.lock = S \longrightarrow increment(\#S) \\
& \qquad\qquad \longrightarrow ACQUIRELOCKS) \\
& \qquad \not<(Val(DATA.lock) = S) \not> \\
& \quad ((refuse(k) \longrightarrow (SUSPEND(T_i) \;|||\; ACQUIRELOCKS)) \\
& \qquad \not<(Val(DATA.lock) = X) \not> \\
& \quad (DATA.lock = S \longrightarrow increment(\#S) \\
& \quad \longrightarrow ACQUIRELOCKS))))
\end{aligned}
$$

The event $test!flag$ is an output event of the transaction manager that evaluates to true when the transaction $T$ needs to access or update a data item not previously accessed or updated. The acquisition of locks stops on the occurrence of the first unlock event. The process *OPERATIONS* is defined as:

$$
\begin{aligned}
OPERATIONS \;=\; & (y : C\, \mu\, y \bullet \\
& \quad (y \longrightarrow OPERATIONS \mid y \longrightarrow SKIP))
\end{aligned}
$$

The definition of the RELEASELOCKS process follows:

$$
\begin{aligned}
RELEASELOCKS \;=\; & (request?k \longrightarrow \\
& \textbf{if } (k = unlock) \textbf{ then} \\
& \quad (\textbf{if } (Val(DATA.lock) = X) \textbf{ then} \\
& \quad (DATA.lock = unlock \longrightarrow
\end{aligned}
$$

$$
\begin{array}{l}
(RESUME(DATA) \;|||\; RELEASELOCKS)) \\
\textbf{else } (decrement(\#S) \\
\longrightarrow \textbf{if } (S.count = 0) \textbf{ then} \\
(DATA.lock = unlock \longrightarrow \\
(RESUME(DATA) \;|||\; RELEASELOCKS)))) \\
\textbf{else } RELEASELOCKS) \\
\mid k \longrightarrow SKIP)
\end{array}
$$

The process *RESUME* resumes one of the transactions waiting in *WAITQ* queue whose data item awaiting lock acquisition is enabled by the unlock event (operation). Formally, it is defined as:

$$
\begin{array}{l}
RESUME : DATA \mapsto PROCESS \\
\forall x : DATA \bullet \\
\quad RESUME(x) = \textbf{if } WAITQ = \langle\,\rangle \textbf{ then } SKIP \\
\qquad \textbf{else if } (\exists\, T_j \text{ in } WAITQ \bullet x \in T_j) \\
\qquad\quad \textbf{then } (RemWAITQ(T_j) \longrightarrow ACQUIRELOCKS(T_j)) \\
\qquad\qquad \textbf{else } SKIP
\end{array}
$$

where $RemWAITQ(T_j)$ is a function that removes a waiting transaction from the *WAITQ* list and compacts the list after the operation.

To guarantee correct behaviour of the above processes requires the imposition of additional constraints on the traces. Let

$$
\begin{array}{l}
p = traces(ACQUIRELOCKS), \\
q = traces(OPERATIONS), \\
r = traces(RELEASELOCKS) \text{ and} \\
O = \{data\_item\_o\}
\end{array}
$$

The trace of *GROWPHASE* must satisfy the following:

$$trace(GROWPHASE) = \textbf{interleaves}(p, q) \wedge (\forall i \in \mathbf{N} \bullet p \upharpoonright O_i \leq q \upharpoonright O_i)$$

Similarly, the trace of *SHRINKPHASE* must satisfy:

$$trace(SHRINKPHASE) = \textbf{interleaves}(q, r) \wedge (\forall i \in \mathbf{N} \bullet q \upharpoonright O_i \leq r \upharpoonright O_i)$$

where the subscript on $O$ serves as an index to the data items.

The trace of the actions of *OPERATIONS* is given by:

$$traces(OPERATIONS) = y^* \mid (\forall p \in C, m \in B_1 \bullet m \leq p \text{ in } y^*)$$

Finally, the trace of the actions of *ACQUIRELOCKS* is given by:

$$traces(ACQUIRELOCKS) = t^* \mid (t = X \vee S) \wedge (\forall i \in \mathbb{N}, X_i \in B \bullet X_i \notin D \mid D = \bigcup_{T_i}^{n} X)$$

where $T$ is the set of transactions, $i = 1 \ldots n$, $n$ the cardinality of $T$, and the $^*$ represents Kleene's star [HU79].

The trace of the *RELEASELOCKS* process is given by:

$$traces(RELEASELOCKS) \;\widehat{=}\; (r : traces(RELEASELOCKS), e : traces(ACQUIRELOCKS) \bullet (\forall i : \mathbb{N}, o : DATA \bullet e \upharpoonright O_i \leq r \upharpoonright O_i))$$

Thus, a parallel combination of the *TWOPHASE* processes up to $n$ times where $n$ is the number of transactions executing at a particular time is possible without failure. The 2-PL protocol ensures that the schedules involving transactions using this rule is always serializable. The order in which executing transactions acquire locks determine the order of transactions in the equivalent serial schedule.

Consider the following bank setting. A customer is transfering some money from one account, say *Savings*, to another, say *Checking*. The customer's transfer operation is a single transaction that can be modelled as two nested subtransactions. Lets call this transaction $T_{cust}$. Suppose there is a timed triggered custom bank transaction, called $T_{bank}$, that sums all account balances at the end of a day's operations. $T_{bank}$ runs at *time* = 24.00 hours. Figure 5.15 shows a possible sequence of operations of these transactions (only $T_{cust}$ operations are shown for $T_{bank}$).

If $T_{bank}$ and $T_{cust}$ coincidentally run at the same time, the computations will generate inconsistent data unless their access to the accounts is controlled. In this example, the access control mechanism is the 2PL. Applying specification [R20], two possible arrangements of the transactions' operations are feasible; either all lock operations (on common data items) by $T_{cust}$ precede all those of $T_{bank}$ or vice versa. Figure 5.16 shows one of the two possible scenarios. Notice that the extended possession of locks beyond

| Transaction $T_{cust}$ | Transaction $T_{bank}$ |
|---|---|
| Read *Savings* | Initialize *sum* |
| $Savings = Savings - amount$ | Read *Savings* |
| Write *Savings* | $sum = sum + Savings$ |
| Read *Checking* | Read *Checking* |
| $Checking = Checking + amount$ | $sum = sum + Checking$ |
| Write *Checking* | Write *sum* |

Figure 5.15: Operations of example transactions.

the point of last use[36] forces a serialization order of the transactions involved thereby producing correct results.

To capture S2-PL, the *SHRINKPHASE* process is altered to reflect the way locks are released. Thus,

$SHRINKPHASE \triangleq OPERATIONS;\ RELEASELOCKS$

Similarly, combining both C2-PL and S2-PL so that schedules are both recoverable and serializable, the following definitions now applies.

$TWOPHASE \triangleq GROWPHASE;\ SHRINKPHASE$

$SHRINKPHASE \triangleq OPERATIONS;\ RELEASELOCKS$

$GROWPHASE = (ACQUIRELOCKS \overset{\nabla}{_{\{\text{unlock}\}}} SKIP$

All other definitions hold.

## 5.12 The Electronic Shopping Mall

The Electronic Shopping Mall model is characterised by active capabilities (for timely response to events and changes in the environment), support for long-running transactions and possible partial sharing of results, allows compensation to undo effects of undesirable

[36]This is necessary to preserve consistency and avoid reading inconsistent data.

| Transaction $T_{cust}$ | Transaction $T_{bank}$ |
|---|---|
| $Lock(Savings) := X$ | $Lock(sum) := X$ |
| Tt1 Read *Savings* | Initialize *sum* |
| $Savings = Savings - amount$ | |
| Write *Savings* | |
| $Lock(Checking) := X$ | |
| $Unlock(Savings)$ | |
| Read *Checking* | $Lock(Savings) := S$ |
| $Checking = Checking + amount$ | Read *Savings* |
| Write *Checking* | $sum = sum + Savings$ |
| $Unlock(Checking)$ | |
| | $Lock(Checking) := S$ |
| | Read *Checking* |
| | $sum = sum + Checking$ |
| | Write *sum* |
| | $Unlock(Savings)$ |
| | $Unlock(Checking)$ |
| | $Unlock(sum)$ |

Figure 5.16: Operations of example Transactions using 2PL.

committed transactions, and supports for heterogenous and autonomous environments. The partial sharing of results necessitates the distinction between vital transactions and non-vital transactions (as previously discussed in Section 5.7 page 125).

When the GTM broadcasts an initial query for a product more than one LDB may respond to the query. The choice of which LDB eventually processes the query uses a selection mechanism that may depend on the following criteria:

- first to affirmatively respond to the request,
- proximity to client and product shipping cost,

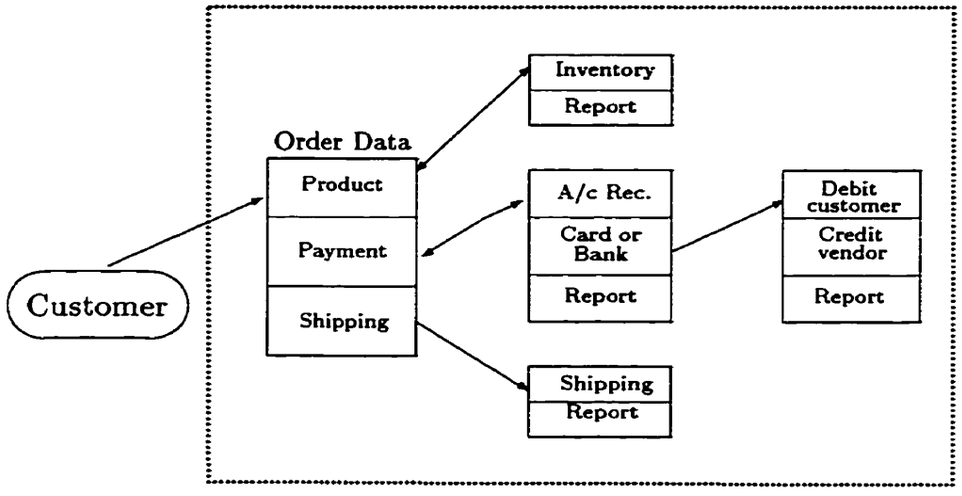


Figure 5.17: A Customer Order Transaction

- least cost of product requested,
- least cost based on the cost analysis of response time and total resources utilization,
- least load of the local databases that responded, and
- other network related issues

The determination of the optimal method for selecting an LDB to continue processing is not the focus of this research. Rather the researcher assumes that the system always selects an LDB optimally.

To illustrate the basic ideas discussed, a simple order entry system is used where customer requests are accepted and processed. An order may require services located at different sites and/or databases. For example, an order transaction may necessitates accessing the inventory database, accounts receivable and payable, bank or credit information, and logistics data as illustrated in Figure 5.17. Therefore, an order transaction can be decomposed into a set of processes that interact to service the transaction.

The relevant objects and functions (services) are abstracted in Figure 5.18. The objects are the customer, product, invoice, inventory, and accounts while the relevant

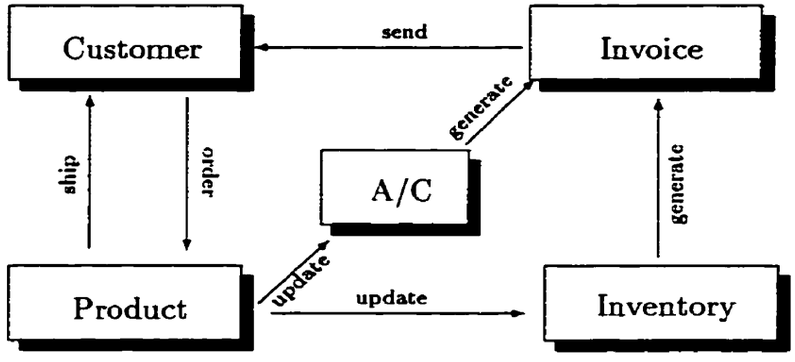


Figure 5.18: Process Model of Order Processing

services are orders, ship, update, generate, and send. The author relies on the reader's intuition about these components functionalities thus focusing on the transaction formalism in this thesis. The complete system may contain subsystems, each containing numerous processes which interact with one another. Processes can call upon other processes to fulfill a request for action and/or information. Process interaction is triggered from some external event such as a user request, or a request from another process.

A diagrammatic representation of the transactions in the application domain under consideration is shown in Figure 5.19.

**Algorithm 1** *The sequences of operations*

1. MDB decomposes the transaction's GT. First the LDBs are queried for the availability of the item.
   - If found then report back to the MDB while the local store local database continues to do its inventory processing. A hold is placed on the item in inventory at the moment.
   - If not found then report back to the MDB and abort immediately.
2. The MDB now request payment.
   - Acquire the account.
   - Validate account balance.
   - Initiate transfer to the store's bank account.
3. Stores's bank reports successful transfer to the MDB's transaction.
   - Then the store local database transaction (inventory update) can now commit successfully.
4. The GT commits successfully.

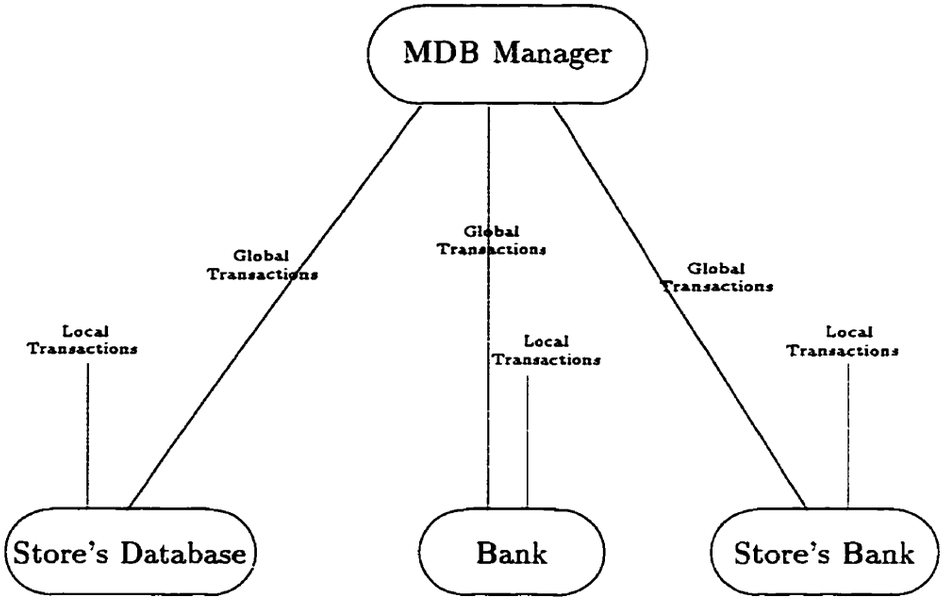


Figure 5.19: A Graphical Representation of Purchase Transaction.

There is a time dimension in the update of the inventory database (which may or may not be located in the store's local database site) thus the global inventory could potentially be held up. The execution of the global transaction allows local transactions at the banks (in this example, the customer's and the store's banks that could possibly be different) and at the participating stores. The shareable data object at the store is the inventory database. Similarly, the shareable objects at the bank are account-number and balance. The banks may enforce strict correctness to avoid the problems of duplicate withdrawal.

In pseudocode, the above transaction could be represented as follows:

**Algorithm 2** *Pseudocode Representation of Purchase Transaction*

```
Purchase-Transaction;
begin
        BROADCAST purchase requirements;
        TRIGGER inventory search;
        MAKE payment
end;
```

where the operations of the procedures *TRIGGER* and *MAKE* are given as:

```
TRIGGER inventory search;
begin
        check the store's database;
        validate inventory quantity;
        update inventory balance;
        produce reports;
        terminate
end;

MAKE payment;
begin
        enter and validate customer information and PIN;
        select account;
        enter amount;
        validate balance;
        update account;
        produce receipt
end;
```

In very high level abstract program design pseudocode specification, the whole transaction could be represented as:

**Algorithm 3** *Program Design Pseudocode Specification*

```
Purchase-Transaction
begin
        BROADCAST-transaction;
        INVENTORY-transactions :
                Check and validate,
                Update inventory,
                Produce reports;
        MAKE-payment :
                Validate account,
                Accept payment,
                Produce bill
end;
```

Examining the above pseudocode representations reveals the following: (1) there is no indication of the operations interleaving order, (2) no indication of concurrent (or parallel) activities thus obliviating the potential benefits of concurrent operations, and (3) it is impossible to deduce causal dependency relationships between the subprocesses.

### 5.12.1 Some Applicable Functions

The following are some useful functions that will become handy in the formal specification of the example problem. Only their signatures are given leaving out their details. This

does not affect the understanding of their application. The function *validBal* checks if there is enough money in an account. It returns *true* if the current balance is greater than the amount to be spent from the account and it returns *false* otherwise. Its signature is given by:

$$validBal : Account \times \mathbf{R} \longrightarrow BOOLEAN$$

The function *decrementDb* reduces the quantity in stock of an item by the amount specified, given by:

$$decrementDb : Db \times item \times \mathbf{N} \longrightarrow \mathbf{N}$$

Similarly, the function *debitAcct* reduces the current account balance by the amount specified, given by:

$$debitAcct : Account \times \mathbf{R} \longrightarrow \mathbf{R}$$

The function *creditAcct* increases the current account balance by the amount specified. Its signature is:

$$creditAcct : Account \times \mathbf{R} \longrightarrow \mathbf{R}$$

Further, the function *isinDb* checks a given data base for a particular item. It returns *true* if the item is found and it returns *false* otherwise. Its signature is:

$$isinDb : Db \times item \longrightarrow BOOLEAN$$

The specification of the sample purchase transactions in CSP follows. *ABORT* and *COMMIT* are transaction specific operations and are used here in their pure database transaction semantics. The set of events executable by the process *PURCHASE* is given by:

$$\alpha PURCHASE = \alpha BROADCAST \cup \alpha INVENTORY$$

where $\alpha INVENTORY = \{prodInfo, inventoryDb, bicycle\} \cup \alpha UPDATE\_INV \cup \alpha PAYMENT$

$$\alpha UPDATE\_INV = \{inventoryDb, bicycle, produceReports\}$$

$$\alpha PAYMENT = \{custInfo, selectAcct, PIN, acctBal, amount\}$$

The communication events *okay* and *fail* are transmitted through the communication channel $e$, represented as:

$$\alpha e(PURCHASE) = \{okay, fail\}$$

The process *PURCHASE* is defined as:

$$PURCHASE = BROADCAST;\ INVENTORY;\ COMMIT \qquad \text{[R21a]}$$

where

$$\begin{aligned} INVENTORY \ = \ & (c?prodInfo \longrightarrow \textbf{acquire}(inventoryDb) \\ & \longrightarrow bicycle = isinDb(inventoryDb, prodInfo) \\ & \longrightarrow (PAYMENT \parallel UPDATE\_INV)\backslash\{e\} \trianglelefteq bicycle \trianglerighteq ABORT \\ & \longrightarrow SKIP) \end{aligned} \qquad \text{[R21b]}$$

The communication along channel $e$ used to synchronize *PAYMENT* and *UPDATE_INV* is hidden to prevent them from further being constrained by the environment.

$$\begin{aligned} UPDATE\_INV \ = \ & ((inventoryDb(qbicycle) = decrementDb(inventoryDb, pbicycle) \\ & \longrightarrow produceReports \longrightarrow e?x \\ & \longrightarrow \textbf{if}\ (x \neq \text{“}okay\text{”})\ \textbf{then}\ ABORT \\ & \qquad\qquad \textbf{else}\ (\textbf{release}(inventoryDb) \longrightarrow SKIP)) \end{aligned} \qquad \text{[R21c]}$$

$$\begin{aligned} PAYMENT \ = \ & (c?custInfo \longrightarrow selectAcct \longrightarrow c?PIN \\ & \longrightarrow \textbf{acquire}(acctBal) \longrightarrow c?amount \\ & \longrightarrow w = validBal(account, amount) \\ & \longrightarrow (e!x \longrightarrow (ACCEPT \parallel BILL) \trianglelefteq w \trianglerighteq (e!x \longrightarrow ABORT)) \\ & \longrightarrow SKIP) \end{aligned} \qquad \text{[R21d]}$$

where $x = okay$ when $w$ evaluates to true and $x = fail$ otherwise.

$\alpha ACCEPT = \{acctBal, amount\}$ and $\alpha BILL = \{produceInvoice, creditAcct\}$

$$ACCEPT \quad = \quad (acctBal = debitAcct(account, amount)$$

$$\longrightarrow \textbf{release}(account)$$
$$\longrightarrow SKIP) \qquad \text{[R21e]}$$

$$BILL = (produceInvoice$$
$$\longrightarrow creditAcct(tax, amount)$$
$$\longrightarrow SKIP) \qquad \text{[R21f]}$$

The **release**(*inventoryDb*) event can be performed only after *validBal*(*account*, *amount*) event of the *PAYMENT* process evaluates to true. That is, *UPDATE_INV* process is commit dependent on the *PAYMENT* process[37]. The trace specification defines the required interleaving order. Figure 5.20 represents the above CSP specification. The shaded node (•) represents parallel operations while $---\rightarrow$ shows commit dependency. When the *ACCEPT* process commits, it initiates a trigger to resume the suspended inventory update process for a successful commit operation.

The communication event *okay* on channel $e$ synchronizes operations dependency relationships between *PAYMENT* and *UPDATE_INV* processes. The *UPDATE_INV* process is temporarily suspended until the *PAYMENT* process is ready to perform the synchronization event ($e!x$) at which point *UPDATE_INV* resumes to accept the input communication event ($e?x$) and then progresses. Note that when two concurrent processes communicate with each other by output and input on a single channel, they can not deadlock [Hoa85].

Recall, the trace of a process is the history of the process up to that time. The trace concept, therefore, is central to recording, understanding and describing the behaviour of processes. The concept of compositionality of traces plays a fundamental role in reasoning about processes. Consider the following:

$$trace(ACCEPT) = \langle debitAcct, \textbf{release}(account)\rangle$$
$$trace(BILL) = \langle produceInvoice, creditAccts\rangle$$
$$trace(UPDATE\_INV) = \langle decrementDb, produceReports,$$

[37]This is the commit dependency of *UPDATE_INV* process on *PAYMENT*. Also, *UPDATE_INV* is causally dependent on *PAYMENT* via the synchronization variable $x$.

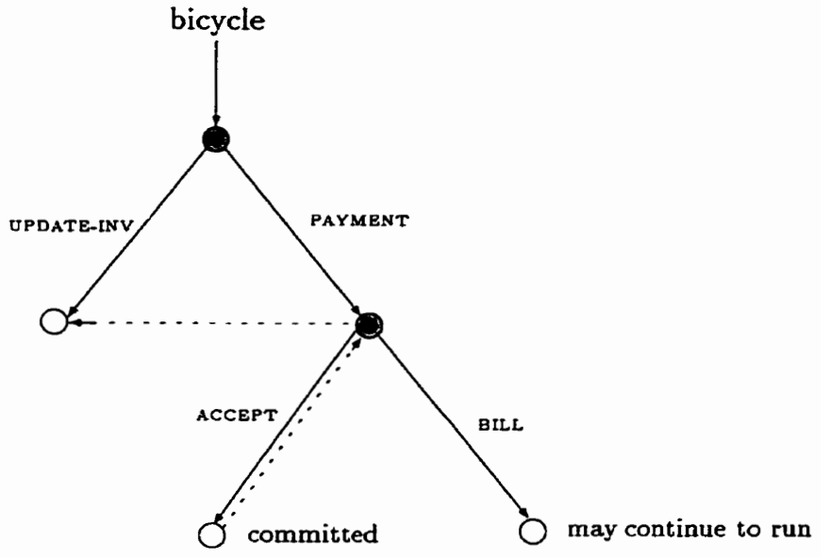


Figure 5.20: Dependence Relationship of Sample Transaction.

$$\textbf{release}(inventoryDb)\rangle$$

Further, for notational convenience, let

$$m = traces(BILL)$$

$$r = traces(ACCEPT)$$

$$k = traces(UPDATE\_INV)$$

$$A = \{\textbf{release}(inventoryDb)\}$$

$$s_1 = traces(UPDATE\_INV \upharpoonright A)$$

The traces of $PAYMENT$ is given by:

$$trace(PAYMENT) = (\langle custInfo, selectAcct, PIN,$$
$$\textbf{acquire}(account), amount, validBal\rangle)$$
$$\wedge\ (\textbf{interleaves}(m, r) \vee ABORT)$$

Similarly, the following abbreviation shall apply:

$$t = traces(PAYMENT)$$

The traces of the process INVENTORY is given as:

$$trace(INVENTORY) = (\langle prodInfo, \textbf{acquire}(inventoryDb), validQty\rangle)$$

$$\wedge\ ((\textbf{interleaves}(t, k)$$
$$\wedge\ (\forall\ t_1 \in t, s_1 \in k \mid s_1 = k \upharpoonright A \bullet t_1 \leq s_1))$$
$$\vee\ ABORT) \qquad \text{[R22]}$$

The final specification of the transaction PURCHASE is given as:

$$PURCHASE = (\forall\ q \mid q \in traces(BROADCAST);$$
$$t \mid t \in traces(INVENTORY)\ \bullet$$
$$traces(PURCHASE) = q \wedge t;\ COMMIT) \qquad \text{[R23]}$$

## Summary

An abstract event-based model that incorporates both causality and timing information was presented. The framework uses time as mathematical objects for descriptive and analysis purposes. Thus, time is a first class object. The framework models transactions as mathematical relations over database states. Transactions' execution correctness rely on the preservation of the predicates instead of only the notion of serializability.

The expressive power of the framework is demonstrated by using the formalism to define (a) some of the existing hierarchical transaction models, (b) transaction correctness and concurrency control protocols, and (c) electronic commerce application.

# Chapter 6

# CORRECTNESS PROOF

Any transaction system's development ultimate goal is to deliver a transaction software product (SP) that meets an application's requirements (AR). Figure 6.1 shows the essential abstract transformations. The transformation AR $\Longrightarrow$ SP is difficult and error prone because it involves both conceptual and formal domains; application requirements and the software that meets these needs, respectively. So the application's requirements are abstracted into a formal specification (FS) from which SP is derivable. Thus, AR $\Longrightarrow$ FS and FS $\Longrightarrow$ SP. The formal domain of FS is subject to mathematical and logical analysis so correctness can be established while the conceptual domain of AR lacks a formal model's precision and mathematical elegance. SP is derivable from FS via detailed transformations of the form:

$$FS \Longrightarrow FS_1 \Longrightarrow FS_2 \Longrightarrow \cdots \Longrightarrow SP$$

FS is AR's initial specification proven correct that forms the basis with respect to which SP's correctness must be preserved.

The steps leading to SP must be proven correct and produce correctness-preserving transformations in a logically correct, consistent, and well-ordered sequence. This ensures that unacceptable behaviours are not introduced and unambiguously establishes the specification's correctness. The transformations form the basis for the logical deductions and the implementations that result in behaviours which satisfy the application's requirements. This in turn supports confirmability and verifiability. Since the creation of

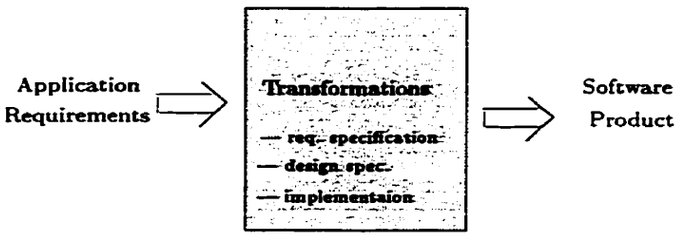


Figure 6.1: Application needs - Software Solution Relationship

FS is the focus of this thesis, it is necessary that the conceptual translation into formal form (i.e., FS) is correct. So we must prove that the specifications given in Chapter 5 are correct in order to ensure that transaction software generated from them are correct. This takes seriously the obligation to produce reliable programs to a known specification.

This chapter presents the correctness proof of the specifications given in the preceding chapter. Rigorous arguments following from the specifications provides the logical basis for the proofs. First Section 6.1 presents the methodology taken.

## 6.1 Proof Methodology

Most proofs in CSP use bottom-up approach where each component is proven correct separately and more complex systems are compositionally developed by using the provably correct components. Thus, a proof of a compound process's property is derivable from a proof of correctness of its parts.

Recall, the decomposition of the complex system into its subcomponents requires conformance to the production of correctness-preserving transformations in a logically correct, consistent, and well-defined sequence with clearly defined interrelationships. Adherence to this principle provides for the validity of the interactions between subcomponents via well-defined interfaces. Thus, the canonical transformations ensure the exclusion of unacceptable behaviours in the interactions between subcomponents when composed and no information or functionality is lost as a result of the decomposition. The

and functional correctness using mechanisms that ensure adherence to and preservation of timing constraints. The transaction aborts if it violates either the temporal or functional correctness. The temporal constraints are used in synchronizing concurrent access to a data object by multiple transactions thereby providing an acceptable degree of functional correctness.

To guarantee correctness, appropriate synchronization mechanisms must be provided. In addition, the processes that capture transaction behaviours must also satisfy both safety and liveness properties.

## 6.2 Safety and Liveness Properties

A trace of a process $P$ satisfies *safety property* if and only if:

1. $trace(P) \neq \langle\ \rangle$ and

2. $trace(P)$ is prefix-closed.

A trace of $P$ is prefix-closed if there exists $\beta$ and $\beta'$ such that $\beta$ in $trace(P)$ and $\beta'$ is a finite prefix of $\beta$ then $\beta'$ in $trace(P)$. Prefix-closure is reasonable because if a trace is safe so is any of its prefix. Note that

$$trace(P) = \langle\ \rangle \Rightarrow \#(events(trace(P)) = 0 \Rightarrow \text{no events occur.}$$

Safety property violation occurs in a trace when some particular events in the trace should not have been admitted (during the trace's extension via the process's progress).

A trace property $P$ is a *liveness property* if every finite sequence over the elements of $P$ has some extension in $trace(P)$. Informally, no matter what has happened up to some point, eventually something good will occur. General liveness properties can also be captured by restricting the event sequences a process can undergo.

## 6.3 Prove Theorems about Specifications

**Theorem 2** All processes are prefix closed.

**Proof**

*Basic idea*: Prove by induction on the length of a finite execution generating the given trace. Given a process $P$ and its $trace(P)$, if $Active(P)$ then $P$'s trace is extendable by at least an event $e \mid e \in \alpha(P)$. Thus,

$$trace(P)' = trace(P) \frown \langle e \rangle$$

Further, if

$$\begin{aligned} &\exists \beta : TIMEDTRACE \mid \beta \text{ in } trace(P) \bullet \\ &\quad (\exists \beta' : TIMEDTRACE \mid \beta' \textbf{ prefix } \beta \wedge \#\beta' < \#\beta) \Rightarrow \beta' \in traces(P)) \\ &\quad \Rightarrow \beta' \text{ in } trace(P) \end{aligned}$$

So elements of all possible traces of $P$, $traces(P)$, are prefixes of $trace(P)$. □

Recall, an important property of a history is the prefix-closed requirement. Thus, given any history, all pre-histories of the history are also histories. This is because any pre-history of an object (or transaction set) is the history of that object (transaction set) at some earlier stage in its evolution and hence represents a possible history of the object's class.

**Theorem 3** Causality respects time.

**Proof**

*Basic idea*: Time is monotonically increasing. Thus

$$\begin{aligned} &\forall t_i, t_j : TIME \bullet \\ &\quad t_i < t_j \Leftrightarrow (\exists S : \text{seq}_1\ TIME; i, j : TIME \mid \\ &\qquad i \in \text{dom}\, S \wedge j \in \text{dom}\, S \wedge \\ &\qquad\quad S(i) = t_i \wedge S(j) = t_j \bullet i < j) \end{aligned}$$

By definition of $\xrightarrow{CD}$ (see [R2]), if $A$ happening at time $\lambda$ causes $B$'s occurrence at time $\mu$ then $\lambda$ must precede $\mu$ in the mapping of the times to a number line. Thus,

$$(\mu, B) \xrightarrow{CD} (\lambda, A) \Rightarrow \mu < \lambda$$

Thus, in any history $H$ that contains the transactions having $A$ and $B$ respectively,

$$A <_H B$$

So by definition [R3], $\mu < \lambda$. Similarly, by application of [R2a] $\xrightarrow{\mathcal{CD}_{\blacksquare}} \Rightarrow \mu < \lambda$. So the cause event's time always precede the effect's time. Therefore, causality respects time.

□

**Theorem 4** If every transaction in a schedule follows the 2PL rule, the schedule is guaranteed to be serializable.

**Proof**

Recall, by specification [R20],

$$2PL \triangleq GROWPHASE; SHRINKPHASE$$

Since the acquisition of locks is monotonically increasing in the GROWPHASE stage because of the semantics of $\overset{\nabla}{_{\{\text{unlock}\}}}$ all lock acquisition must precede the first unlock operation.

$$GROWPHASE = (ACQUIRELOCKS \;|||\; OPERATIONS) \;\overset{\nabla}{_{\{\text{unlock}\}}}\; SKIP$$

But to guarantee serializability, the locking and unlocking operations in every transaction must obey the simple rule that *all* locking operations precede the first unlock operation. Suppose

$$\begin{aligned}
&\forall P : PROCESS \bullet \\
&\quad \exists p, q : Op \mid p, q \in \alpha(P) \;\wedge\; p = lock \;\wedge\; q = unlock \;\wedge \\
&\qquad\qquad p \text{ in } traces(p) \;\wedge\; q \text{ in } traces(p) \bullet \\
&\qquad\qquad\quad time(q) < time(p) \Rightarrow \text{violation of 2PL rule}
\end{aligned}$$

But specification [R20] will not permit such operations. So [R20] admits only consistent operations that guarantees serializability. □

**Corollary**

The locking mechanism, by enforcing 2PL rules, also enforces serializability.

**Proof**

Given a process $P$ that obeys the 2PL rule. Thus,

$$\forall s : TIMEDTRACE \mid s \in traces(P) \bullet$$
$$\forall p : Op \mid p = lock \bullet p \text{ in } s \Rightarrow$$
$$\neg (\exists q : Op \mid q = unlock \bullet time(q) < time(p))$$

If process $P$ executes concurrently with a similarly defined process $Q$, their execution is serialized according to their lock acquisition order on a common data object. That is,

$$\forall P, Q : PROCESS \bullet$$
$$\exists x : Data \mid (\exists p, q : Op \bullet p \in \alpha(P) \wedge q \in \alpha(Q) \wedge Conflict(p(x), q(x)) \bullet$$
$$time(p(x)) < time(q(x)) \Rightarrow time(lock_P(x)) < time(lock_Q(x))$$

In this case, $P \rightsquigarrow Q$. But $P \rightsquigarrow Q \equiv P; Q$ in effects so the schedule containing $P$ and $Q$ is equivalent to a serial execution (P;Q) of $P$ and $Q$. Thus, the schedule is serializable

□

**Theorem 5** Every schedule produced by the process PURCHASE admits a consistent enumeration (or schedule).

**Proof**

*Basic idea*: Any history produced by PURCHASE is serializable.
Note that PURCHASE is modelled as a nested transaction (see specifications [R21a - d])
Following from specification [R23]

$$PURCHASE \textbf{ sat } traces(BROADCAST) \wedge traces(INVENTORY)$$

By assumption *BROADCAST* is deadlock free so we need only prove that

$$traces(INVENTORY)$$

admits consistent enumeration. Thus, we prove that *INVENTORY* process (1) is deadlock free, and (2) preserves the database's correctness after execution.

To prove the absence of deadlock in the above specification, reduces to demonstrating that

$$\forall s \in traces(PURCHASE) \bullet$$
$$trace(PURCHASE/s) \neq STOP$$

which formally represents the problem.

We know that given a process $P$,

$$traces(P) \frown STOP = STOP \Rightarrow \text{Deadlock} \qquad \text{(see [Hoa85])}$$

Thus,

$$trace(PURCHASE/s) = s \frown (t \mid t \text{ in } trace(PURCHASE) \wedge$$
$$t = trace(PURCHASE) \textbf{ after } time(last(s))$$

But $STOP \notin \alpha PURCHASE \Rightarrow$
$$STOP \notin traces(PURCHASE)$$

Therefore $traces(PURCHASE) \neq STOP$

But by [R21b] and [R22] the process *INVENTORY* only admits consistent events enumeration consistent with the partial ordering of the process's events. Therefore, every schedule produced by the process PURCHASE admits only a consistent schedule. □

**Theorem 6** Schedules admitted by both TO and 2PL protocols are behaviourally equivalent.

**Proof**

Let $Tset = \{T_1, T_2, \cdots, T_n\}$ be a transaction set consisting of $n$ transactions. The concurrent execution of transactions in *Tset* is the history given by:

$$HISTORY_{Tset} = \text{seq}_1 S \mid \text{ran } S \in T_i \wedge T_i \in Tset$$

By [R19] any schedule generated using the TO protocol enforces serialization order that corresponds to the order of the concurrent transactions' timestamps. So, using [R19] let

$$HISTORY_{Tset} = X \quad \text{where } X \text{ is a sequence of } Tset\text{'s transaction operations}$$

Similarly, applying [20] let

$$HISTORY_{Tset} = Y \quad \text{where } Y \text{ is a sequence of } Tset\text{'s transaction operations}$$

By Theorem 4, the 2PL produces serializable schedules. But $X$ and $Y$ contain the same operations since

$$\#X = \#Y = \#S \ \wedge$$
$$\forall p : Op \mid p \in T_i \wedge T_i \in Tset \bullet$$
$$p \text{ in } S \ \Rightarrow \ p \text{ in } X \wedge p \text{ in } Y$$

Thus, TO and 2PL accepts the same transactions. Also, in both $X$ and $Y$ the following holds:

$$T_i \rightsquigarrow^* T_j \Rightarrow \neg \, (\exists \, T_i \rightsquigarrow T_j \rightsquigarrow \cdots \rightsquigarrow T_j \mid T_i = T_j)$$

So, $X \equiv Y$ = conflict serializable (by [R19] and [R20])

Since both protocols generate serializable schedules, reordering nonconflicting events in the schedule produces a new schedule consisting of the same events as the previous. Thus, they have equivalent computational effects. [*Aside*: Only the relative ordering of conflicting operations determines the outcome of a transaction set's concurrent execution.]

So TO and 2PL protocols are behaviourally equivalent. □

Thus, given any system either the 2PL or TO can be applied; selecting the TO protocols for applications most suitable to it and the 2PL for others. The 2PL is better when transaction operations are predominantly updates whereas TO is better for read only transactions. Figure 6.2 illustrates the behavioural equivalence of the two protocols.

The 2PL determines serialization orders dynamically (according to the order in which data items are accessed) while TO statically determines serialization orders (when a transaction starts). The power of Theorem 6 is that given a set of transactions that can execute concurrently

- 2PL guarantees that the execution is equivalent to **some** serial execution of those transactions.
- TO guarantees that the execution is equivalent to a **specific** serial execution of those transactions corresponding to the order of their timestamps.

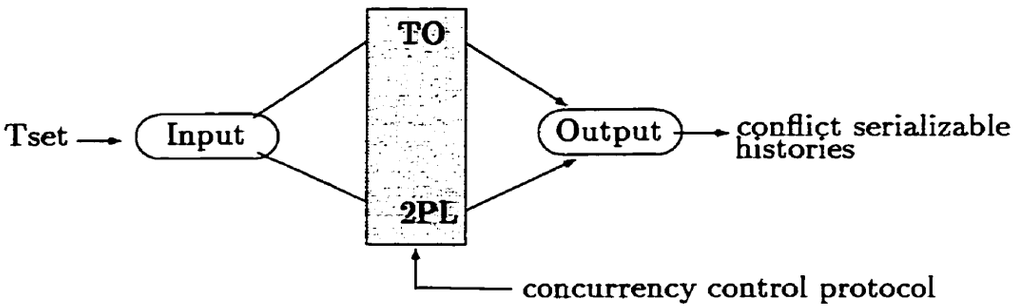


Figure 6.2: Behavioural Equivalence of TO and 2PL.

**Theorem 7** The specified protocols only admit mutually consistent schedules.

**Proof**

*Basic idea*: Show that there are no cycles in schedules accepted by both TO and 2PL. From Theorem 6 we know that both TO and 2PL produce serializable schedules and thus are equivalent in effects. Using definition [R4] and its transitive closure, any schedule admitted by both protocols is of the form $T_i \rightsquigarrow T_j \rightsquigarrow T_k \rightsquigarrow T_l \rightsquigarrow \cdots \rightsquigarrow T_n$ where $T_i \neq T_n$.

$\therefore$ both TO and 2PL produce mutually consistent schedules. □

**Theorem 8** All serializable executions are correct.

**Proof**

DBMS may execute transactions in *any* order as long as the *effect* is the same as that of *some* serial order.

By Theorem 1 (see page 134) each serializable execution has the same effect as some serial execution. Thus, proof of Theorem 1 directly applies. □

# Chapter 7

# CONCLUSION AND FUTURE WORK

Formal methods for developing transaction systems software are becoming increasingly necessary. To handle the complexities inherent in transaction systems requires combining these methods with sound development methodology which supports modularity and reusability

This thesis models transactions as mathematical relations over database states. Transactions' executions correctness rely on the preservation of the predicates instead of only the notion of serializability. By representing transactions as mathematical relations on database states having nested transactions with concurrently executing subtransactions and a schedule as a composition of the relations enables the capture of the full semantic scope of long-duration transaction systems. Partial ordering relation on the subtransaction's set represents the runtime dependencies between the subtransactions. The execution is correct if every subtransaction can access a database state that satisfies its pre-condition and if the result of all the subtransactions satisfies the post-condition of the transaction. This notion of correctness can produce a multi-level correctness criterion by extension to both the ancestors and descendants of a given transaction.

Recall, one important requirement of transaction specification language is the ability to capture concurrency which requires the specification language to model the simultaneous occurrence of multiple events in a transaction. Also, one important requirement

of transactions is the generation of schedules that satisfy the transactions event's timing and causality constraints while maintaining consistency and correctness. By using an event-based model that incorporates both causality and timing information in specifying transactions ensures (i) timing constraints and causal dependencies of the executions, and (ii) transactions correctness through concurrency control protocols by controlling any interleaving. Uncontrolled interleaving of transaction execution may violate the database consistency and thus its correctness because different conflicting transactions may simultaneously access the same data item.

Although the objective of a specification is often to capture the behaviour of some system over time, most specifications do not explicitly capture time information. However, it is desirable to explicitly provide mechanisms that facilitates the analysis of a transaction system's evolving behaviour as it executes over time. Recall, the execution of a transaction gives rise to sequences of timed actions, states, etc. Although such time sequences are usually not captured explicitly in the program text, it is frequently useful to have them available as mathematical objects for descriptive and analysis purposes. By using set comprehension to extract those time points at which events occur and placing appropriate constraints on those sets specifying transaction safety and liveness properties is feasible. The timed-event specification (the sequences of states and events which a transaction can undergo over time) facilitates the specification of liveness properties such as fairness, termination, and the guaranteed occurrence of events (by using the temporal operators defined). An abstract model of transaction models based on causality and timed execution is presented. This is used to specify history invariants which restrict the set of acceptable histories derivable from transactions event occurrences[1].

The model of CSP language applied in this thesis uses time as a first class entity to permit the specification of timing constraints within the language. This provides high level constructs for specifying timing constraints which helps to separate the specification of timing constraints from the means used to ensure their satisfaction and assigns

[1]Recall, the order of appearance of an operation in the schedules differentiates one schedule from another. This order is based on the process's interactions. Any two schedules are equivalent if the order of the causally dependent events in one is the same as the other.

further study.

Transaction's response time unpredictability may arise due to the requirement of maintaining transaction atomicity over participants in different sites and from the management of distributed data[2]. The response time of a transaction may be influenced by the location of its required data, which may not be known until its actual execution time. In addition, the underlying network's performance affects the transactions performance in overall. An investigation of mechanisms to accurately predict transaction response times is required so that the results presented in this thesis can be adapted to real-time applications.

While the transaction concept is by no means new and have had considerable application outside the academia, a complete formal specification of the different models and their semantics and theory is necessary to integrate them and prove their correctness. By doing so will also enable the development of semi-automatic tools to assist in developing more dependable applications based on these models. Extension of the specifications to include other models like workflow, interactions, and real-time models is vital.

Another possible extension is to incorporate object-orientation features into the TCSP specification language. This will enable a specifier to specify transaction models using the object-oriented paradigm and verify that a particular specification is realizable using a method similar to that of Object-Z so a history is contained within a class. To have an object-orientation features requires a definition of full formal semantics mapping constructs in the language to some semantic domain. This enables the development of a proof system for the language and the possibility of creating semi-automatic tools to aid in the software development process.

Another area that requires further investigation is the derivation of a new correctness criterion that incorporates transaction's semantics and serializability. By exploiting intra-transaction concurrency (based on the transaction's semantics) as many subtransactions of the same parent transaction as possible can concurrently execute thereby reducing the transaction wait times for consistent data states. Reducing transaction wait

[2]The creation of a transaction at one site is independent of the creation of transactions at other sites. Recall, each created transaction is assigned a unique transaction identifier ($id$) and a start time $time_{id}$.

behaviour (e.g., value and temporal dependencies) will provide a good initial framework of reference. Without considerations for such dependencies between subtransactions of each global transaction, past correctness criteria provide weak consistency guarantees. Data manipulation in an MDB environment must take into consideration interdependency of data and control their access to ensure the preservation of interrelated data's mutual consistency. Specifying dependencies among MDB transactions must succinctly capture the dependency conditions, the data consistency requirements or constraints (state and temporal properties of the data), and consistency restoration mechanisms in case a violation is encountered.

Further, the specifications can be refined into lower level specifications that are eventually executable. Although, specifications are not necessarily executable, refining the specifications into executable forms further reduces the amount of coding that the programmers will do and also aids in the development of automated tools for proving and verifying the correctness of the specifications. Also, there is a need for a testbed environment to permit the implementation of the specifications to empirically determine their correctness and consistency.

Similar to the above, it will be worthwhile to develop a simulator for the specifications (that directly implements the specifications). This means we can execute the specifications to get statistics about the behaviour of the modelled system. It is possible to set breakpoints and display simulation results as the execution of the system progresses. Since most implementation languages provide mechanisms for making system calls to primitive constructs that allow the creation and termination (and other concurrency aspects) of concurrent processes the implementation stage will be relatively straight forward. Examples of low level system call are **fork** and **join** for specifying concurrency (available in most UNIX implementations[4] ). When a **fork** instruction is executed by a process, a new process (child) is created; this child process can load another program code that executes concurrently with its parent (the creator)[5]. To synchronize with completion of a child process, the parent process can execute a **join** operation thereby

[4]Microsoft's *Windows NT* operating system provides similar primitives.

[5]The **fork** instruction is synonymous to the *SPAWN* function defined in Chapter 5, Definition 28.

to capture the different aspects of the system in a unified framework that can be verified and implemented in the testbed.

# ≪ THE END ≫

| Symbols | Meaning |
|---|---|
| $LHS == RHS$ | Definition of $LHS$ as syntactically equivalent to $RHS$. A definition is distinguished from an equality ('=') syntactically by the use of the symbol '=='. A definition defines the left side to be equivalent to the right side, while an equality is a predicate that is either true or false. |
| $Op \triangleq T$ | $Op$ is defined by $T$.<br>$\triangleq$ is really a shorthand notation for definition. Note that $\triangleq$ and $==$ can be used interchangeably. |
| $fn : atype \longrightarrow otype$ | A function declaration where the domain constructor $\longrightarrow$ is read as "produces". The function name is $fn$ while its input and output are *atype* and *otype* respectively. |
| $\mid fn : atype \longrightarrow otype$ | A function declaration where only its signature is specified. |
| $x : T$ | A declaration, $x : T$, introduces a new variable $x$ of type $T$. This should be distinguished from from the membership test, $x \in T$, which is a predicate that is either true or false. $T$ must be a nonempty set, consequently if $x$ has been declared to be of type $T$, $x : T$, then $x \in T$ must be true. |
| $x : T; p : Op; \cdots; a : Data$ | List of declarations. |
| $p, q, \cdots, r : Op$ | $== p : Op; q : Op; \cdots; r : Op$ |
| $[NODE, EVENTS]$ | Introduces free types named *NODE* and *EVENTS*. They are distinct new types whose structure is unconstrained by this introduction. |

## A.2 Logic

The following are logic definitions. Let $a, p, q, x$, and $y$ be expression terms, $D$ be a declaration, and $P$ and $Q$ be predicates.

| Symbols | Meaning |
|---|---|
| $true, false$ | Boolean logical constants. |
| $\neg P$ | Not $P$ — Negation. |
| $P \wedge Q$ | $P$ and $Q$ — Conjunction. |
| $P \vee Q$ | $P$ or $Q$ — Disjunction. |
| $P \Rightarrow Q$ | $P$ implies $Q$ or **if** $P$ **then** $Q$ — Implication. This is equivalent to $== (\neg P) \vee Q$ |
| $P \Leftrightarrow Q$ | $P$ is logically equivalent to $Q$ — Equivalence $== (P \Rightarrow Q) \wedge (Q \Rightarrow P)$ |
| $\exists x : T \bullet P$ | There exists an $x$ of type $T$ such that $P$ holds — Existential quantification. |
| $\forall x : T \bullet P$ | For all $x$ of type $T$ such that $P$ holds — Universal quantification. |
| | The scope of the variable $x$ is the quantified predicate $P$. This scope extends as far to the right as possible so parentheses are used to delimit the scope. Furthermore, all quantifiers specify the type of the bound variable thereby defining the the values over which the quantification ranges explicitly. |
| $\exists_1 x : T \bullet P$ | There exists a unique $x$ of type $T$ such that $P$ holds — Unique existence. $\Leftrightarrow \exists x : T \bullet P \wedge \neg (\exists y : T \bullet x \neq y \wedge P)$ Note that $y$ is a variable not the same as $x$ and must not occur in $P$ |
| $\exists D \mid P \bullet Q$ | $\Leftrightarrow \exists D \bullet P \wedge Q$ There exists a type $D$ that satisfies the constraint (or predicate) $P$ such that the predicate $Q$ holds. |
| $\forall D \mid P \bullet Q$ | $\Leftrightarrow \forall D \bullet P \Rightarrow Q$ |
| $p = q$ | Equality between terms |

| Symbols | Meaning |
|---|---|
| $p \neq q$ | $\Leftrightarrow \neg (p = q)$ |
| $\forall x : T; p : Op; \cdots; a : Data \bullet P$ | For all $x$ of type $T$, $p$ of type $Op, \ldots$, and $a$ of type $Data$ such that the predicate $P$ holds. |
| $\exists x : T; p : Op; \cdots; a : Data \bullet P$ | Definition similar to $\forall$ |

## A.3 Numbers

| Symbols | Meaning |
|---|---|
| $\mathbb{Z}$ | The set of integers — this includes positive, zero, and negative integers. |
| $\mathbb{N}$ | The set of natural numbers — this is the set on non-negative integers.<br>$== \{n : \mathbb{Z} \mid n \geq 0\}$ |
| $\mathbb{N}_1$ | The set of strictly positive natural numbers<br>$== \mathbb{N} \setminus \{0\}$ |
| $m \cdots n$ | The set of integers between $m$ and $n$ inluding both $m$ and $n$. |
| $\mathbb{R}^+$ | The set of positive real numbers. |

## A.4 Sets

Let *Sets*, $S$, $T$, and $X$ be sets, $P$ a predicate, $D$ a declaration, and $t$ a term.

| Symbols | Meaning |
|---|---|
| $t \in S$ | $t$ is a member of $S$ — Set membership |
| $t \notin S$ | $\Leftrightarrow \neg (t \in S)$ |
| $\{\}$ or $\varnothing$ | The empty set |
| $S \subset T$ | $\Leftrightarrow (\forall x : S \bullet x \in T \land S \neq T$<br>Strict set inclusion. |
| $\{x : T \mid P\}$ | The set containing exactly those $x$ of type $T$ that satisfies the predicate $P$. |
| $\{D \mid P \bullet t\}$ | The set of values of the term $t$ for the variable declared in $D$ that ranges over all values for which $P$ holds. |
| $\mathbb{P}\, S$ | The set of all subsets of $S$ — Powerset |
| $\mathbb{P}_1\, S$ | The set of all nonempty subsets of $S$<br>$== \mathbb{P}\, S \setminus \{\varnothing\}$ |
| $\mathbb{F}\, S$ | The set of finite subsets of $S$<br>$== \{T : \mathbb{P}\, S \mid T$ is finite $\}$ |
| $\mathbb{F}_1\, S$ | The set of finite subsets of $S$<br>$== \mathbb{F}\, S \setminus \{\varnothing\}$ |
| $S \cap T$ | Set interscetion<br>$== \{x : X \mid x \in S \land x \in T\}$ |
| $S \cup T$ | Set union<br>$== \{x : X \mid x \in S \lor x \in T\}$ |
| $S \setminus T$ | Set difference<br>$== \{x : X \mid x \in S \land x \notin T\}$ |

| Symbols | Meaning |
|---|---|
| $\bigcap Sets$ | Intersection of a set of sets.<br>$== \{x : X \mid (\forall S : Sets \bullet x \in S)\}$ |
| $\bigcup Sets$ | Union of a set of sets.<br>$== \{x : X \mid (\exists S : Sets \bullet x \in S)\}$ |
| $\#S$ | The number of distinct members of a finite set.<br>— The cardinality. |

## A.5 Relations, Functions, and Sequences

Let $C, X$ and $Y$ be sets; $x : X$; $y : Y$; $p$ and $q$ be functions; $a_i$ and $b_i$ be terms; and $A$ and $B$ are sequences.

| Symbols | Meaning |
|---|---|
| $X \leftrightarrow Y$ | The set of relations between $X$ and $Y$.<br>$== \mathbb{P}(X \times Y)$ |
| $(x\ ,\ y)$ | The relation relating $x$ to $y$<br>$== x \mapsto y$ |
| dom $R$ | The domain of a relation |
| ran $R$ | The range of a relation |
| $p(x)$ | The function $p$ applied to variable $x$. $p(x)$ is defined if and only if $x \in \text{dom}\, p$. Its value is the unique value in range associated with the value $x$ in its domain. That is, a function is a set of pairs with each member of its domain associated with a unique member of its range. Thus,<br>$p(x) = y \Leftrightarrow (x, y) \in p)$ |
| $p \circ q$ | Functional composition.<br>$==\ q\ ;\ p$ |

| Symbols | Meaning |
|---|---|
| $X \nrightarrow Y$ | The set of total functions from $X$ to $Y$<br>$==$ $\{\text{rel: } X \leftrightarrow Y \mid (\forall x : \text{dom rel} \bullet (\exists_1 y : Y \bullet x \text{ rel } y))\}$ |
| $X \longrightarrow Y$ | The set of total functions from $X$ to $Y$<br>$==$ $\{\text{rel: } X \nrightarrow Y \mid (\text{dom rel} = X$ |
| seq $X$ | The set of finite sequences whose elements are drawn from $X$<br>$==$ $\{A : \mathbb{N}_1 \nrightarrow X \mid (\exists n : \mathbb{N} \bullet \text{dom } A = 1 \ldots n)\}$ |
| $\langle a_1, a_2, \cdots, a_n \rangle$ | $= \{1 \mapsto a_1, 2 \mapsto a_2, \cdots, n \mapsto a_n\}$ |
| $\#X$ | The length of sequence $X$. |
| $\langle\,\rangle$ | The empty sequence.<br>$==$ $\{\,\}$ |
| $\text{seq}_1 X$ | The set of non-empty sequences.<br>$==$ $\{s : \text{seq } X \mid s \neq \langle\,\rangle$ |
| $\langle a_1, a_2, \cdots, a_n \rangle \frown \langle b_1, b_2, \cdots, b_m \rangle$ | Concatenation.<br>$= \langle a_1, a_2, \cdots, a_n, b_1, b_2, \cdots, b_m \rangle$ |
| $head(A)$ | The first element of a nonempty sequence.<br>$A \neq \langle\,\rangle \Rightarrow head(A) = A[1]$ |
| $last(A)$ | The last element of a nonempty sequence.<br>$A \neq \langle\,\rangle \Rightarrow last(A) = A[\#A]$ |
| $tail(A)$ | The sequence with the first element of a nonempty sequence removed.<br>$A \neq \langle\,\rangle \Rightarrow tail(A) = A - head(A)$ |
| $A \upharpoonright C$ | The restriction of the range of sequence $A$ to elements from the set $C$. |

| Symbols | Meaning |
|---|---|
| $A$ in $B$ | $A$ is a continguous subsequence of $B$. |
| $A$ **prefix** $B$ | $A$ is a continguous subsequence of $B$ such that the head of $A$ and $B$ are the same. $\Leftrightarrow \exists B_0 : \text{seq}\, B \bullet A \frown B_0 = B$ <br> *Note*: The operator **prefix** differs from in because in may hold for *any* continguous subsequence of any given sequence whereas **prefix** holds only when the head of *both* the subsequence and given sequence is the same. |

## A.6 TCSP symbols

The details of these operators and symbols are discussed in Chapter 4.

| Symbols | Meaning |
|---|---|
| $\alpha P$ | Events $P$ can engage in — the alphabets of $P$ |
| $STOP$ | Deadlock |
| $SKIP$ | Successful termination |
| $WAIT$ | Delayed termination |
| $\longrightarrow$ | Prefixing. It is also used as an event transition relation. |
| ; | Sequential composition |
| $WAIT\ t; P$ | Delay for $t$ units of time and then execute $P$ |
| $\Box$ | external choice |
| $\mid$ | Internal choice |

| Symbols | Meaning |
|---|---|
| $\sqcap$ | Nondeterministic choice |
| $\parallel$ | Parallel execution; must be synchronized on every event. |
| $\|\|\|$ | Interleaving; no synchronization is required. |
| $\backslash$ | Hiding |
| $c?a$ | Input the variable $a$ via channel $c$ |
| $c!a$ | Output the value of $a$ via channel $c$ |
| $c?a : A \longrightarrow P_a$ | Prefix choice |
| $\mu X \bullet P$ | Recursion |
| $trace(P)$ | $==$ seq $x \mid x : EVENTS \bullet x \in \alpha P$ |
| $P \not< bool \not> Q$ | If *bool* is true then execute $P$ otherwise execute $Q$. Note that *bool* must be a logical expression that evaluates to either *true* or *false*. This is another form of the familiar **if-then-else** construct. |
| $P \overset{t}{\triangleright} Q$ | Timeout program |
| $P \overset{\nabla}{_i} Q$ | Event interrupt |
| $P \overset{\lightning}{_t} Q$ | Timed interrupt |
| **sat** | Satisfy relation |
| $\oplus$ | Record data type constructor |
| . | Record data type's member element accessing operator. |

# References


[Aal92] van der Aalst W.M.P. "Timed Coloured Petri Nets and their Application to Logistics", *Ph.D Thesis*, Eindhoven University of Technology, 1992.

[ABL89] Ackerman A. F., Buchwald L. S., and Lewshi F. H., "Software Inspections: An Effective Verification Process", *Software*, Vol. 6, No.3, May 1989, pp 31-36.

[ACL87] Agrawal R., Carey J. M., and Livny M., "Concurrency Control Performance Modelling: Alternatives Implications", *ACM Transactions on Database Systems*, #12, 1987, pp 609-654.

[AE92] Agrawal D. and El Abbadi A. "Transaction Management in Database Systems". In A.K Elmagarmid (ed), *Database Transaction Models for Advanced Applications*. Morgan- Kaufmann, 1992.

[AFLMW88] Aspenes J., Fekete A., Lynch N., Merritt M., and Weihl W. "A Theory of Timestamp-based Concurrency Control for Nested Transactions". *Proceedings of the 14th VLDB Conference*, Los Angeles, California, 1988.

[AM95] Alfaro L. and Manna Z. "Verification in Continuous Time by Discrete Reasoning". In Alagar V.S. and Nivat M. (Eds), *Algebraic Methodology and Software Technology*. Proceedings of 4th International Conference, AMAST' 95, Montreal, Canada, July 1995, LNCS 936, Springer 1995.

[AJR95] Ammann P., Jajodia S., and Ray I., "Using Formal Methods to Reason about Semantics-based Decomposition of Transactions", In *Proceedings of the International Conference on Very Large Databases*, Zurich, Switzerland, September 1995, pp 218-227.

[ANSI84] ANSI / IEEE STD-830-1984, *IEEE Guide to Software Requirements Specifications*. IEEE Software Engineering Technical Committee, IEEE Standard for Requirements Specification, STD-830-1984.

[Anw92] Anwar E. *Supporting Complex Events and Rules in an OODBMS: A Seamless Approach*. Master's Thesis, Database Systems R and D Center, Department of Computer and Information Sciences, University of Florida, Gainesville, Florida 32611, November 1992.

[Bak95] Baker, William Douglas. *Trigger Management in Active Multidatabase Systems*. Master's Thesis, Department of Computer Science, University of Manitoba, Winnipeg, Canada, 1995.

[Bar90] Barker K.E. *Transactions Management on Multidatabase Systems.* Ph.D. Thesis, Department of Computer Science, The University of Alberta, Edmonton, Alberta, Canada, Fall 1990.

[Ber88] Bergstra J.A., "Process Algebra for Synchronous Communication and Observation", Technical Report No. P8815, Programming Research Group, University of Amsterdam, 1988.

[Ber90] Bernestein P.A. "Transaction Processing Monitors". *Comm. ACM*, Vol.33, November 1990, pp 75-86.

[BB91] Baeten J.C.M. and Bergstra J.A., "Real Time Process Algebra", *Formal Aspects of Computing*, Vol. 3, No. 2, 1991, pp 142 - 148.

[BG80] Burstall R.M., and Goguen J.A., "The Semantics of Clear. A Specification Language". In *Proc. 1979 Copenhagen Winter School Abstract Software Specification*, Lecture Notes in Computer Science 86, Springer-Verlag, 1980.

[BG89] Bernstein P.A. and Goodman N., "Concurrency Control in Distributed Database Systems". *ACM Computing Surveys*, Vol. 13(2), April 1989, pp 230-269.

[BGH87] Bernstein P.A., Goodman N., and Hadzilacos V. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading Massachusetts, 1987.

[BGHL87] Birell A.D., Guttag J.V., Horning J.J., and Levin R. "Synchronization Primitives for a Multi-processor: A formal Specification". SRC Research Report No.20, August 1987.

[BH94] Bowen J.P and Hinchey M.G, "Seven More Myths of Formal Methods". Proceedings of FME'94 Symposium, *Industrial Benefit of Formal Methods*, Barcelona, Spain, 24-28 October 1994, Lecture Notes in Computer Science, Springer-Verlag, 1994.

[BHMC90] Buchmann A., Hornick M., Markatos E., and Chronaki C. "Specification of a Transaction Mechanism for Distributed Active Object System". In *Proceedings of the OOPSLA/ECOOP 90 Workshop on Transactions and Objects*, Ottawa, Canada, October 1990, pp 1-9.

[BK91] Barghouti N. S. and Kaiser G. E., "Concurrency Control in Avanced Database Applications", *ACM Computing Surveys*, Vol. 23, 1991, pp 269-317.

[BL93] Bernstein A. J. and Lewis P. M., *Concurrency in Programming and Database Systems*, Jones and Bartlett Publishers, London, 1993.

[Blu92] Blum B. I., *Software Engineering: A Holistic View.* Oxford University Press, Oxford, 1992.

[BO90] Barker K. E. and Ozsu M. T., "Concurrent Transaction Execution in Multidatabase Systems", *Proceedings of COMPSAC'90*, Chicago, Illinois, October 1990, pp 282-288.

[Cla92] Claybrook B. G., *OLTP : Online Transaction Processing Systems*, New York : John Wiley, 1992.

[CM87] Clocksin W.F and Mellish C.S. *Programming in Prolog, (Third Edition)*. Springer-Verlag, 1987.

[CM91] Chakravarthy S. and Mishra D. "An event Specification Language (Snoop) for Active Databases and its detection". Technical Report No. UF-CIS-TR-91-23, University of Florida, Department of Computer and Information Sciences, Gainesville, Florida 32611, September 1991.

[Col90] Coleman D. "The Technology Transfer of Formal Methods: What's Going Wrong?". *Proc. 12th ICSE Workshop on Industrial Use of Formal Methods.* Nice, France, March 1990

[CR90] Chrysanthis P.K. and Ramamritham K. "ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behaviour". *ACM SIGMOD Record*, Vol. 19, No. 2, June 1990.

[CR91] Chrysanthis P.K. and Ramamritham K. "A Formalism for Extended Transaction Models". *Proceedings of the 17th VLDB Conference*, Barcelona, Spain, 1991.

[CR92a] Chrysanthis P.K. and Ramamritham K. "ACTA: The SAGA Continues". In A.K Elmagarmid (ed), *Database Transaction Models for Advance Applications.* Morgan-Kaufmann, 1992.

[CR92b] Chrysanthis P.K. and Ramamritham K. "In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties". *Pre-proceedings of the International Workshop on Distributed Object Management*, Edmonton, Canada, August 1992.

[Dav88] Davis A. M. "A Comparison of Techniques for the Specification of External System Behaviour". *Comm. ACM* Vol. 31, No.9, September 1988.

[DC96] De Giacomo G.D. and Chen X.J., "Reasonng about Nondeterministic and Concurrent Actions: A Process Algebra Approach." In *Proceedings of National Conference on Artificial Intelligence (AAAI-96)*, 1996.

[DE89] Du W. and Elmagarmid A., "Quasi-Serializability: A Correctness Criterion for Global Concurrency Control in InterBase", In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB)*, Amsterdam, August 1989, pp 347-355.

[DeM78] DeMarco T., *Structured Analysis and System Specification.* Englewood Cliffs, NJ, Yourdon Press / Prentice Hall, 1978.

[Des90] Desai B.C. *An Introduction to Database Systems.* West Publishing Company, St. Paul, Minnesota, 1990.

[DK83] Dasgupta P. and Kedem Z.M. "A Non-two-phase Locking Protocol for Concurrency Control in General Databases". *Proc. of the 9th VLDB Conference*, Florence, Italy, 1983.

[DS9*] Davies J. and Schneider S.A. "Real-Time CSP: Processes and Properties"

[DS92a] Davies J. and Schneider S.A. "A Brief History of Timed CSP". Technical Monograph PRG-96, Oxford University, 1992.

[DS92b] Davies J. and Schneider S.A. "Using CSP to Verify a Timed Protocol Over a Fair Medium". In *Proceedings of CONCUR 92, LNCS 630.* Springer-Verlag, 1992.

[EC75] Eswaran K.P. and Chamberlin D.D. "Functional Specification of a Subsystem for Database Integrity". *Proceedings of VLDB*, Framingham, Massachusetts, U.S.A., Vol. 1. No. 1, Sept. 1975.

[EC90] Elmagarmid A.K. and Calton P. "Guest Editors' Introduction to the Special Issue on Heterogeneous Databases". *ACM Computing Surveys*, Vol. 22, No.3, Sept. 1990, pp 183 - 236.

[EB93] Ehikioya S.A. and Barker K.E., "A call for the Formal Specification of Transaction Systems Protocols: A Survey". In Perrizo W. and Goli V.N.R. (Editors), *Proceedings of the First Annual Mid-Continent Information Systems Conference (MISC-93)*, May 3-4, 1993. Fargo, North Dakota, USA. Pages 258 - 268.

[EBO95] Ehikioya S.A., Barker K.E., and Onibere E.A., "Specifying Correctness in the Automation of Banking Operations". In Adagunodo E.R., Kehinde L.O., Akinde A.D., and Adigun M.O. (Eds.) *Computer-Based Automation in Developing Countries (Auto-DC '95).* Lagos, Nigeria. COAN Conference Series, Vol. 6, May 1995, pp 103 - 115.

[EGLT76] Eswaran K., Gray J., Lorie R., and Traiger I. "The Notion of Consistency and Predicate Locks in a Database System". *Comm. ACM*, Vol. 19, No.11, November 1976, pp 624-633.

[EN94] Elmasri R. and Navathe S.B. *Fundamentals of Database Systems (2nd Ed.).* The Benjamin / Cummings Publishing Company Inc., 1994.

[Fai85] Fairley R.E. *Software Engineering Concepts.* McGraw-Hill Inc., 1985.

[Fid94] Fidge C. "A Comparative Introduction to CSP, CCS, and LOTOS", Technical Report No. 93-24, Software Verification Research Center, Department of Computer Science, The University of Queensland, Australia, April 1994.

[Flo85] Floyd C., "On the Relevance of Formal Methods to Software Development". In Ehrig H., Floyd C., Nivat M., and Thatcher J. (Eds), *Formal Methods and Software Development, Vol. 2: Colloquium on Software Engineering, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT).* Berlin, Springer-Verlag, March 1985.

[FS93] Formal Systems (Europe) Limited. *Failure Divergence Refinement (FDR): User Manual and Tutorial (Version 1.3).* August 1993.

[Fuj90] Fujimoto R. M., "Parallel Discrete Event Simulation", *Comm. ACM*, Vol.30, No.10, Oct. 1990, pp 30-53.

[Geo91] Georgakopoulos D., "Multidatabase Recoverability and Recovery", In *Proceeding of 1st International Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, 1991, pp 348-355.

[Ger83] Gerhart S.L. "Program Specification". In Ralston A. and Reilly E.D. (Jr) (Edited), *Encyclopedia of Computer Science and Engineering (2nd. Edition)*. Van Nostrand Reinhold Company, 1983, pp 1243-1246.

[GF90] Garbrielian A. and Franklin M. K. "Multi-level Specification and Verification of Real-Time Software". *1990 IEEE 12th International Conference on Software Engineering*. 1990 pp 52-62.

[GGKKS90] Garcia-Molina H., Gawlick D., Klein J., Kleissner K., and Salem K. "Coordinating Multi-Transaction Activities". Tech. Report CS-TR-247-90, Dept. of Computer Science, Princeton University, February 1990.

[GH80] Guttag J.V and Horning J.J. "Formal Specifications as a Tool". In *Proc. 7th ACM Symposium on Principles of Programming Languages*. Las Vegas, 1980, pp 251 - 261.

[GHW82] Guttag J.V., Horning J.J., and Wing J.M. "Some Remarks on Putting Formal Specifications to Productive Use". *Science of Computer Programming*. North-Holland, Vol. 2, No. 1, October 1982.

[GHW85] Guttag J.V., Horning J.J., and Wing J.M. "Larch in Five Easy Pieces". Tech. Report 5, DEC Systems Research Center, July 1985.

[GJ91] Gehani N.H. and Jagadish H.V. "Ode as an Active Database: Constraints and Triggers". In *Proceedings 17th International Conference on VLDB*. Bacelona, Spain, September 1991, pp 327-336.

[GJS92] Gehani N.H., Jagadish H.V, and Shmueli O. "Event Specification in an Object-Oriented Database". In *Proceedings International Conference on Management of Data*. San Diego, CA, U.S.A,.June 1992, pp 81-90.

[GM94] Gunter C.A. and Mitchell J.C (Eds.), *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1994.

[Gog88] Goguen J.A. "OBJ as a theorem prover with Applications to Hardware Verification". Tech. Report SRI-CSL-88-4R2, Stanford Research Inst., Menlo Park, California, August 1988.

[GR93] Gray J. and Reuter A., *Transaction Processing: Concept and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[Gra81] Gray J. "The Transaction Concept: Virtues and Limitations". In *Proceedings of The 7th International Conference on VLDB*. 1981.

[Gra94] Graham P.C.J. *Applications of Static Analysis to Concurrency Control and Recovery in Objectbase Systems*. Ph.D Thesis, Department of Computer Science, University of Manitoba, Winnipeg, Canada, 1994.

[GS87] Gracia-Molina H. and Salem K. "Sagas". In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* May 1987.

[Har88] Harel D. "On Visual Formalism". *Comm. ACM* Vol.31 (5), 1988, pp. 514-530.

[HG97] Harel D. and Gery E. "Executable Object Modeling with Statecharts", *Computer*, IEEE, July 1997, pp. 31-42.

[Hay84] Hayes C.T., "A Theory of Data Type Representation Independence", In Kahn G., MacQueen D.B., and Plotkin G. (Eds.) *Semantics of Data Types International Symposium Proceedings*, Sophia-Antipolis, France, June 1984. *Lecture Notes in Computer Science*, Vol 173, Springer-Verlag, New York, NY, pp. 157-175.

[HJ95] Hinchey M. G. and Jarvis S. A., *Concurrent Systems: Formal Development in CSP.* McGraw-Hill Book Company, 1995.

[HK91] Huston I. and King S. "CICS Project Report: Experiences and Results from the use of Z in IBM". In Prehn S. and Toetenel W.J (Eds): *VDM '91, Formal Software Development Methods.* Springer-Verlag, LNCS 551, PP 588-603, 1991.

[HO93] Hull M.E.C and O'Donoghue P.G. "Family Relationships Between Requirements and Design Specification Methods". *The Computer Journal*, Vol. 36, No.2, 1993.

[Hoa85] Hoare C.A.R. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[How94] Howles F. *Distributed Arbitration in the IEEE Futurebus Protocol.* M.Sc Thesis, Oxford University Computing Lab. Programming Research Group, UK, 1994.

[HPS93] Haghjoo M.S, Papazoglou M.P., and Schmidt H.W. "A Semantic-based Nested Transaction Model for Intelligent and Cooperative Information Systems". In *Proceedings of International Conference on Intelligent and Cooperative Information Systems*, Huhns M., Papazoglou M.P., and Schlageter G. (Edited), IEEE Computer Society Press, Los Alamitos, CA, USA, May 12- 14, 1993, pp 321-331.

[HR83] Haeder T. and Rueter A. "Principles of Transaction-Oriented Database Recovery". *ACM Computing Surveys*, Vol.15, No.4, Dec. 1983, pp 287-317.

[HU79] Hopcroft J. and Ullman J. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA 1979.

[HZ92] Hsu M. and Zhang B., "Performance Evaluation of Cautious Waiting", *ACM Transactions on Database Systems*, #17, 1992, pp 477-512.

[ISO87] International Standards Organization. "Information Systems Processing: Open Systems Interconnection - LOTOS". Tech. Report, 1987.

[Jac96] Jacobs M. A., *A Visual Query Language for a Federation of Databases*, Master's Thesis, Department of Computer Science, University of Manitoba, Winnipeg, Canada, 1996.

[Jar92] Jarvis Steve. *Using CSP to Verify a Reliable Network Protocol.* M.Sc Thesis, Programming Research Group, University of Oxford, U.K, Sept. 30, 1992.

[JDS85] Jackson M.I, Denvir B.T. and Shaw R.C. "Experience of Introducing the VDM into an Industrial Organization". In Ehrig H., Floyd C., Nivat M., and Thatcher J. (Eds), *Formal Methods and Software Development, Vol. 2: Colloquium on Software Engineering, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT).* Berlin, Springer-Verlag, March 1985.

[JM86] Jahanian F. and Mok A.K. "Safety Analysis of Timing Properties in Real-Time Systems". *IEEE Trans. on Software Engineering* SE-12 (9), Sept.1986.

[Jon86] Jones C.B. *Systematic Software Development Using VDM.* Prentice Hall International, 1986.

[JS90] Jones C.B and Shaw R.C. (Eds.) *Case Studies in Systematic Software Development.* Prentice Hall, 1990.

[Ken96] Kenney J. J., *Executable Formal Models of Distributed Transaction Systems Based on Event Processing*, Ph.D Thesis, Stanford University, U.S.A., June 1996.

[KJ90] Kogan B. and Jajodia S. "Concurrency Control in Multilevel-Secure Databases based on Replicated Architecture". *ACM SIGMOD Record*, Vol. 19 (2), June 1990.

[KKB88] Korth H.F., Kim W., and Bancilhon F. "On Long-duration CAD Transactions". *Information Sciences*, Vol. 46, No.1-2, Oct-Nov. 1988, pp 73-107.

[KL83] Kiessling W. and Landherr G. "A Quantitative Comparison of Lock Protocols for Centralized Databases". *Proc. of the 9th VLDB Conference*, Florence, Italy, 1983.

[KL91] Kenny K.B. and Lin K., "Building Flexible Real-Time Systems Using the Flex Language", *Computer* Vol. 24, No.5, May 1991, pp 70-78

[Kna87] Knapp E. "Deadlock Detection in Distributed Databases". *ACM Computing Surveys*, Vol. 19, Dec. 1987.

[KS86] Korth H.F. and Silberschatz A. *Database System Concepts.* McGraw-Hill Inc., 1986.

[KS88] Korth H.F. and Speegle G. "Formal Models of Correctness without Serializability". In Proceedings of the ACM SIGMOD *International Conference on Management of Data*, Chicago Illinois, June 1988, pp 379-386.

[KS92] Keller M. and Shumate K., *Software Specification and Design: A Discipline Approach for Real-Time Systems.* John Wiley and Sons Inc., 1992.

[KS94] Korth H.F. and Speegle G. "Formal Aspects of Concurrency Control in Long-Duration Transaction Systems Using the NT/PV Model". *ACM Trans. on Database Systems*, Vol. 19, No.3, Sept. 1994, pp 492-535.

[Lam78] Lamport L., "Time, Clocks, and the Ordering of Events in a Distributed System", *Comm. of the ACM*, Vol. 21, No.7, 1978, pp 558-565.

[Lap89] Laprie J.C. "Dependability: A Unifying Concept for Reliable Computing and Fault Tolerance". In Anderson T. (Ed.), *Dependability of Resilient Computers, Chapter 1.* Blackwell Scientific Publications, Oxford, 1989, pp 1-28.

[LH85] Luckham D.C. and von Henke F.W. "An overview of Anna, A Specification Language for Ada". *IEEE Software*, Vol. 2, No. 2, March 1985.

[LM93] Li Pei-yu and McMillin Bruce. "Formal Model and Specification of Deadlock". Tech. Report CSC-93-31, Dept. of Comp. Science, University of Missouri-Rolla, Rolla, Missouri, August 1993.

[LM94] Li Pei-yu and McMillin Bruce. "Formal Verification of Distributed Deadlock Detection Algorithm Using a Time Dependent Proof Technique". Tech. Report CSC-94-06, Dept. of Computer Science, University of Missouri-Rolla, Rolla, Missouri, 1994.

[LMR90] Litwin W., Mark L., and Roussopoulos N. "Interoperability of Multiple Autonomous Databases". *ACM Computing Surveys*, Vol. 22, No.3, Sept. 1990, pp 183 - 236.

[LMWF94] Lynch N., Merritt M., Weihl W., and Fekete A. *Atomic Transactions.* Morgan Kaufmann Pub. Inc., San Mateo, CA., 1994.

[LN93] Leestma S. and Nyhoff L., *Turbo Pascal: Programming and Problem Solving*, (Second Edition), Macmillan Publishing Company, 1993.

[Low93] Lowe G., *Probabilities and Prorities in Timed CSP.* Ph.D Thesis, University of Oxford, Hilary Term, 1993.

[Low94] Lowe G., "Formal Development of Aircraft Control Software: A Case Study in the Specification, Design and Implementation of a Real-Time System", Technical Report PRG-TR-15-94, Oxford University Computing Laboratory, 1994.

[LR91] Lo M. and Ravishankar C.V., "A Concurrency Control Protocol for Nested Transactions", Technical Report #: CSE-TR-96-91, Computer Science and Engineering Division, Dept. of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan 48109-2122, 1991.

[LR82] Landers T. and Rosenberg R. L., "An Overview of Multibase", In Schneider H. J. (Ed.), *Distributed Data Bases*, New York, North Holland, 1982, pp 153-188.

[LS85] Leveson N.G and Stolzy J.L. "Analyzing Safety and Fault Tolerance Using Time Petri Nets". In Ehrig H., Floyd C., Nivat M., and Thatcher J. (Eds), *Formal Methods and Software Development, Vol. 2: Colloquium on Software Engineering, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT).* Berlin, Springer-Verlag, March 1985.

[LS87] Leveson N.G and Stolzy J.L. "Safety Analysis Using Petri-nets". *IEEE Trans. on Software Engineering*, No.13, PP: 386-397, 1987.

[Lus94] Lustman Francois. "Specifying Transaction Based Information Systems with Regular Expressions". *IEEE Trans. on Software Engineering*, Vol.20 No.3, March 1994.

[Mar91] Marateck S.L., *PASCAL*, John Wiley and Sons, Inc., 1991

[McC83] McCracken D.D. "Procedure-Oriented Languages: Survey". In *Encyclopedia of Computer Science and Engineering (2nd Edition)*, Ralston A. and Reily E.D. (Jr) (Edited). Van Nostrand Reinhold Company, 1983.

[MCFP96] Morpain C., Cart M., Ferrie J., and Pons J., "Maintaining Database Consistency in Presence of Value Dependencies in Multidatabase Systems", In *SIGMOD RECORD*, Vol. 25, No.2., June 1996, Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.

[Mic96] Microsoft Corporation, *ODBC 3.0 : Programmer's Reference*, 1996.

[Mil80] Milner R. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

[Mis91] Mishra D. *Snoop: An event Specification Language Active Databases.* Master's Thesis, Database Systems R and D Center, Department of Computer and Information Sciences, University of Florida, Gainesville, Florida 32611, August 1991.

[MLTL93] McMillin B., Lutfiyya H., Tsai G., and Liu J. "Parallel Algorithm Fundamentals and Analysis". Tech. Report No. CSC-93-17, Dept. of Computer Science, University of Missouri-Rolla, 1993.

[MM79] Menasce D.A. and Muntz R.R. "Locking and Deadlock Detection in Distributed Databases". *IEEE Trans. Software Engineering*, Vol. SE-5, May 1979.

[Mos81] Moss J.E.B. *Nested Transactions: An approach to Reliable Distributed Computing.* Ph.D Thesis, Massachusetts Institute of Technology, Cambridge, MA, April 1981.

[Mos85] Moss J.E.B. *Nested Transactions: An approach to Reliable Distributed Computing.* The Massachusetts Institute of Technology Press, Cambridge, MA, 1985.

[Mot89] Motrol Amihai. "Integrity = Validity + Completeness". *ACM Transaction on Database Systems*, Vol.14 No.3, Sept. 1989, pp 480-502.

[Nie89] Nielsen M., *et al.* "The Raise Language, Methodology, and Tools". *Formal Aspects of Computing*, Vol. 1, 1989.

[NJH92] Nico Plat, Jan van Katwijk, and Hans Toetenel. "Application and Benefits of Formal Methods in Software Development". *Software Engineering Journal*, September 1992.

[NPL91] National Pysical Laboratory. "Formal Description Techniques and Security Standard Conformance Testing". NPL Report DITC 175/91, March 1991.

[NW90] Neel Madhav and Walter Mann. "A Methodology for Formal Specification and Implementation of Ada Packages". In *Proc. of 14th Annual International Computer Software and Applications Conference.* IEEE Computer Society Press, 1990.

[Obe82] Obermarck R. "Distributed Deadlock Detection Algorithm". *ACM Transaction on Database Systems*, Vol. 7, June 1982.

[Ost94] Ostroff J.S. *Formal Methods for the Specification and Design of Real-Time Safety Critical Systems.* (Draft 2.0 for IEEE Press book to be called "Tutorial on Specification of Time"), May 1994. Also appeared in *Journal of Systems and Software*, April 1992.

[Ost91] Ostroff J.S. *Survey of Formal Methods for the Specification and Design of Real-time Systems.* (Draft 1.0 for IEEE Press book to be called "Tutorial on Specification of Time"), May 1991.

[OV94] Ozsu M. T. and Valduriez P., "Distributed Data Management: Unsolved Problems and New Issues", In Casavant T. and Singhal M. (Eds.) *Readings in Distributed Computing Systems*, IEEE/CS Press, 1994, pp 512-544.

[OV91] Ozsu M.T. and Valduriez P. *Principles of Distributed Database Systems.* Prenctice Hall, 1991.

[Pag81] Pagan F.G., *Formal Specification of Programming Languages: A Panoramic Primer.* Englewood Cliffs, N.J.: Prentice-Hall. 1981.

[Pap86] Papadimitriou C.H. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[Pet77] Peterson J.L. "Petri nets". *Computing Surveys*, Vol. 9, No. 3 September 1977.

[PNM91] International Workshop on Petri Nets and Performance Models. *Petri Nets and Performance Models.* Proceedings of the Fourth International Workshop on PNPM 91, Melbourne, Australia, Dec. 2 - 5, 1991. IEEE Computer Society, 1991.

[PNS96] ISO/IEC JTC1/SC7/WG11, "Petri Nets Standard", 1996. (Available at: *http://www.daimi.aau.dk/PetriNets/standard/*).

[Pre92] Pressman R.S. *Software Engineering: A Practitioner's Approach* (Third Edition). McGraw-Hill Inc., 1992.

[Pu86] Pu C. *Replication and Nested Transactions in the Eden Distribution System.* Ph.D Thesis, University of Washington, 1986.

[Ros84] Rosenberg J. M. *Dictionary of Computers, Data Processing and Telecommunications.* John Wiley and Sons, 1984.

[RU71] Rescher N. and Urquhart A. *Temporal Logic.* Springer-Verlag, Library of Exact Philosophy, 1971.

[Rum91] Rumbaugh J. *et al.*, *Object-Oriented Modelling and Design.* Prentice Hall, 1991.

[SC88] Stachowitz R.A. and Chang C. *Verification and Validation of Expert Systems: Tutorial Program SP2.* AAAI, 1988.

[Sch9*] Schneider S.A. "Rigorous Specification of Real-Time Systems".

[ST90] Sannella D. and Tarlecki A. "Algebraic Specification and Formal Methods for Program Development: What Are The Real Problems". *EATCS Bulletin* No.41, 1990.

[Sto82] Stoy J., "Some Mathematical Aspects of Functional Programming." In Darlington J., Henderson P., and Turner D.A (Eds.) *Functional Programming and Its Applications.* Cambridge University Press, 1982, pp 217 - 252.

[Tel94] Tel Gerard. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

[TL82] Tsichritzis D. C. and Lochovsky F. H. *Data Models.* Prentice-Hall, Inc., 1982.

[TM91] Tsai Sumei and McMillin Bruce. "Formal Methods of Real-Time Systems". Tech. Report No. CSC-91-17, Dept. of Computer Science, University of Missouri at Rolla, Rolla, Missouri, August 6, 1991.

[TMW89] *The New Mariam-Webster Dictionary.* Mariam-Webster Inc. Publishers, Springfield, Massachusetts, 1989.

[Ver78] Verhofstad J. "Recovery Techniques for Database Systems". *ACM Computing Surveys*, Vol. 10, No.2, June 1978, pp 167-196.

[VW87] Voung S.T. and Weber J.F. "Protocol Specification and Validation Using Prolog". In *Proceedings of Intelligence Integration.* CIPS Edmonton, Canada, November 16-19, 1987, pp 167-175.

[Wal90] Walshe Ann. "NDB: The Formal Specification and Rigorous Design of a Single-user Database System". Chapter 2 in [JS90].

[Was80] Wasserman A.I., "Introduction to Data Types". In Wasserman A.I. (Ed), *Tutorial Programming Language Design: CompSac 80. IEEE Computer Society 4th International Computer Software and Applications Conference*, IEEE Computer Society Press, New York 1980, pp 184 - 188.

[WC93] Wilson L.B. and Clark R.G., *Comparative Programming Languages*, (Second Edition), Addison-Wesley, 1993.

[WD97] ISO/IEC JTC1/SC7/WG11, "High-level Petri Nets Standard: Working Draft Version 2.1", *Project 7.19.3 — Petri Nets*, February 18, 1997. (Available at: *http://www.itr.unisa.edu.au/tsec/sections/standard.html*).

[Wei88] Weihl W. "Commutativity-Based Control for Abstract Data Types". *IEEE Trans. on Computers*, Vol. 37, No.12, Dec. 1988, pp 1488-1505.

[WF83] Wotschke D. and Fischer P.C. "Well-Formed Formula". In Ralston A. and Reilly E.D. (Jr) (Edited), *Encyclopedia of Computer Science and Engineering (2nd. Edition).* Van Nostrand Reinhold Company, 1983, pp 1568.

[Win90] Wing J.M. "A Specifier's Introduction to Formal Methods". *Computer* Vol. 23, No. 9, September 1990.

[WKC91] Wallace D.R., Kuhn D.R., and Cherniavsky J.C. *Report of the NIST Workshop of Standards for the Assurance of High Integrity Software: NIST Special Publication 500-190.* Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA, August 1991.

[Woo87] Wood D. *Theory of Computation.* John Wiley and Sons, 1987.

[Woo89] Woodcock J.C.P. "Calculating Properties of Z Specifications". *ACM Software Engineering Notes*, Vol. 14(5), July 1989, Page 43-54.

[YC79] Yourdon E. and Constantine L., *Structured Design.* Englewood Cliffs, NJ, Prentice Hall, 1979.

[YM85] Yonezawa A. and Matsumoto Y. "Object Oriented Concurrent Programming and Industrial Software Production". In Ehrig H., Floyd C., Nivat M., and Thatcher J. (Eds), *Formal Methods and Software Development, Vol. 2: Colloquium on Software Engineering, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT).* Berlin, Springer-Verlag, March 1985.

[Zwa96] Zwass V., "Electronic Commerce: Structures and Issues", *International Journal of Electronic Commerce*, Vol. 1, No. 1, Fall 1996, pp 3-23.