EXECUTABLE FORMAL MODELS OF DISTRIBUTED TRANSACTION SYSTEMS BASED ON EVENT PROCESSING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING DEPARTMENT
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
John Jerome Kenney
June 1996

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

David Luckham
(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Gio Wiederhold

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Arthur Keller
(Computer Science Department)

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Preface

This dissertation presents formal models of distributed transaction processing (DTP) that are executable and testable. These models apply a new technology, RAPIDE, an object-oriented executable architecture description language designed for specifying and prototyping distributed, time-sensitive systems. RAPIDE's execution model is based upon partially ordered sets of events (posets) and can represent *true concurrency.* This dissertation shows how the RAPIDE technology can be applied to specify, prototype, and test DTP models.

In particular, this dissertation specifies a *reference architecture* for the X/Open DTP industry standard. The standard describes several components and their interfaces, a few instances of a software architecture, as well as several protocols – including the two–phase commit protocol. The reference architecture, written in RAPIDE, defines architectures and behaviors of systems that comply with the X/Open standard. It contains machine processable definitions of the component interfaces, their behaviors, and how the components may be connected together into architectures. It also includes formal, checkable constraints on the individual component behaviors and on the communication protocols between the components.

An extension to the X/Open standard is also presented. We add to the standard's constraints and behaviors associated with the two–phase commit protocol for the property atomicity, the constraints and behaviors for the property isolation via the two–phase locking protocol. These DTP properties, protocols, and system architecture are interpreted in *views* that leaves a family of models (or framework) for testing DTP standards. Each view is a reference architecture that is formally defined and may be simulated, animated, and related to other views.

This dissertation also applies a technique developed previously by Gennart and Luckham [28] for testing applications for conformance with reference architectures. This technique is based upon *pattern mappings*. Pattern mappings specify relationships between architectures, and they permit an application's execution to be automatically runtime tested for conformance with the constraints of a reference architecture. As an example of this technique, an executable application system is described, and it's executions are shown to violate the published X/Open DTP standard.

We expect the technology described in this dissertation to demonstrate to X/Open (and other standards organizations) that formal and poset–based specification of standards can contribute significantly to automated conformance testing. Checking conformance of products is of practical importance to companies like X/Open who promote "open" standards. Currently, X/Open uses test suites that are very costly in time and effort to produce and maintain. Improvements in the testing process based upon RAPIDE should reduce this cost and aid early detection of defects, assure quality, and minimising time–to–market. Furthermore, we are in process (collaboratively) with SRI International to extend the present X/Open DTP reference architecture with security properties and to demonstrate conformance testing of an actual system developed by SRI. Thus, the RAPIDE technology is poised revolutionize the development and the automated conformance checking of industry standards.

Acknowledgements

I want to thank my advisor, Professor David Luckham, for suggesting this topic, for encouraging me to pursue it through its various incarnations, and for providing both financial and intellectual support throughout my dissertation research. He has helped extensively in formulating and refining ideas and in guiding the overall direction of the research.

I am also grateful to my associate advisors, Professors Gio Widerhold and Arthur Keller, for their friendship, for helping me to become a Ph.D. student and a computer scientist, for their willingness to serve on my committee, and for their comments and advice. I will always be indebited to them for their guidance with their wisdom and common sense through my first two years at Stanford.

I am also grateful to my fellow members in the PAVG projects who have been very supportive. Special thanks to Doug Bryan, Walter Mann and James Vera for much technical help and for building the Rapide compiler and several peripheral tools. I also thank Suvan Gerlach for her assistance in dealing with "the system."

Years ago, my parents and my brothers taught me to love learning. I thank them for that lesson and for their love; for without either one, this thesis would never have been written.

I also want to thank my many friends, most notably Damian Rouson and Tim Pinkston, for making my time at Stanford enjoyable.

Finally, I would like to thank my fiance, Deborah Wilson, for patiently encouraging me in this work during the past several years and for impatiently urging me to finish. Her love, support, and understanding gave me the strength to write this thesis. I am in her debt for the many hours I spent working when we could have been together, and I look forward to spending the rest of my life discharging that debt.

This work was supported by DARPA under the Office of Naval Research (ONR) contract N00014-92-J-1928 and by AFOSR under Grant AFOSR91-0354. I was also supported in part by the IBM Graduate Fellowship program. The views and conclusions in this document are those of the author and should not be interpreted as representative of the official policies, either expressed or implied, of the ONR or the U.S. Government.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter provides first a brief, general overview of the properties and implementation protocols associated with transaction systems. Section 1.2 briefly describes several exemplary standards for distributed transaction processing (DTP) systems. Methods to deal with formally specifying the complexities of such systems are introduced in Section 1.3, and this section also describes the benefits obtained from a formal reference architecture. Section 1.4 summarizes the specific, original contributions of the dissertation. An overview of some of the work related to these contributions is given in Section 1.5, and Section 1.6 outlines and summarizes each major part of this dissertation.

1.1 Transaction Systems

1.1.1 Databases

Databases are partial models that provide a means to record facts about particular aspects of a domain. Implicitly, databases have state, and at every instant the database state is the collection of all the facts it contains. However, just as a model changes to remain current, so must a database change. Changes are introduced into a database by the execution of *transactions*. Transactions are programs that access and manipulate the data[1] in the database. As changes occur, the transaction system insures the preservation of the database's validity.

1.1.2 Consistency

Determining the validity of the database involves checking that the data satisfies constraints placed on the database. One class of constraints, called *integrity constraints*, restrict the static properties of database states and implicitly the dynamic properties of state transitions. The database is said to be *consistent* if it satisfies all its integrity constraints. Consider the example of a bank that has, as its data, variables representing the balances of the accounts and a variable representing the bank's total assets. It has an integrity constraint that the system must maintain the equivalence between the sum of all the balances of the accounts and the total assets. Whenever a user deposits or withdraws from an account, the transaction system must change both the account and the total assets variable to keep the database state consistent.

Consistency does not mean correctness, because eventhough a database is consistent it may not be correct. Transactions can make factual errors. For example, if a transfer transaction is incorrectly implemented as two deposits, rather than as one deposit and one withdrawal. As long as after the transaction executes, the total assests variable is equal to the sum of the accounts, the database will be consistent. However, this transaction (and therefore the database) is obviously incorrect.

1.1.3 Atomicity

Often the database must become temporarily inconsistent in order to transform it to a new consistent state. To cope with these temporary inconsistencies, sequences of primitive actions on the database's resources are grouped to form larger transactions. Since any transaction can be described as consisting of read and write operations of simple data objects, these operations are taken as primitives. These primitive

[1] The terms resource and data objects are commonly used to refer to the data in the database. The two terms are distinguished by assuming data objects are simple and have an interface limited to reading and writing, while resources have more complex interfaces and are assumed to be composed of several data objects.

operations are assumed to have behaviors that are not decomposable, i.e., *atomic*. Atomic operations are operations that are performed entirely or not at all; they cannot be only partially done at termination. In general, transactions are larger atomic operations on the database that transform it from one consistent state to a new consistent state.

The most common approach to achieving atomicity is the *two–phase commit protocol* [31, 32, 57]. The protocol divides transaction commitment into two phases. *Commitment* refers to whether the transaction can end successfully, i.e., can do what it was requested to do. In Phase 1, the polling phase, each resource is asked whether it can commit its part of the transaction, if it is requested to do so. If a resource determines that it can commit its work, it replies affirmatively. This positive response is a promise, guaranteeing that if asked to commit, it will. A negative reply reports failure for any reason. When all the resources have responded, Phase 2, the decision phase, is entered. If all resources responded affirmatively, then all of the resources are requested to commit their parts of the transaction creating a new consistent state. Otherwise the resources are all requested to *undo* their parts of the transaction thereby restoring the database to a previous, consistent state. Thus, the entire transaction is ensured of being either atomically committed or undone.

Since transactions are atomic operations, if a transaction system has a critical constituent primitive action that aborts or fails, making completion of the entire transaction impossible, then the entire transaction must be undone. Failures are related to accidents encountered by the transaction system during a particular attempt to fulfill a transaction request. Why should the transaction system allow a transaction that may fail to begin executing? That is because it is not always possible to a priori determine whether a transaction will fail. Similar to system failure, abortion is when the user wants the requested transaction to be undone. Of course, a user may request an abort of a transaction at any time, but the transaction system may not always be able to fulfill an abortion request. In particular, if a transaction has already ended, then the associated abortion request will be denied.

A general approach for undoing a transaction is to maintain a history of all changes made to the database and the status of each transaction. This history is called a *log*.

It contains records that encode the set of data objects updated by the transactions and their respective old and new values. Using the old and new values and a recovery scheme, the transaction system can undo a transaction. The undo procedure scans backwards from the end of the log until the beginning indicator for the transaction is found. During this scan each record denoting a write for the transaction is used to rewrite the old value back.

1.1.4 Isolation

Since even with abortion and failure transactions can maintain their atomicity, it seems natural to assume that transactions explicitly mark the boundaries between consistent database states. This assumption is often expressed as a property of transactions: that if a transaction is begun on a consistent database state it will be guaranteed to end with a state that is also consistent, i.e., transactions are *consistency preserving*. However, transactions are only guaranteed to transform the database between consistent database states when they do not overlap. Two transaction executions *overlap* when they occupy the same time interval or access the same resource. Therefore, the transaction system must also insure that transaction executions are non-overlapping. This property is called *isolation*. An isolated transaction has changes that are invisible to other transactions happening at the same time.

If *i*) all transactions are consistency preserving and are executed in isolation from other transactions, *ii*) initially the database state was consistent, and *iii*) the transaction system behaves "correctly," then every execution must result in a consistent database state. This definition of correctness is based upon the concept of a serial execution. In a *serial* execution, the transactions execute one at a time. This is one correct way to execute them, because each consistency preserving transaction will begin in a consistent database state and terminate with the database in a consistent state. However, serial executions provide very poor performance, because they do not allow transactions to concurrently share resources, and a single transaction rarely requires all the resources. Clearly, for performance reasons, a more general notion of correctness is needed.

If a transaction is executed concurrently with other transactions, then it is natural to assume that the concurrent execution is correct if and only if its effects are the same as that obtained by running the same transactions serially in some order. Since a serial execution is correct, any execution that is equivalent to a serial execution must also be correct. An execution that is equivalent to a serial execution is called a *serializable* execution.

Clearly, the usefulness of serializability is dependent on the definition of equivalence. Equivalence of transaction executions is commonly based upon conflict. *Conflict* refers to the ability of an action's effect on the resources to adversely affect another action's effect on the resources. If such interference is possible, those actions are said to be in conflict. Two actions are in conflict if they operate on the same data item and one of them is a write. An execution is *conflict serializable*, if all conflicting actions in the execution are executed in the same order as some serial execution of the same transactions.

Two general approaches have been used to achieve serializability: optimistic and pessimistic. Pessimistic approaches limit transaction executions to only consistent ones; transactions are prevented from executing unless they can be guaranteed a priori to never come into conflict with any other transaction currently being executed. This is called pessimistic, because it assumes maximum contention among concurrent transactions. For example, assume a transaction that will give a raise to the lowest paid employee. When executed under a pessimistic approach, the transaction would prevent any other transaction that would access the employees' salaries from executing, because any employee may be selected as the lowest paid one.

One pessimistic protocol is called *two–phase locking* [23]. *Locks* are a synchronization method in which transactions dynamically reserve data before accessing them. In the first phase of the two–phase locking protocol, a transaction only acquires (does not release) locks, and in the second phase the transaction only releases (does not acquire) locks. The protocol ensures serializability, because it insures that the order in which any two transactions access the same object (a possible source of conflict) is the same as the order in which those transactions access any other object.

Optimistic protocols [56] are not fundamentally based upon locking. These approaches are based upon the assumption that conflicts occur infrequently. They allow all transactions executions and then validate correctness at commitment. They work by allowing updates to occur on a private copy of the data and then checking (effectively at commitment) to see whether there are in fact any conflicts. If not, the updates are installed in the database; otherwise the transaction is undone.

1.1.5 Durability

It is not enough to set the transaction system to just any consistent database state whenever a transaction commits. The system must not erase the effects of committed transactions. Transactions are *durable*, if their committed changes are guaranteed to persist for other transactions. Durability in the ideal form is impossible to implement because of failures that lose information. Once such a loss is detected, the durable transaction system must attempt to restore the lost database state to the state that existed prior to the occurrence of the failure. If too much information was lost, it will be impossible to restore that previous state.

The process of restoring the state is called *recovery*. Recovery strategies and algorithms are based upon replicating the data. The approaches range from making an exact replica or duplicate copy of the data to using a log of all update operations. Duplicate copies are used, because it is assumed the copies will have independent failures from the primary copy and will provide a greater probability of having at least one usable copy of the data. However, creating copies is commonly done only when no transactions are being executed. Such quiescent times rarely occur naturally during a system's execution, and if the system creates one, the user delays are often unacceptable. Therefore, it is more common to maintain a log as the system executes. After a failure, the information stored in the log is used to re–execute the transactions, and thereby restore the database state.

Durability and the other transactional properties (atomicity, consistency and isolation) are more difficult to ensure when the transaction system is dispersed geographically around several sites. In particular, where control, knowledge (database schema), processing or state is spread (or *distributed*) among the sites, each site dependent on the others. As a consequence of dispersing the transaction processing engines, the distributed system must adhere to common communication protocols (or standards) to behave properly.

1.2 DTP Standards

Distributed transaction processing (DTP) is an important aspect of information systems, and it's importance will grow. As databases have become more commonplace and decentralized, transactions have begun to execute across multiple databases. This trend will continue, and the complexity and requirements on the interoperability between the transaction processing engines will increase.

To facilitate this trend, agreements on how transaction processing components communicate will be needed. These agreements take the form of software interoperation *standards*, i.e., models to which others must conform inorder to communicate. Such standards describe component interfaces, behaviors, communication protocols, and software architectures. For example, a typical DTP system is a collection of transaction processing engines, collections of data (databases) and application programs that logically belong to the same system but are spread over sites of a computer network. There are many DTP standards, and we describe only a few of the well established or emerging ones.

1.2.1 LU 6.2

Systems Network Architecture (SNA) is IBM's architecture for data communication between distributed systems. SNA's protocols have been enhanced via the "Logical Unit 6.2" (LU 6.2) protocols [38] to support transaction processing and other function applications. LU 6.2 includes facilities for transaction invocation, data transfer, two–phase commit, and recovery after transaction or network failure. It defines an application programming interface (or API) and underlying layers of services. All of the LU 6.2 protocols are "fully" specified, published, and tested. However, its specification is not in a form that is directly ameniable to automated testing nor reusable to developing standards.

1.2.2 ISO OSI-TP & OSI-CCR

The International Standards Organization (ISO) has defined several standard protocols for "Open Systems Interconnection (OSI)" that allows transaction systems to interoperate [42]. Two of these standards are significant to this dissertation. They are: Open System Interconnect — Transaction Processing (OSI-TP) [45] and Open Systems Interconnection — Commit Control and Recovery (OSI-CCR) [43]. These ISO–OSI standard protocols operate atop the OSI six–layer communications protocol stack, and they do not define any programming interfaces; they simply define message interfaces (or formats) to allow computers to interoperate using the two–phase commit protocol.

1.2.3 X/Open DTP

X/Open Company Limited is defining a standard [98, 99, 100, 101] that includes an application programming interface to allow applications to be portable. The X/Open DTP standard's purpose is to define a standard communication architecture through which multiple application programs may share resources while coordinating their work into transactions that may execute concurrently. It describes several components and their interfaces, a few instances of a system architecture, as well as several protocols — including the two–phase commit protocol. Overall, the X/Open DTP documents total over 400 pages. Even so, the standard attempts to ensure only the *atomicity* of transaction systems and does not (so far) address the other ACID properties.

1.2.4 POSIX

The IEEE POSIX Working Group (Institute of Electrical and Electronics Engineers, Portable Operating System Interface for Computing Environments) is producing standards for portable operating systems. POSIX refers collectively to a number of standards specifications, but at the time of this writing, the only a few specifications have been approved. Most notable is POSIX.1 [80] which has taken a number of UNIX [2] concepts and replaced them with abstractions. Enabling otherwise UNIX incompatible systems potentially "POSIX–compliant" (whatever that means). The standard does not redefine UNIX; it defines interfaces, not an implementation.

One POSIX group (POSIX.11 [81]) is working on a standard for transaction processing. This standard, like the POSIX.1 standard, may have some advantage as a consensus standard, but in general they are too skimpy and watered down. Also, because it is a consensus standard and vendors have different interests, it might take years before this POSIX extension is defined.

1.3 Formal Methods

Because of the complexity of DTP standards, especially their system architectures, there is a need for a formal computational methodology for the design and construction of DTP systems, including the architectural aspects of the systems.

1.3.1 Formal Methods

Formal methods are mathematically based techniques for describing system properties. They provide frameworks for specifying, developing, and verifying systems in a systematic, rather than ad hoc, manner. Formal methods can be used in all phases of a system's development. They assist in performing system design during decomposition and refinement, as well as recording design decisions and assumptions. To learn from the experiences of building a prototype system, developers can do a critical analysis of its functionality and performance after it has been built. Using a formal method will help the critical analysis in revealing unstated assumptions, inconsistencies, and unintentional incompleteness in the system.

[2] UNIX is a registered trademark in the United States and other countries of Santa Cruz Operations, Inc., licensed exclusively through X/Open Company Limited.

1.3.2 Formal Specification

A *specification* serves as a contract, a valuable piece of documentation, and a means of communication among a client, a specifier, and an implementor. Specifications are high level and declarative in nature; they describe the "what" rather than the "how." If specifications are formal, they also serve as an expression of the correctness criteria against which system implementations can be verified and validated.

Specifically for transaction systems, each application has specific axioms of its theory of integrity constraints. The *verification* of integrity constraints involves the consistency proof of the set of constraints. The *validation* of a transaction against a set of integrity constraints involves the proof that the transaction preserves the validity of the integrity constraints. Proofs of this nature ensure the correctness of the system design. And such provably correct formal specifications are what is needed to represent complex critical transaction system applications.

1.3.3 Reference Architecture

Formal methods are not just needed for the applications, but for the transaction processing engines as well, albeit for different reasons. The engine must ensure the atomicity, isolation, and durability properties of the transactions as it executes them. In general, ensuring such properties in a system is difficult. One approach to handling this difficulty is to emphasize architecture. Architectures should be represented formally in a machine processable, architecture description language.

Architecture, when used in this context, deals with the gross overall structure and composition of software components that are used to build a system. Each component has an interface that defines the ways the component can communicate with other components. A minimal architecture definition consists of a set of component interfaces and a set of connections between those interfaces.

The component interfaces and connections alone are not sufficient to describe a system; the behavior of the system is needed as well. The behavior can be described in two ways, either as a constraint or as executable code. Programmer can use executable code to develop a control flow abstraction of the system, and from this abstraction understand what the system is doing. Executable code can be simulated, and the programmer can observe what happened during that execution. Constraints assist the programmer in further understanding the system by describing properties of *all* conforming executions.

These requirements can be abstracted to define a *reference architecture*, a clear, precise, executable, and testable specification of how a standard should be represented. A reference architecture should contain formal, machine processable definitions of the component interfaces, their behaviors, and how the components may be connected together into architecture instancess, as well as formal constraints on those behaviors and on the communication protocols between the components. It can be interpreted as both a goal and a yardstick. As a goal, it is clear description of the desired behavior. As a yardstick, it must be precise and testable.

1.3.4 Rapide

This dissertation uses a new technology developed by the Program Analysis and Verification Group led by Prof. David Luckham, called Rapide [9, 67, 83, 84, 85, 86, 87, 88, 89], to define reference architectures. In Rapide, it is possible to specify a program as a collection of modules whose execution is triggered by specifiable conditions. The specifications of the triggering conditions may include event execution and state. Rapide uses *potential causality* and time between events, as well as the state associated with the events as the primary triggering conditions. We define the potential causality relation as the transitive closure over the following: *i*) every event a particular Rapide process generates potentially causes all subsequent events that process generates, *ii*) the event denoting the sending of a message potentially causes the event denoting the receiving of the same message, and *iii*) events triggered by a condition are potentially caused by the condition. From this definition, it is easy to see that potential causality is a partially ordered relation over events.

Rapide provides features for specifying and simulating both abstract system architectures and protocols. This dissertation takes these constructs and specifications and shows how to perform very powerful analyses with them. Partial orders of events are used to develop algorithms and methodology to analyze the interactions among transaction models, concurrency control mechanisms, recovery strategies, and abstract system architectures.

1.4 Summary of Results

The specific contributions documented in this dissertation include:

- a definition of the distributed transaction processing domain,
- a reference architecture for the X/Open DTP industry standard,
- an extension to the X/Open reference architecture to specify isolation, and
- an application of a methodology for testing applications for conformance to reference architectures developed by Gennart and Luckham [28].

1.4.1 Domain of DTP

This dissertation develops concepts and language for formally defining the domain of DTP. A *domain* is a set of concepts, predicates, components, protocols, and architectures. The domain of DTP presented includes a collection of basic and predefined types for formally specifying commitment protocols, concurrency control mechanisms, recovery strategies, and abstract system architectures.

An *architecture* specifies the types of components of the system, the number each type of component, and how the components interact so as to satisfy the requirements of the system. The particular style of architecture [68] used in this dissertation is the interface connection architecture. An *interface connection architecture* defines all communication between the components of the system using only the interfaces. Interfaces specify both *provided* and *required* features. Connections are defined between a required feature of one component and a provided feature of another. Thus, all of the connections between components in an architecture are defined as connections between interface features of those components. This is only possible because the architecture definition language used allows component interfaces to specify required features.

This dissertation also presents a methodology to analyze the architectures of the DTP domain based upon developing prototypes. Prototype models are built because the kinds of systems being modelled are not practical starting points for analysis or experimentation, usually because they are too expensive to build, or inaccessible when built, or simply too complex. The prototyping methodology used in this dissertation is an example of evolutionary prototyping. *Evolutionary prototyping* is the process of developing a prototype for a system gradually, satisfying some requirements before attempting to improve its capabilities so as to satisfy others. There are several reasons to do this. The most important is to understand the requirements themselves. Information gained from the prototype should help uncover inconsistencies, incompleteness and inaccuracies. If we simply try to build a program to satisfy everything a once, we will be in danger of getting one big mess. This is another reason for formalizing programs, because the programs are too complex to reason about in their natural, unified state.

After building a prototype that satisfies the requirements, it is natural to expect that the prototype's architecture should be a template for guiding the development of the system (or family of systems). As a template, an architecture defines constraints on a system. A system "has" that architecture if it conforms to those constraints (in a sense to be defined), and if it does, then it meets the given requirements. Conformance to architectural constraints can be checked at runtime, or, in some cases, decided by proof methods.

1.4.2 X/Open DTP Reference Architecture

The description of the industry standard contained in the X/Open documents is informal, consisting of English text together with component interfaces given in C. The important features of the standard are the interfaces and protocols (in English and state tables) by which the interface functions must be called in specified sequences. The calling sequences are described in terms of a single thread of control (or thread) and C function calls.

Many different systems with various applications and resources may satisfy this standard. Ones that do are easier to combine together, thus promoting "open" systems. The goal of open systems is to build compliant systems with composable parts from more than one software vendor at relatively low cost by "instantiating" an "open" shared design.

The X/Open DTP shared design is a software architecture that should be less concerned with the algorithms and data structures used *within* components than with the overall system structure. Structural issues include: gross organization and global control structure; assignment of functionality to system components; protocols for communication and synchronization; scaling and performance; and selection amoung design alternatives. The X/Open DTP architecture defines constraints on compliant system instances and is therefore a *standard*. A standard in this sense is a constraint on the structure and behavior of compliant system instances. Codification of such architectural standards can be critically important in assuring that the various components of a system are integrated correctly.

The reference architecture we developed for the X/Open standard is a formalization that is executable and testable. It formally defines the component interfaces, behaviors, and architectures. It also formally defines constraints and protocols on the executions of the architectures. The approach formalizes many of the documents informal ideas in terms of RAPIDE's computational model — partial orders of event sets (posets).

1.4.3 Isolation Extension

The X/Open DTP standard is still evolving and currently only defines a commitment protocol for ensuring atomicity, and even in this definition no guarantees are made in the presence of failures. In the case of isolation, only very brief descriptions are given that relate to only the basic forms, even though more relaxed forms of isolation are viewed as a significant advance for SQL2 [44]. We studied the effects of extending the standard with several constraints for isolation and with prototypical behaviors that implement the two–phase locking protocol that ensures isolation.

1.4.4 Conformance Testing

X/Open Company Limited is a consortium of vendors who are defining portability standards for the UNIX environment. One function they perform is the *branding* of compliant components. X/Open has introduced a trademark to identify products that conform to its specifications. Branding assures the user that the product works as outlined by the standard. When branding works (and X/Open hasn't yet got it working across the board), it ensures that information systems buyers can choose platform vendor, or even application vendor, without fear of interoperability problems.

At present, X/Open endorses 750 products that carry the X/Open brand via conformance testing. *Conformance testing* involves detecting differences between an application's execution and a standard's constraints. Generally, such testing is called *runtime consistency checking*. Runtime consistency checking is a technique for verifying whether an execution of a program satisfies a specification at run time. X/Open markets their testing environment and test suite to software vendors who perform the testing themselves. X/open receives the results from the vendor and uses the vendors' disclosures as the basis for branding.

Current approaches for runtime consistency checking[3] are limited and cannot detect all violations of distributed transaction processing, e.g., a response to a poll based upon some (not all) of its inputs. However, RAPIDE uses an execution model that allows the encoding of causal information so that such violations may be detected. The fundamental principle for conformance testing in RAPIDE is based upon *event pattern mappings*. Event pattern mappings enable several improvements for consistency checking including:

- fully automated testing of system behavior for conformance to reference architecture constraints,
- automated detection of constraint violations,
- causal history of violations for analysis of compliance errors, and

[3] We don't know very much about X/Open's specific "branding" techniques; this comment is based upon discussions with several commercial DTP system vendors.

- possibility of permanent self–testings of systems, e.g., for critical properties like security.

Thus, mappings specify relationships between architectures, and they permit an application's execution to be automatically runtime tested for conformance with the constraints of a reference architecture.

1.5 Related Work

This dissertation is related to a considerable amount of work done by others. Most of the software architectural work was directly influenced by and done in conjunction with other members of the PAVG group at Stanford. A few other groups are working on environments for designing and analyzing composable software systems, architecture description languages, and software architecture models with varying levels of formalization. Additionally, there has been much work on standardizing transaction processing. There are several general transaction processing books and models, as well as extensive work on formalizing the concurrency control strategies. Their work will be highlighted below.

1.5.1 PAVG

Many of the concepts embodied in RAPIDE [83, 84, 85, 86, 87, 88, 89] have evolved from prior work at Stanford in the Program Analysis and Verification Group (PAVG) led by Prof. David Luckham in association with Frank Belz from TRW. They have (separately and together) made pioneering advances in the area of language design and tool development for the purpose of applying formal methods and analysis techniques to software and system design. RAPIDE has been defined, evaluated and evolved by the Stanford/TRW ProtoTech and DSSA team, and is the result of a long series of experimental prototyping and executable architecture definition languages (CPL [9, 67], RAPIDE-0.2 [14], Micro-Rapide [65, 66], Interim RAPIDE-1, Graphical Rapide), which themselves drew heavily upon the prior work of the PAVG.

Anna [51, 59, 61, 69], developed by PAVG, is a language extension of Ada [94] to include facilities for formally specifying the intended behavior of Ada programs. Anna tools include a specification analyzer, a tool for the debugging of and reasoning about the implications of formal specifications; and a runtime annotation checker. The Anna tool suite has been distributed to around sixty sites, twenty in academia and forty in industry. There is an Anna User's Group mailing list with over fifty active members.

TSL [36, 58, 62, 70] (Task Sequencing Language), developed by PAVG, is an annotation and specification language for multi-tasking Ada programs. The principal constructs in TSL are aimed at making it easy to describe sequences and other patterns of events in a program that is executing on many processors simultaneously. TSL tools include a preprocessor that automatically instruments Ada code to generate events for important tasking actions, a TSL run-time system that checks specifications at run-time, and an interactive debugger.

VAL [5, 6], developed by PAVG, is an annotation language extension to VHDL [39] suitable for specification, especially specification of timing behavior. A VAL tool suite that implements run-time checking of VAL specifications has been implemented. This tool suite includes a full VHDL front-end, and uses any commercial VHDL simulator as a back-end. A version of VAL called VAL+ based on event patterns has also been designed. A VAL+ specification uses patterns of events to map a detailed simulation consisting of many events to a more abstract simulation consisting of fewer events. VAL+ has been applied hierarchically to map simulations consisting of tens of thousands of events into a few events [27]. A paper on VAL+ [28] won best paper award at the 1992 Design Automation Conference.

The Rapide technology effort has taken a group of approximately ten university researchers over five years to produce. My specific contributions to this effort include varying degrees of involvement in the design of the language and the implementation of the tool suite. I contributed heavily to the language design, in particular, the specification of the runtime semantics of how events are matched and to a lesser extent how component's interoperate [15]. However, I did not contribute very much to the language's type system [71] nor the architecture definition concepts [64, 68].

I also participated in the development of the RAPIDE 1.0 tool suite as the principal designer and implementor of the pattern matching parts of the compiler and runtime system [50]. I also interacted with the primary implementor of the rest of the compiler's backend in the design of the backend's interfaces. I also gave extensive feedback to other tool builders on their code in detailed code reviews.

I have also used the language and tool suite to develop examples and methodologies for maximizing the benefits of using the RAPIDE technology. The methodologies are presented in this dissertation and summarized in [63], and the results of my teaching examples will be released in [52].

1.5.2 Software Architecture and Composition

There have been other groups working to provide a scientific and engineering basis for designing, building, and analyzing composable software systems, as well as languages, tools, environments, and techniques to support such a basis. The Composable Software Systems group at CMU led by David Garlan, Daniel Jacson, Mary Shaw, and Jeannette Wing [26] is one example. Their innovative claim is that connectors are first–class entities, just as components are, in a system. They also have an approach for system consistency checking [4] and for conformance definition to an architectural style [2].

Other work contributing to the field of languages, tools and environments for architectural design include architecture description languages (ADLs). ADLs address the need to find expressive notations for representing architectural designs and styles. The focus of this research is to provide precise descriptions of the "glue" for combining components into larger systems. Three other ADLs are related to RAPIDE: LILEAnna [92], MetaH [95] and QAD [35].

Library Interconnect Language Extended with Annotated Ada (LILEAnna) refines and merges two preexisting languages, LIL [29] and Anna with mechanisms to specify abstraction, composition and reuse of Ada [94] packages. Architecture specification and construction are supported by two LILEAnna features called views and makes. A view allows users to specify how generic parameters, exported services and (for LILEAnna packages) imported services are bound to other LILEAnna theories, packages and the objects exported by them. Make allows a user to specify how Ada packages can be composed and instantiated to form other Ada packages, where views can be used to refine and control this process.

MetaH is an ADL focused on capturing the connectivity and behavioral information relevant to real–time scheduling, fault–tolerance, security, and scalable multiprocessing. It is not a complete environment on its own, but is intended to be used with other specialized tools, languages, and library facilities that specify comonent functionality, e.g., ControlH [48].

Given notations and models for characterizing software architectures, it becomes possible to support architectural design with new tools and environments. *Design patterns* [25] is one tool for architectural design. Design patterns is an abstraction mechanism to promote reuse of software architectures. It captures the static and dynamic structures and collaborations of components that arise when building software from a domain. Design patterns is more concerned with recording and reusing a software architectural design than with using the design as a reference.

It is clear that the format used by design patterns to encode a design is quite useful for capturing general purpose programming. However, it is not obvious how useful design patterns that rely on natural language rather than formal methods are at capturing the added complexity of design associated with concurrent or distributed programming. Design patterns represent a design primarily with only object diagrams and structured prose, and Schmidt [91] have found it necessary to augment a design pattern with source code, because the system's behavior was not sufficiently described. Also, his validation of patterns was done by human inspection rather than by automated testing as is done in RAPIDE.

There is a definite need to integrate architectural activities into broad methods and processes for software development, and to develop techniques for determining and predicting properties of architectures. Moriconi and Qian [74] considers the problem of architecture refinement. The authors argue that refinement should preserve certain structural and semantic properties, and show how this notion leads to the use of conservative extension as a refinement criterion. Similarly, Inverardi and Wolf [46] have developed an approach based on viewing software systems as chemicals whose reactions are controlled by explicitly stated rules. Both approaches appear to be more theoretical exercises than practical solutions.

Booch [12] also describes a method (or more accurately a notation) for capturing the architecture of software systems, and it also exemplifies this method with several case studies. This notation is able to capture the object–oriented, static as well as dynamic properties of system architectures. However, this notation is not formal nor ameniable to anything but the simplest machine–processable analysis.

In general, there are very few published case studies of architectural design including retrospective analyses of architectural development. Many of the previously described techniques include case studies, but the Domain Specific Software Architecture program sponsored by the U.S. Department of Defense [30] does not present a general language or framework, but ratherly solely concentrates on case studies on gathering architectural information. This is an excellent sourse of raw data from which approaches can be abstracted.

Compositional mechanisms on middleware are an emerging field of research. One example aimed at easing development of distributed runtime systems in a CORBA–based environment [33] is Aster [47]. However, this work does not have (and we are not aware of anyone else's work that has) the behavioral analysis or constraint–based tools that Rapide has [68].

1.5.3 Concurrent and Simulation Languages

VHDL [39] is an IEEE standard hardware description language based on discrete event simulation. Rapide generalizes many of the concepts that appear in VHDL. The VHDL computation model is a totally ordered stream of events, while the Rapide model is a partial order. VHDL processes trigger on sets of events, while Rapide processes trigger on more general patterns. VHDL architectures are static (fixed number of components and fixed interconnections), while Rapide architectures are dynamic.

Strand is a concurrent logic programming language. Rapide processes behave in a manner similar to concurrent rules in Strand. Rapide patterns are more general in that they operate on the partial order directly. Also, the communications architecture, implicit in Strand, is made explicit in Rapide.

LOTOS [11] is a specification language for distributed systems based on CSP [37]. LOTOS defines an interleaving semantics for concurrency, but recent work has begun on defining a partial order based semantics for LOTOS. However, the LOTOS language itself does not permit explicit use of the partial order the way patterns do in Rapide.

1.5.4 Formal Methods

The European ESPRIT projects have been very active in the areas of formal specification, concurrent systems specification, and formal verification (both methodology as embodied in the VDM [10, 20] project, and underlying proof rules and formal semantics as embodied in work based on CSP [37] and CCS [72]). A recent ESPRIT project, PROCOS [16], is aimed at applying these kinds of specifications and proof rules to build reliable concurrent systems. However, none of these languages and underlying analysis techniques utilize partial order models of concurrency.

1.5.5 Transaction Processing

There exists a large amount of literature on transaction processing and transaction processing systems. Some excellent, general books on database and transaction processing systems include: Date [18, 19] Gray [32], Gray and Reuter [31], Korth and Silverschatz [55], Papadimitriou [77], Ullman [93], and Wiederhold [96].

Formally specifying transaction processing properties and developing protocols and system architectures for them has had considerable interest in the literature. Babin [7] has developed a framework for transforming the requirements of transactions into a system that implements them. However, not only is it unclear how much of this process can be automated, but it also needs a formal language for specifying the process logic and tools for verifying the correctness of the specifications.

Chrysanthis and Ramamritham [17] introduce ACTA, a framework for specifying and reasoning about transaction structure and behavior. ACTA is not another transaction model but is intended to unify the existing models with a semantic model capable of specifying previously proposed transaction models. This framework is quite expressive but neither a rational, methodology nor benefits of using it have been presented.

Gabrielian [24] addresses the difficult problem of formal specification of real-time systems. A specification consists of a basic nondeterministic machine and of an ordered list of other machines which express constraints on the behavior of the future system. The authors' approach combines the description of nondeterministic state machines and temporal constraints in the same specification language.

Agrawal and Dewitt [3] takes a unified view of the problems associated with concurrency control and recovery for transaction–oriented multiuser centralized database management systems, and presents several integrated mechanisms.

1.6 Outline

This dissertation consists of six chapters and a set of supporting appendices. The aim of the chapters is to concisely describe the contributions of the dissertation. The appendices, on the other hand, provide useful supplemental information.

Chapter 2 describes the principles behind transaction systems. It defines the distributed transaction processing terminology to be used in the dissertation.

Chapter 3 describes an approach to define models for formally specifying and developing prototypes of distributed time–sensitive systems, including their architectural aspects. Many examples written in RAPIDE, and this chapter can be used as an overview of the RAPIDE technology.

Chapter 4 defines and formally specifies transaction processing system architectures, properties and protocols. Each property and corresponding exempliary implementation is specified via a RAPIDE reference architecture, complete with component interfaces, connections, constraints, and behaviors.

Chapter 5 applies the RAPIDE technology to specify and test conformance to a distributed transaction processing standard, the X/Open DTP reference model.

Chapter 6 contains our conclusions, including a review of the original contributions of this dissertation, as well as our ideas for continuing this research.

Appendix A is the Rapide reference architecture for the X/Open DTP industry standard.

Appendix B is the code for the example banking system.

Appendix C contains the maps used to perform the conformance testing.

Chapter 2

Principles of Transaction Systems

2.1 Databases

A database consists of individual data objects that are based upon the basic building blocks of the database's data model. The nature and size of the objects are defined by the data model. In general, each object is assumed to have a name and to be assignable with values from its domain. The collection of all such values determines the state of the database. Current systems use a relational data model where the objects manipulated are regular. They may be large, like relations, or small like individual tuples or even the attributes of tuples. Relational systems also restrict every data object in the database to have exactly one value at any time. However, this value changes as updates are made. This semantic model of data objects is called a *single–version* model, since only a single value (or *version*) exists at any time.

The single–version semantic model is contrasted with multi–version or temporal models that keep old data values for the data objects. *Multi–version* models retain old versions after a new version containing the new value of the object is created. One of the mechanisms required by systems using multiple versions is some method to determine which of the many possible versions a transaction should read.

Definition 2.1.1 *Let $\mathcal{E}$ denote the set of all data objects in the database, and $\forall e \in \mathcal{E}$, let $dom(e)$ denote the domain of object e. An unique state, s, is defined to be an interpretation (or a function with a domain E and range $\bigcup_{e \in \mathcal{E}} dom(e)$) that assigns* *to each object* $e \in \mathcal{E}$ *one value from the appropriate domain* $dom(e)$*, denoted by* $s[e]$. $\forall e \in \mathcal{E} : s[e] \in dom(e)$*. Let* S *represent the set of all unique states.*

2.2 Transactions

The state of the database is not static. It is continually undergoing changes due to *transactions* (or operations) performed on the system that modify the database state. Transactions are the units of work performed by a transaction processing system, and in the single–version semantic model they change the database from one unique state to another. Thus, in the single–version semantic model a transaction can be modelled as a function from unique states to unique states.

Definition 2.2.1 *A transaction,* t*, in the single–version semantic model is a mapping from* S *to* S*. The result of a transaction,* t*, applied to unique state* s *is another unique state denoted* $t(s)$*.*

A transaction is a computer program, where a distinction is made between the program code and the program's execution. A transaction implementation is the program code and describes what executions of the transaction are going to be like.

In general, a transaction may be made up of other transactions. As transactions are broken down into smaller and smaller pieces, the eventual result is a primitive action. *Primitive actions* are the simplest operations the TP system can perform. They form the basis from which all transactions will be composed.

Which actions should be chosen as primitive? Since all actions (operations) on objects are expressible (at the lowest level) in terms of reads and writes of data, those actions are selected to be primitive. They respectively return and modify the database state. Hence with no loss of generality, read and write actions are defined to be primitive.

Definition 2.2.2 *Let* $w(t, e)$ *and* $r(t, e)$ *denote respectively a write and a read primitive action by transaction* t *on object* e*. Also, let* $a(t, e)$ *denote either primitive action by transaction* t *on object* e*.*

A transaction implementation is structured such that it contains other nested transactions or primitive actions and a partial ordering on those "sub" actions. The partial ordering reflects the intention of the programmer. In the single–version semantic model, the ordering is a total order. This structure may be flattened out to an equivalent transaction that contains only primitive actions. The ordering on the primitive actions of the equivalent transaction is obtained by substituting the primitive actions for the subactions.

A transaction's execution is based upon an ordering and a function for each transaction. The ordering is over the set of subactions associated with the transaction. This ordering reflects the dependencies of the subactions that occurred in the execution. The result produced by the transaction is determined by the runtime execution order of the subactions. The only formal restriction placed on this ordering is that if two actions are related in the ordering associated with the transaction implementation, then the execution ordering cannot reverse that relationship. In other words, if a transaction designer defines subaction p to occur after subaction q, then it must be the case that all actions of p follow the corresponding actions in q.

The function assigns a virtual database state to the transaction. In a *virtual* database state, every object in the database is assigned a value, but the state may not actually exist at any time. This database state represents the values read by the transaction. If the transaction does not read an object, then any value from the object's domain written by another transaction can be used.

Definition 2.2.3 *A transaction t is a tuple (T, I, R, X), where T is a set of subactions, I is a partial order on T, $R \subseteq T \times T$ is a mathematical relation on T such that $(t_i, t_j) \in I^* \rightarrow (t_j, t_i) \notin R^*$, where I^* and R^* are the transitive closure of P and R respectively, and X is a mapping from T to a version state. If $t_k \in T$, then $X(t_k)$ is the input state of t_k.*

Either $(X(t_i))(e)$ is the original value assigned to e, or there exists some other transaction t_j such that $(X(t_i))(e) = (t_j(X(t_j)))(e)$, meaning some other transaction t_j has updated the value of e and t_i accessed the modified value. In the single–version semantic model, a transaction consists of only primitive actions, execute these actions sequentially (R is a total order), and allow only one version of each data object.

2.3 Isolation

A transaction processing system may execute more than one transaction, and isolation answers the question, "How should the correct execution of concurrent transactions be defined?" Since it is always possible that transactions will execute one at a time (serially), it is reasonable to assume that the normal, or intended, result of a transaction is the result obtained when it is executed with no other transactions executing concurrently. Thus, the concurrent execution of several transactions is assumed to be correct if and only if its effect is the same as that obtained by running the same transactions serially in some order.

Alternatively this principle can be viewed as a property of transactions. That is, a transaction shall not make updates to the database that affect the other transactions happening at the same time. This defines isolation; transactions are *isolated* only when they do not overlap other transactions, where the overlap is with respect to time or database state.

2.3.1 Serial Executions.

An execution is *serial* if each transaction runs to completion before the next one starts. Specifically, no action of a transaction can be found temporally between the first action and the last action of another transaction in the execution. This is the opposite of concurrent execution, where the transactions overlap in time; when the execution of transaction's actions overlaps with another transaction's actions. Thus, a serial execution is one in which all of the transactions do not overlap in time.

Definition 2.3.1 *An execution R is a serial execution, $Serial(R)$, of transaction T iff:* $\forall t_i, t_j \in T : i \neq j \rightarrow ([\forall a(i, e_i) \in t_i, a(j, e_j) \in t_j : (a(i, e_i), a(j, e_j)) \in R] \wedge [\forall a(i, e_i) \in t_i, a(j, e_j) \in t_j : (a(j, e_j), a(i, e_i)) \in R])$.

2.3.2 Serializable Executions.

Forcing the transactions to obey a serial execution will lead to poor performance, because serial executions do not take advantage of possible concurrency. To permit greater concurrency while preserving consistency, a transaction system's execution of a set of transactions may be serializable. An execution is *serializable* (SR) if it is "equivalent" to some serial execution of those same transactions. In other words it must be possible to order the actions of the transactions $t_1, t_2, \ldots, t_n$ such that t_1 "sees" the initial database, and t_2 "sees" the database that would have been produced if t_1 had run to completion, etc. The execution produced by this sequential view of the transactions' executions is called a *serialization* or *linearization* of the transactions.

Since serializability is based on equivalence to a serial execution, the question of defining serializability transforms into defining this equivalence. Three common definitions of equivalence can be found in the literature [77]. These are final-state equivalence, view equivalence and conflict equivalence. Each of these definitions describe progressively smaller classes of correct executions, but each smaller class has advantages which offset the reduction in potential concurrency.

Note, there may also be correct executions that are non–serializable. Definitions for non–serializable executions generally involve using additional information about the transaction model. These definitions are useful, becasuse they can be used to aviod hotspots that degrade performance.

2.3.3 Final–state Equivalence.

An execution is final–state equivalent to a serial execution if it leaves the database in a state which could be reached by a serial execution. Intuitively, two executions are final–state equivalent if they produce the same final state of the database for all interpretations of transactions and all initial database states. The problem with this equivalence is the database states generated during the execution is unimportant. Thus, transactions could see inconsistent database states, and therefore, the execution would, of course, violate our isolation constraint.

2.3.4 View Equivalence.

An execution is view equivalent to a serial execution if every transaction executes upon a database state that could be generated by a serial execution. Since serial executions are considered to correct, this ensures that every transaction sees a consistent database state. In order for an execution to be view serializable, one criterion is that if a transaction, say t_1, reads a value for a data item written by another transaction, t_2, then any third transaction, t_3, which writes that data item must either do so after the read step of t_1 or before the write step of t_2 in the execution. A common view inconsistency problem is the phantom tuple anomaly [23]. It arises when one transaction examines a group of tuples, and then before the first transaction ends, another transaction inserts a new tuple that would have belonged to the group.

2.3.5 Conflict Equivalence.

This definition is used commonly used as "the" definition of *serializability*, since it can be recognized in polynomial time, and there exist efficient algorithms to ensure that an execution is conflict serializable (CSR). Conflict equivalence is a proper subset of view equivalence, thus an execution that is CSR has had all transactions to view consistent databases.

Conflict equivalence constrains that every pair of conflicting operations between transactions should be ordered the same as in a serial execution. Conflict refers to the ability of a action's effect on the objects to adversely affect another action's effect on the objects. If such interference is possible, those actions are said to be in conflict.

The only way actions can interfere with each other is through reads and writes of shared data objects. The *read set* of a transaction is the set of objects the transaction reads, and the *write set* is the set of objects the transaction writes. Thus, two transactions *conflict* if the read set or write set of one intersects the write set of the other. If two transactions have one or more conflicts, all of the conflicts must be ordered the same as in a serial execution.

Two read primitive actions by two different transactions to the same object cannot violate consistency, because reads do not change the object state. Only write primitive actions may create violations. Two write primitive actions to an object by the same transaction do not violate consistency, because it is assumed that the transaction knows what it is doing to its data; it is assumed that if a transaction runs in isolation, it will correctly transform the database state. Consequently, only write–related interactions between two concurrent transactions can create inconsistency or violate isolation.

2.4 Consistency

The transaction processing system ensures that the data in the database is correct. Isolation defines the correctness during concurrent executions, and consistency defines correctness before and after executions. Consistency constrains the values of the data and ensures the database state satisfies certain properties called *consistency constraints*.

2.4.1 Consistency Predicate

Transaction consistency is based on the notion that it is possible to determine whether or not the database state is consistent. Any validity (or consistency) metric is assumed to be expressible as a predicate on the values of the data objects.

Definition 2.4.1 *Let P denote the validity predicate on unique states. A transaction t is defined to be consistency preserving, CP, if and only if it maps from a consistent state, $P(s)$, to a consistent state, $P(t(s))$, i.e., $t \in \mathcal{CP}$ iff $\forall s \in S : \mathcal{P}(s) \rightarrow \mathcal{P}(t(s))$.*

2.4.2 Consistency Preserving Transaction

A *consistency preserving* transaction is defined to be a transaction that upon executing in isolation on a initially consistent database, will terminate with the database in a state that also satisfies the database consistency constraint. When a set of consistency preserving transactions is serializably executed on an initially consistent database state, each transaction will read and produce a consistent database state and therefore the system will end in a consistent database state.

Theorem 2.4.2.1 *A property of serial executions, which is a requirement for transaction processing applications, is that each transaction executes upon a consistent database state. This property holds assuming i) the initial database state c_0 is consistent, ii) if a transaction is executed in isolation, it will preserve the database consistency, and iii) the system doesn't allow for the existence of arbitrary states after the executions of transactions.*

Proof: A consistent database state c_n will be obtained from the initial, consistent database state c_0 if the transactions are executed serially, since there will be an ordering of the transactions, $t_1, t_2, \ldots, t_n$, such that t_1 sees the initial state c_0, for each $i > 1$ transaction t_i sees the consistent state, c_i, that will be produced by running $t_1, t_2, \ldots, t_{i-1}$ sequentially to completion, and c_n is the state produced by t_n.

2.5 Atomicity

Transactions are not guaranteed to execute to completion, but instead are ensured to a weaker but still sufficient property: transactions are performed entirely or not at all; they cannot be only partially executed at termination. This all–or–nothing property is called *atomicity*.

This ideal definition is commonly refined to deal with the effect of a transaction on the database state. The execution of an atomic transaction will leave the system in either the state derived from all of the instructions or the initial state. Atomic transactions consist of instructions that when executed transform the system from one state to another. They may also be implemented by more than one instruction, and in some cases their execution may have to be stopped, and the state of the system restored. Upon being restored and if the transaction was isolated from every other transaction, then it will be as if the transaction never happened. This process of restoring states is called *undoing* the transaction.

During the execution of a transaction, the system may wish to abort the transaction or whoever invoked the transaction may decide to abort it. The invoker (or user) may wait to see the results before deciding whether to issue the abort or commit request, and the user can even first issue a commit request and later change his mind and issue an abort request. If the system receives an abort request and can abort the transaction, i.e., the transaction can be undone, then it undoes the transaction and issues an aborted result. If the system receives an abort request but cannot undo the transaction then the system will issue an not aborted result.

2.6 Durability

The effects of a committed transaction's execution must not be erased, but rather the system must guarantee them to be permanent for other transactions. This property is called durability; if a transaction is *durable*, then its changes will persist. However, durability in this ideal form is impossible to implement because of processor failure, transmission failure, disk crash or even catastrophic failures like natural disasters. In each of these cases, some of the information concerning the database is lost.

Failures are related to accidents encountered by the transaction system during a particular attempt to fulfill a transaction request. Why should the transaction system allow a transaction that may fail to begin executing? That is because it is not always possible to a priori determine whether a transaction will fail or not.

Once such a loss is detected, the system must restore the database state to the state that existed prior to the occurrence of the failure. Detecting the loss is not very difficult, and the primary concern of the database designers is the restoration. Performing this restoration is usually accomplished through the initiation of various backup and crash recovery procedures on the system storage.

Storage is traditionally viewed as a hierarchy. The lowest level of storage is *volatile* storage, like main and cache memory. This data is transient and easily lost due to system crash or power failure. Information residing in *nonvolatile* storage usually survives system crashes but may be lost. Typical nonvolatile storage devices are disks and magnetic tapes, which have failures like media failures or head crashes. *Stable* storage is the highest level of the storage hierarchy, and information residing in it is "never" lost. Theoretically, this cannot be guaranteed, because stable storage media are still susceptible to catastrophic failures like fires, earthquakes, floods, wars or acts of God.

2.7 Common Protocols

There are several common protocols for ensuring atomicity, isolation, and durability. Of course, they are not perfect solutions, and this dissertation will omit (for simplicity) discussion on their limitations and the various extensions to the protocols to improve their generality. Instead (for brevity) this dissertation will concentrate on the most common, general approaches.

2.7.1 Two–Phase Commit Protocol

Atomicity is ensured by coordinating managers of the database's resources to make a common decision about the commitment of the transaction. The most common commit protocol is the *two–phase commit protocol.* It is so called, because the protocol is divided into two phases. In the first phase, called the polling phase, each resource's manager is asked whether that resource can commit its part of the transaction, if it is requested to do so at some time in the future. This is a "prepare to commit" message. If a resource manager can commit its work, it prepares to do so and replies affirmatively. A negative reply reports failure for any reason. When all the resource managers have responded, phase 2, the decision phase, is entered. If all of the resource managers responded affirmatively, then the decision is made to commit, and all of the resource managers are subsequently requested to commit (otherwise they are requested to abort) their parts of the transaction. Thus, the entire transaction is assured of being atomic.

2.7.2 Two–Phase Locking Protocol

The *two–phase locking protocol* is used to implement isolation. The isolation theorems state that transactions can execute concurrently in isolation if the objects each transaction reads and writes are disjoint from those written by others or if the changes to shared resources that a transaction effects do not become visible outside the transaction until the transaction commits. The theorems indicates how locking can achieve this: lock everything you access and hold all locks until commit.

2.7.3 Write–Ahead Logging Protocol

The durability of transactions is assured using volatile, non–volatile and stable storage devices via recovery strategies and protocols. The difficulty in designing them is in the methodology of optimally using the storage hierarchy such that modifications to the database persist only if the associated transaction commits. Many approaches are based upon maintaining a log of all changes made by each transaction. One in particular, called the write–ahead logging protocol, requires that the changes to the database be recorded in the log before the resources. Upon detection of a failure, the transaction system uses the log to redo all of transactions, and thereby restore the database state.

Chapter 3

Principles of Formalization

This chapter presents concepts for formal modelling of distributed programs and uses Rapide [9, 67, 83, 84, 85, 86, 87, 88, 89] to exemplify the modelling concepts. A formal model of distributed programs must include how a distributed program is composed of executable parts called *components* and how the components are composed to define the system's behavior. A system's set of components and behavior is called the system's *architecture*.

The *behavior* of the architecture is dependent upon how its components behave individually, and how the components communicate to one another. The behavior of a component defines the relationship between the input the component reacts to and the output the component generates. Communication between the components must be made through *connections* that define the relationship between the output communication a component generates and the input communication the other components receive.

A behavior definition can be expressed in two ways: as a *constraint* and as an *executable* specification. Executable specifications are procedural processes that receive input and generate output communication. Constraint specifications (or constraints) are declarative conditions on the executable specifications. Constraint and executable specifications may be used, not only to define but also to analyze an architecture.

Comparative analysis of an architecture can be performed formally if the executable and constraint specifications are also formal. Such analysis is useful for several reasons:

- prove properties about the specifications, e.g., prove that the constraint specifications are satisfiable,
- construct an executable specification by manipulating the constraint specification,
- *verify* that an executable specification satisfies a constraint specification, either by mathematical argument, called *validation*, or by checking at runtime that the execution produced by the executable specification conforms to the constraint specifications, and
- check the conformance of a system architecture to a more general architecture specification.

Conformance checking of architectures is performed by interpreting a system architecture, called the domain architecture, as if it were another architecture, called the range architecture. The interpretation maps executions of the domain architecture onto the "universe" of the range architecture to produce a range execution. This kind of mapping is especially useful when the range execution can be checked for conformance to the constraints of the range architecture. In this way, the domain architectures' executions can be checked for conformance to the range architecture's constraints.

RAPIDE will be used to illustrate these formal modelling concepts. RAPIDE is an object–oriented executable architecture description language designed for specifying and prototyping distributed, time–sensitive systems. It separates a component's *interface* specification, e.g., the constituents by which the component communicates with other components, from the component's executable specification, called its *module*. A RAPIDE architecture consists of a collection of interfaces, a collection of connections between the interfaces, and a set of formal constraints that define legal and illegal behavior.

Figure 3.1 shows a system of modules connected together in an architecture. The shaded parallelograms represent interfaces, the thick lines represent connections and

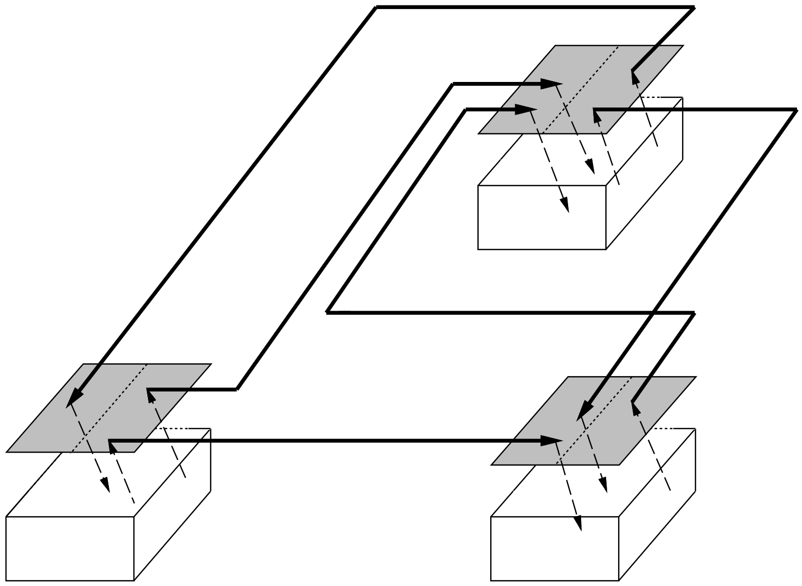

Figure 3.1: A RAPIDE architecture (top layer) and system (all).

the boxes represent modules. The architecture is shown as the set of interfaces and connections; the system is shown as the set of modules wired together by the architecture.

The reader may imagine an architecture executing by communication flowing up from the modules to their interfaces, along the architecture connections to other interfaces, and down into the receiving modules, and so on. Each module and connection can execute independently and concurrently. The communication at each interface must satisfy the constraints in the interface, and the communication in the connections must satisfy the architecture's constraints.

RAPIDE's execution model is based upon partially ordered sets of events (posets). The partial orders used are time and *potential causality* between events. Briefly, when a RAPIDE architecture executes it generates a poset. When an event is generated, it can produce a reaction in the connections and modules. Modules and connections will typically await the generation of some events and some condition of the program's state associated with those events, and then react to them. A module or connection reacts by executing some code that may include modifications to the program's state and the generation of additional events. Modules and connections often cycle through waiting, reacting, and generating events many times. The generated poset is checked for violations of the interface and architectural constraints.

The system in Figure 3.1 (i.e., the interfaces, connections, and modules) differs from a program written using Ada [94] packages or C++ [22] classes in that the modules communicate *only* if there are connections between their interfaces. In RAPIDE the architecture can be defined *before* the modules are written. It is a framework for composing modules, and it also constrains the communication between the modules of the system.

The system in Figure 3.1 can be written in RAPIDE to support different analysis methods. It can be executable to allow simulation and animation of the behavior of the architecture. It can also be a purely constraint–based definition of the allowable behavior of the system. Thirdly, the constraint–based definition can be used to check the architecture's behavior, as well as an actual implementation's behavior, for conformance to the constraints.

The rest of this chapter describes the underlying formalism of RAPIDE as well as the interface constituents that may be used to communicate with other components. The first section will address the formal computational model that is based upon event processing. Section 3.2 describes how to declare constituents of component interfaces. The basis for the architectural connections, executable and constraint specifications is the event pattern language described in Section 3.3. How to connect the component interfaces together into architectures is addressed in Section 3.4. The individual components (and therefore the architectures) are executable, and Section 3.5 describes the constructs used to associate executable specifications with component interfaces. Section 3.6 describes the constraint specifications that are compiled into runtime monitoring code. Finally, Section 3.7 addresses how to relate architectures to one another using event pattern mappings.

3.1 Event Processing

3.1.1 Event–based Semantics

Distributed program executions, especially the interactions between different components of programs, may be formally modelled with events or states. State–based approaches generally model program executions as sequences (or traces) of states, where the state of a program is defined as values given to the set of variables contained in the program. A trace of states represents the states of the program at successive points in time. Unfortunately, in state–based models, the interactions of the system are not obvious, and they have to be inferred from the changes in state.

In event–based approaches, such as RAPIDE, each event represents the occurrence of some activity within a program, e.g., an interaction between two components. Events are classified by *actions* and *functions* that define the kinds of activities that may occur during a distributed program's execution. Actions model asynchronous communication, and functions model synchronous communication between components. The execution of an action call will generate a single event, while a function's execution will generate two events, representing the call and return of the function. Thus, event–based models represent component interaction in a more natural and understandable manner than state–based models.

An event contains information such as the process and component generating the event, the name of the operation being invoked, and the data being passed.

The syntax for actions and functions is:

action_name_declaration ::= **action** *mode* *identifier* '(' [*formal_parameter_list*] ')' ';'

mode ::= **in** | **out**

function_name_declaration ::=
  function *identifier* '(' [*formal_parameter_list*] ')' [**return** *type_expression*] ';'

For example,

> **action in** Write(value : Data);
> **function** Read() **return** Data;

Commentary: This example illustrates action and function declarations. It declares a Write action that contains a formal parameter of type Data, denoted by value, and a Read function that contains no formal parameters. A Write event with a value of 5 is generated when a process makes a call to the declared action passing 5 as a parameter, e.g., Write(5). Note the use of the action *mode*, **in**, will be explained in the Section 3.2.

3.1.2 Concurrency versus Interleaving

Historically, an event–based execution has been modelled as a trace of events where the events in the execution are totally ordered. Executions that are modelled as traces of events are deficient in that they cannot truly represent a concurrent occurrence of events. Such models do not distinguish concurrency from interleaving of events. For example, if two events, say A and B, are concurrent then the set of traces $\{AB, BA\}$ (all the interleavings of A and B) represent the fact that A and B are concurrent. The same two traces also represent arbitrary non–deterministic interleaving of A and B. Thus, any system that must distinguish between concurrency and interleaving cannot be adequately modelled by trace models.

Pratt [82] proposed models that can distinguish between concurrency and interleaving. These models are called *event–based models with true concurrency*. Instead of traces, these models are based upon partially ordered sets of events (or *posets*). Time is one such partial order. In time–based models, activities that occur before other activities are given lesser *timestamps*, while later activities are given greater timestamps. Concurrency is represented by sharing the same timestamp value.

Temporal poset models have a characteristic that make time an unpractical partial order for accurately representing concurrency, because activities that are modelled with the same timestamp value may not have been concurrent. A single clock may produce timestamps that are not precise enough to distinguish concurrent from sequential activities. In this case, two events that are marked with the same timestamp value, may not have occurred concurrently. One may have represented an activity that preceded the other activity, but because the timestamp granularity was not fine enough, the two events were stamped with the same temporal value.

Additionally, distributed models may use more than one clock, each of whose timestamps are incomparable to the other clocks. In such situations, it is impossible to determine if events with incomparable timestamps actually occurred concurrently. Such a limitation usually leads the modeler to assume that incomparable timestamps implies concurrency.

Of course, a general solution to these two characteristics is to use one global, fine–grained clock. However, such a single, fine–grained clock model is not a practical starting point for representing distributed programs. Thus, a model of distributed programs that use only a temporal ordering, in general, is not an ideal choice, nor is it the only choices, to distinguish concurrency in distributed programs.

Another partial ordering used by Pratt to make the distinction between concurrency and interleaving is called *causality*. An event is caused by another if the first event could not have occurred without the occurrence of the second event. If there are two events that did not cause each other, they are said to be *independent*. Concurrency is defined to be causal independence in the poset, i.e., if two events occur independently in the causal partial order, they could have occurred concurrently in the execution.

Thus, we decided to use an event–based model that uses causality *and* time, since together these partial orders can naturally represent concurrency information between events that cannot be expressed by trace models. Additionally, the causal information is quite useful in analyzing program executions, and the causal dependency cannot be expressed in an obvious and understandable way using only traces, i.e., totally ordered sequences.

Definition 3.1.1 *A* partial order *on a set S is an irreflexive, anti–symmetric and transitive relation $\prec$ on the elements of S.*

Note this dissertation refers to the set on which the partial order relation is defined and the partial order relation, together, as a partial order. It will be clear from the context whether the relation or the set and the relation are meant.

A total order is a partial order that is also a total relation.

Definition 3.1.2 *A* total order *on a set S is a partial order $\prec$ on S such that if s_1 and s_2 are distinct elements in S then $s_1 \prec s_2$ or $s_2 \prec s_1$.*

3.1.3 Causality based upon Dependency

Causal relationships between events are represented in RAPIDE by the *dependency* partial order. An event B depends on an event A, written $(A \rightarrow B)$ if and only if:

1. A and B are generated by the same process, and A is generated before B. (Processes are sequential; all events generated by a process have a total dependency ordering), or
2. A process is triggered by A and then generates B, or
3. A process generates A and then assigns to a variable[1] v. Another process reads (in RAPIDE called dereferencing) v and then generates B. Note: this is a form of data dependency where an assignment to a ref object is analogous to a write of a protected variable, and the dereference is a read of the value written, or
4. $A \rightarrow C$ and $C \rightarrow B$, (i.e., transitivity).

Less formally, if an event B depends on an event A, then A must occur before B. Alternatively, A causes B.

RAPIDE programs may contain multiple clocks, and a RAPIDE execution uses a separate temporal ordering to express timing between events with respect to each clock in the program. A consequence of the rules of time and dependency is that there is a consistency relation between the dependency order and temporal orders: an

[1]The traditional programming language concept of "variable" is implemented in RAPIDE using objects of reference types which are defined in [86].

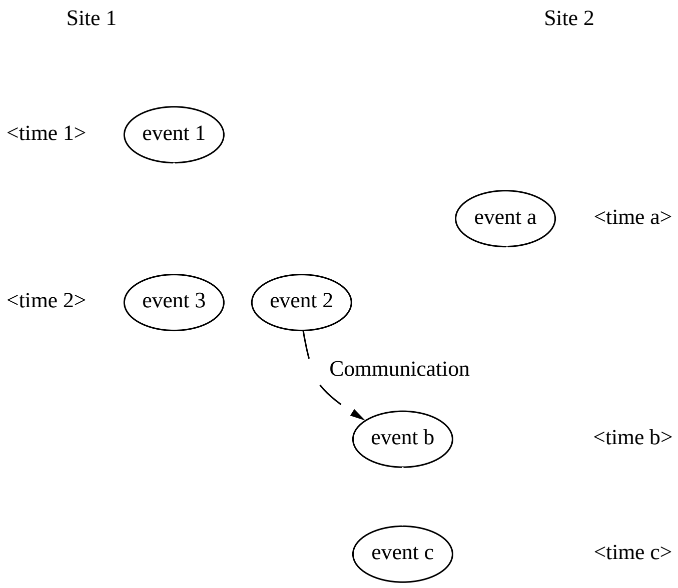

Figure 3.2: Timed poset

event cannot occur temporally earlier (by any clock) than an event that it depends on. Also, the various time orderings obey a consistency invariant: if event A temporally precedes event B for one clock, B cannot temporally precede A for another clock.

For example, please note the posets in Figs. 3.2, 3.3 and 3.4 that respectively show for the same execution a timed poset, dependency poset, and a timed, dependency poset (the union of the timed and dependency posets).

Commentary: The example graphs in Figs. 3.2, 3.3 and 3.4 illustrates that a timed, dependency poset is more expressive that either a timed poset or a dependency poset. In all of the graphs a labeled node represents the occurrence of an event at one of two sites; the nodes labeled with a number occur at site 1, and the nodes labeled with a letter occur at site 2. An arrow from one node to another represents dependency between the events, e.g., in Fig. 3.3 event 2 depends on event 1.

In the timed posets the timestamps of the site 1's (2's) events is denoted by a label to the left (right) of and parallel to the event's associated node. Thus, event 1 occurs at time 1, events 2 and 3 occur at time 2, event a at time a, event b at time

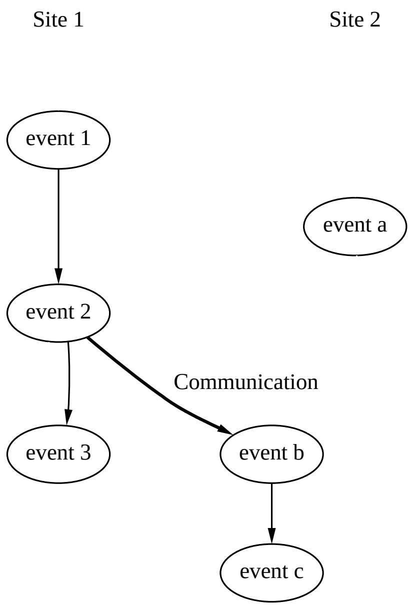

Figure 3.3: Dependency poset

b, and event c at time c.

The example models a distributed system where each site has its own clock, and time progresses in the figures top to bottom, e.g., site 1's event 1 is temporally earlier than site 1's events 2 and 3. Since events 2 and 3 are given the same timestamp, they are concurrent. Similarly, site 2's event a is earlier than site 2's event b that is again earlier than site 2's event c. Since the numbered timestamps and the alphabetic timestamps come from different clocks, they are incomparable.

In the timed poset, the dashed arrow emphasises the fact that the dependency occurring because site 1 (event 2) communicates with site 2 (event b), is not representable in a model that only has time.

In the dependency poset, the following can be deduced for the dependencies: event 1 causally precedes event 2 that precedes both events 3 and b, and event b precedes event c. Event a could have executed concurrently with all of the other events. In this poset, the thick arrow emphasises that the communication dependency *that was not representable in the timed poset* is represented in this model.

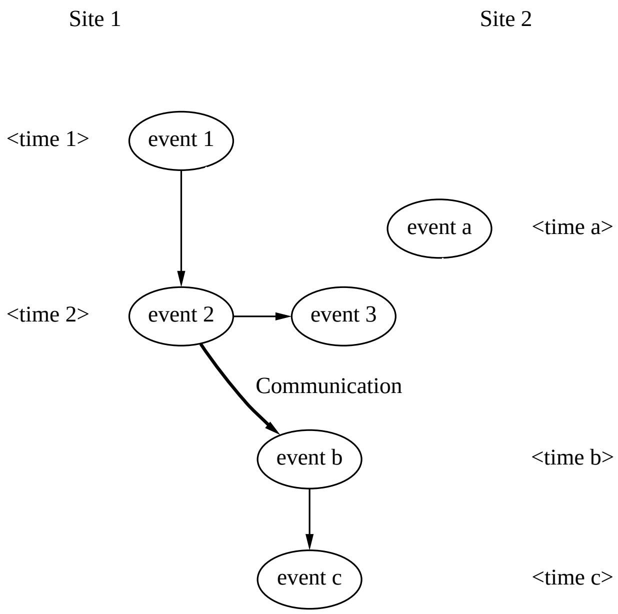

Figure 3.4: Timed, dependency poset

In the timed, dependency poset, all of the previous information could have been determined. However, we are still unable to determine whether event 1 and event a occurred concurrently.

3.2 Interface Types

The *interface type* of a component consists of the set of constituents by which the component communicates with other components. An interface type has seven kinds of declarative regions — (*i*) a *provides* part, (*ii*) a *requires* part, (*iii*) an *action* part, (*iv*) a *private* part, (*v*) a *service* part, (*vi*) a *constraint* part and (*vii*) a *behavior* part. Declarations may be names of types, names of modules or names of functions, and they associate a type with the identifier, but do not give an implementation.

Interface types are declared using the following syntax:

type_declaration ::= **type** *identifier* **is** *interface_expression* ';'

interface_type_expression ::=
  interface { *interface_constituent* }
    [**behavior** *behavior_declaration*]
  end [**interface**]

interface_constituent ::=
    provides { *interface_declarative_item* }
  | **requires** { *interface_declarative_item* }
  | **action** { *action_name_declaration* }
  | **private** { *interface_declarative_item* }
  | **service** { *service_declarative_item* }
  | **constraint** *pattern_constraint_list*

All modules of an interface type must contain types, modules, and functions matching the declarations in the *provides* part, and these types, modules, and functions are *visible*, which means that they may be referred to outside the module. The constituents of a *requires* part are names of components that are visible to the module but are not contained within it. Those *requires* components are assumed to be contained within another external module. The constituents of a *private* part are only visible and used within modules of the interface type.

The constituents of an *action* part may be used by any component in the system. Exactly how is dependent upon the *mode* of the action. *Out* actions declare the types of events the component may generate and thereby send to other components, while *in* actions are used by other components to send events of the action type into the component.

The *service*, *constraint* and *behavior* parts of interfaces are discussed later in this chapter. The *behavior* part of an interface type will be described in Section 3.5.4, the *constraint* part in Section 3.6, and the *service* part in Section 3.4.3.

For example,

```
type Data_Object is interface
   provides
      Read  : function() return Data;
      Write : function(d : Data);
end interface Data_Object;
```

Commentary: This example illustrates the use of *provides* constituents of an interface type. The interface type Data_Object contains two name declarations, Read and Write. Components of type Data_Object must contain actual objects (functions) with these names that other components may also use. Note that this interface has no *requires* or *private* parts.

3.2.1 Interface Type Constructors

Interface type constructors are templates for interface types. The templates contain slots into which type and object expressions may be placed to obtain a type expression. Type constructors are defined using *type constructor declarations*, and their "instantiation" to a type expression is called a type–constructor application.

For example,

```
type Data_Object_TC(type Data; init_val : ref(Data) is nil(Data)) is interface
   type Call_Type is enum Action_t, Function_t end enum;
   provides
      Read  : function() return Data;
      Write : function(d : Data);
   action
      in    Read_call();
      out   Read_retn(value : Data);
      in    Write_call(value : Data);
      out   Write_retn();
```

```
  private
    action
      Read(c : Call_Type; value : Data; version : Integer);
      Write(c : Call_Type; value : Data; version : Integer; initial : Boolean);
end interface Data_Object_TC;
```

Commentary: This example illustrates the use of type constructors. Data_Object_TC is a type constructor. It has one type parameter Data that can be replaced by any type during application. A second, optional parameter init_val may be replaced by any reference to an object of the Data type. Data_Object_TC contains eight name declarations. The *provides* functions, Read and Write, will be associated with bodies defined by components whose type is constructed by an application of Data_Object_TC. Similarly, these components, whose type is constructed by an application of Data_Object_TC, must also define *in* actions, Read_call and Write_call, and they may generate the *out* actions, Read_retn and Write_retn. The *private* actions, Read and Write, are defined by and may only be used by Data_Object_TC components.

Note the Read and Write functions are not associated with the similarly named actions (Read_call, Read_retn, Write_call and Write_retn) by the semantics of functions and actions in the RAPIDE language.

Data Object Model

Since the Data_Object_TC type constructor will be the subject of all of the rest of the examples given in this chapter, we will give a brief description of the model of data objects for which it was designed. Data objects are entities that store data values, e.g., memory or variables in computer programming languages. Without loss of generality data objects can be modelled with a basis of only two kinds of operations, reads and writes. A read operation returns the value of the object, while a write operation supplies a new value for the object. An object goes through a sequence of "versions" as it is written and read by the operations. We distinguish between *version* and value, because during an object's lifetime it may have the same value but never the same version. Reads do not change the object version, but each time an object is written

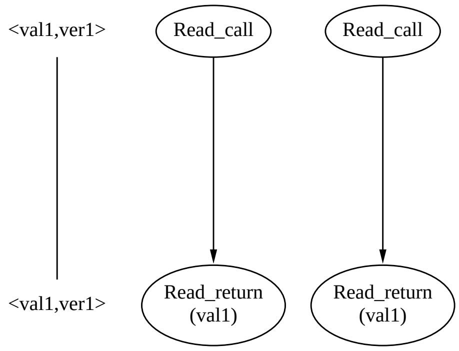

Figure 3.5: This example shows two reads of version $ver1$ on the object by two modules that use the same data object. The nodes in the graph are the events and the arcs are the dependencies. The evolution of the state is depicted on the left–hand side with the tuple "¡val1,ver1¿" indicating the value and version of the data object. At the top of the figure when the read calls occur, the state of the data object is the same as at the bottom of the figure when the read returns occur.

it gets a new, unique version. Thus, each object has a version generated at the time of the write operation as a part of it's value.

Read and write operations are represented in RAPIDE's poset–based execution model as pairs of events, a call and a return event. Thinking in terms of this model suggests a data flow graph, fragments of which are shown in Figures 3.5–3.8. The figures show the four possible executions of two operations operating on versions of an object. These executions exhibit dependencies between the read and write operations. Figure 3.5 depicts that there are no READ→READ dependencies. This is because operations reading the same version of an object create no depencency on one another. Only write operations create versions and dependencies. Another subtle point is the READ→WRITE dependency case, depicted in Figure 3.6 That dependency states that the Read read the object before the Write altered the object. Contrast Figure 3.6 with Figure 3.8 where the read reads after the write producing a WRITE→READ dependency.

In a single–version semantic model, every read returns the value last written. Multiversion models provide increased concurrency by allowing multiple versions of

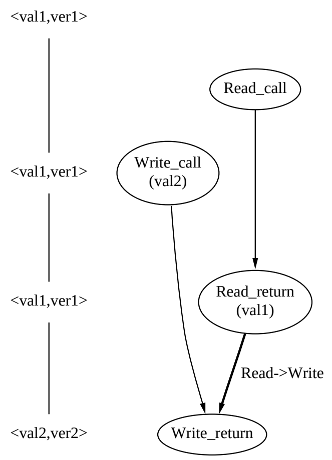

Figure 3.6: This example shows the READ→WRITE dependency; the write of version $ver2$ on the object occurs after version $ver1$ is read.

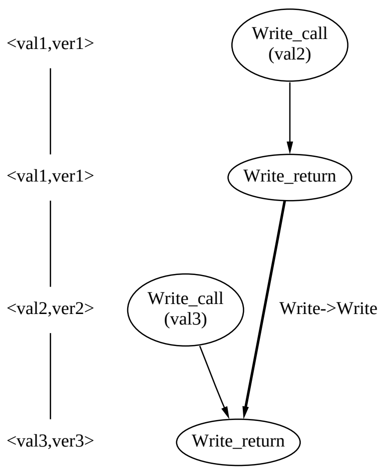

Figure 3.7: This example shows the WRITE→WRITE dependency; the write of version $ver3$ on the object after the write of version $ver2$.

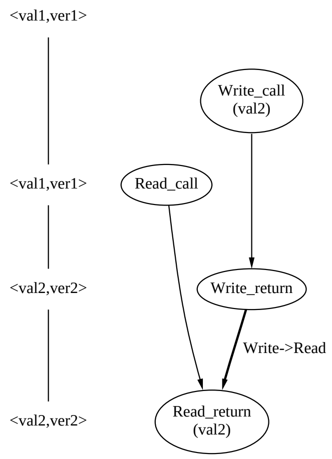

Figure 3.8: This example shows the WRITE→READ dependency; the read of version $ver2$ on the object occurs after version $ver2$ is written.

an object to coexist; old versions of an object can still be read after a new version is written. This is exemplified in Figure 3.8. The value returned by the read in the figure is $val2$ which represents the single–version semantcs. The multiversion semantics would allow the value returned to be either $val2$ or $val1$, the version before the write.

3.3 Pattern Language

When a Rapide program is executing, the program's computation consists of the timed, dependency poset and the state of the program's variables. Computations are specified using *pattern specifications* (or patterns) that are an extension of *regular expressions* [54] to specify posets instead of sequences of events. Each pattern defines a set of posets and state, called its *instances*. The process of recognizing whether a poset is an instance of a pattern is called *pattern matching*.

This section informally defines pattern specifications and gives examples of pattern specifications and operators. The formal definition of the pattern language may be found in [85].

Patterns may be constructed using the following syntax:

pattern ::=
    basic_pattern | '(' *pattern* ')'
  | **empty** | **any**
  | *pattern* *binary_pattern_operator* *pattern*
  | *pholder_decl_list* *pattern*
  | *pattern* '^' '(' *iterator_expression* *binary_pattern_operator* ')'
  | *pattern* **where** *boolean_expression*
  | *pattern_macro*

binary_pattern_operator ::= '$\rightarrow$' | '$\|$' | **or** | **and** | '$\sim$' | '$\equiv$'

iterator_expression ::= '$*$' | '$+$' | *expression*

3.3.1 Basic Patterns

A *basic pattern* is simply the name of an action, with optional parameter associations. A basic pattern specifies a set of posets, each of which is a single event labeled by the action with the given parameters.

For example,

Read_retn(o)

Commentary: This example illustrates the use of basic patterns. The pattern is only matched by Read_retn events whose first parameter is equal to the object "o."

3.3.2 Constants

There are two pattern constants, **empty** and **any**. The **empty** pattern is matched only by the *empty poset*, the poset that consists of zero events. The **any** pattern is matched by any single event; that is, by any poset that consists of one event.

3.3.3 Composite Patterns

Composite patterns are pattern expressions built from smaller patterns; while basic patterns are always matched by exactly one event, composite patterns may potentially be matched by any number of events. The binary pattern operation '$\rightarrow$' on patterns is called the *dependent* combinator, '$\|$' is called the *independent* combinator, '**or**' is called the *disjunction* combinator, '**and**' is called the *conjunction* combinator, '$\sim$' is called the *disjoint conjunction* combinator, and '$\equiv$' is called the *equivalent conjunction* combinator. Informal semantics for each combinator is given below.

- **Dependent:** P $\rightarrow$ P'. A match of patterns P and P' where all of the events that matched P' depend on all of the events that matched P.
- **Independent:** P $\|$ P'. A match of patterns P and P' where none of the events that matched P are dependent on any of the events that matched P' and vice versa.
- **Disjunction:** P **or** P'. A match of pattern P or a match of pattern P'.
- **Conjunction:** P **and** P'. A match of patterns P and P'.
- **Disjoint Conjunction:** P $\sim$ P'. A match of patterns P and P' where all of the events that matched P are distinct from the events that matched P'.
- **Equivalent Conjunction:** P $\equiv$ P'. A match of pattern P that is also a match for P'.

For example,

Read_call $\rightarrow$ Read_retn(o)

Commentary: This example illustrates the use of dependent composite patterns. The pattern is only matched by a poset consisting of two events, a Read_call and a Read_retn event whose first parameter is equal to the object "o." They are ordered in the poset, with the Read_call event causally preceding the Read_retn event.

3.3.4 Placeholders

Patterns may contain occurrences of *placeholders*. There are two kinds of placeholders: universal and existential. Placeholders beginning with "?" are called *existential placeholders* because of their similarity to the logical quantifier $\exists$. Existential placeholders represent "holes" in the patterns, where any value that is of the type the placeholder was defined may fit.

Placeholders beginning with "!" are called *universal placeholders* because of their similarity to the logical quantifier $\forall$. Universal placeholders are used in representing multiple instances of the pattern in which they occur, one instance for each object in the type of the universal placeholder. The operator in the universal placeholder declaration indicates the relationship the instances of the subpatterns have with each other. Placeholders may be declared in a pattern by giving a list of placeholder declarations, followed by a pattern.

The syntax of a placeholder declaration and placeholder declaration list is:

pholder_decl_list ::= '(' *pholder_decl* { ';' *pholder_decl* } ')'

pholder_decl ::=
    '?' *identifier* { ',' '?' *identifier* } **in** *expression*
  | '!' *identifier* **in** *expression* **by** *operator*

A poset C matches the pattern (?i **in** T) P(?i) if there exists an object o whose type is (or is a subtype of the type) defined by the expression T such that P(o) is matched by C. A poset C matches the pattern (!i **in** T **by** Op) P(!i) if, where L = [o_1, o_2, ...] is the list of objects whose type is (or is a subtype of the type) defined by the expression T and poset C matches P(o_1) Op P(o_2) Op .... Placeholder declarations may appear within patterns so the replacement can be restricted to subpatterns.

For example,

(!d **in** Integer **range** 1 .. 10 **by** $\rightarrow$) Write_call(!d)

Commentary: This example illustrates the use of universal placeholders. The universal placeholder !d is defined with respect to the binary pattern operator $\rightarrow$. This pattern is matched by a poset that contains ten totally ordered Write_call events, where each event's parameter, the universal placeholder !d, has a value in the integer range 1 .. 10, and there is exactly one event for each value in the range.

3.3.5 Iteration

An iteration pattern has the following syntax:
P^(*binary_pattern_operator iterator_expression*),
where *iterator_expression* is one of $*$, $+$, or n. Such a pattern is matched by *iterator_expression*, $*$ (zero or more), $+$ (one or more), and n (exactly n), matches of P, each match being related to the others by *binary_pattern_operator*.

For example,

```
Read_retn^(~ *)
```

Commentary: This example illustrates the use of iteration patterns. This pattern is matched by zero or more distinct Read_retn events. A poset consisting of three Read_retn events would actually contain eight matches for this pattern: one of size zero (the empty poset), three of size one, three of size two, and one of size three.

3.3.6 Pattern Macro

Pattern macros can be used to abstract and parameterize a pattern, and to define new operations on patterns.

The syntax for defining pattern macros is:

pattern_macro ::= **pattern** *identifier* '(' [*macro_parameter_list*] ')' **is** *pattern* ';'

macro_parameter_list ::= *macro_parameter* { ';' *macro_parameter* }

macro_parameter ::=
    macro_parameter
  | *type_parameter*
  | **pattern** *identifier_list*

Pattern macros serve several purposes. They may act as reusable patterns to be used in more than one location. They also simplify patterns by breaking them up into smaller units. Recursive macros allow the definition of patterns that are otherwise inexpressible. Finally, new pattern operators can be defined using macros (see Section 3.3.7).

For example,

pattern Ticks() **is** Tick → Ticks();

Commentary: Assume Tick is an action name. The pattern "Ticks" is an instance of the above macro. It describes an infinite chain of ordered Tick events. Pattern macros permit the description of infinite poset that are otherwise inexpressible. Iteration can only describe finite, though arbitrarily large poset.

3.3.7 Timing Operators

The *timing* operators are predefined macros that allow patterns to refer to the time at which events are generated. All timing operators can be defined in terms of a single basic timing operator, during, that is not otherwise expressible in the language. The pattern during(P, t1, t2, clk), where P is a pattern, t1 and t2 are of type Integer, and clk is of type Clock, is matched by a poset C such that (1) C matches P, and (2) all events in C are related to clk, and (3) t1 is the minimum (earliest) clk.Start value of, and t2 is the maximum (latest) clk.Finish value of any event in C.

The **during** pattern macro declaration is:

pattern during(**pattern** P; Time1, Time2 : Integer; C : Clock).

The other temporal pattern operators are defined as:

```
pattern at(pattern P; Time : Integer; C : Clock) is during(P,Time,Time,C);

pattern before(pattern P; Time : Integer; C : Clock) is
  (?First, ?Last in Integer) during(P,?First,?Last,C) where ?Last <= Time;

pattern after(pattern P; Time : Integer; C : Clock) is
  (?First, ?Last in Integer) during(P,?First,?Last,C) where ?First >= Time;

pattern within(pattern P; Time : Integer; C : Clock) is
  (?First, ?Last in Integer) during(P,?First,?Last,C) where (?Last−?First <= Time);

pattern within(pattern P; Time1, Time2 : Integer; C : Clock) is
  (?First, ?Last in Integer) during(P,?First,?Last,C)
      where Time1 <= ?Last−?First and ?Last−?First <= Time2;

pattern "<"(pattern P1, P2; C : Clock) is
  (?First, ?Last, ?First2, ?Last2 in Integer)
  during(P1,?First,?Last,C) ~ during(P2,?First2,?Last2,C) where ?Last1 < ?First2;
```

For convenience, an infix shorthand is defined for the timing macros. Any timing macro of the form op(P,T1,T2,C) may be written as P op(T1,T2) **by** C, and op(P,T,C) may be written as P op T **by** C. If the clock C is omitted, the default clock is used.

For example,

```
Read_call < Read_retn
```

Commentary: This example illustrates the use of the less than pattern operator that is defined by a pattern macro. The pattern is only matched by a poset consisting of two events, a Read_call and a Read_retn event. They are ordered in the poset, with the Read_call event temporally preceding the Read_retn event.

3.3.8 Guarded Patterns

Context may be determined by the use of *guards*. A match of pattern P **where** guard occurs when the poset matches P and the boolean expression guard is true. References to state in guard refer to their values when the last event participating in the match of P is generated.

For example,

```
Read_call where state_value = 0
```

Commentary: This example illustrates the use of a guarded pattern. The pattern is only matched by Read_call events that are generated when the value of state_value is equal to zero.

3.4 Architectures

Architectures declare *component interfaces* and *connections* between *requires* and *provides* constituents of component interfaces. Architectures may also declare *formal constraints* that define legal and illegal patterns of communication among the component interfaces. Components are modules that can generate and receive events as well as call and execute functions. As a result of a connection, *(i)* events generated by one component *cause* events to be received at another component, or *(ii)* functions called by one component are executed by another component. In this way, architectures define dataflow and synchronization between components using only their interfaces.

An architecture is a template for a family of systems. As an analogy, architectures can be viewed as printed circuit boards in which interfaces play the role of sockets into which component chips can be plugged, and connections play the role of wires between the sockets. Components communicate *only* if there are connections between their interfaces.[2] Thus, RAPIDE architectures are communication networks, defined independently of actual implementations.

[2]This notion of communication integrity is not enforced by the semantics of RAPIDE but rather via a particular style of writing RAPIDE programs.

An architecture is declared in RAPIDE using the following syntax:

architecture_declaration ::=
  architecture *identifier* '(' [*parameter_list*] ')' [**return** *interface_expression*] **is**
    [*module_constituent_list*]
  [**connect** { *connection* }]
  end [**architecture**] [*identifier*] ';'

connection ::=
  pattern connector pattern ';'
  | *other kinds of pattern connections ...*

connector ::= '**to**' | '=>' | '||>'

3.4.1 Components

Components are *active* modules, active in the sense that they can receive and generate events, make function calls, execute functions, as well as contain other active and passive modules. As a stylistic rule but not a language restriction, neither active modules nor references to active modules can be passed as a parameter of an action or function call. This restriction ensures communication integrity. Active modules only communicate through their interfaces.

Components can themselves be architectures. This provides a simple capability to develop a hierarchically structured architecture rather than one flat communication structure. When an interface in an architecture is associated with a module or another architecture, the resulting architecture is called an *instance* of the original one. The second (sub)module's interface or (sub)architecture's *return interface expression* must conform to (be a subtype of) the component interface it is being associated with. If omitted, the return interface expression of an architecture is the empty interface, called Root. Figure 3.1 depicted a fully instantiated architecture in which each interface has been associated with a module.

3.4.2 Connections

Whenever an event is generated and its associated action is a constituent of a component interface's *provides* part, that event will be tested to see if it matches any of the architecture's connection rules. If the event matches a connection's left–hand side (a pattern), the connection will *trigger*, causing the events on the right–hand side of the connection to be generated at another component interface's *requires* part. Similarly for functions, if a function is called and the function was declared as a constituent of a component interface's *requires* part, that function call will be compared will the architecture's connection rules. If the function call matches a connection's left–hand side, the connection will trigger, causing the function on the right–hand side to be called. The returned value from the right–hand side's call will be passed back to the original left–hand side's call as its return value. A component's *requires* function calls will be aliased to some other component's *provides* functions. Thus, connections define a flow of events and remote function calls between components. A crucial point is that any connection only depends upon the interfaces of component modules of a system, and not upon the actual modules that might be implementing those interfaces.

A connection can also relate events generated at the interface of the architecture to events at its components' interfaces, and conversely – thus defining dataflow into and out of the architecture.

If an architecture contains more than one connection, they will all trigger and execute concurrently.

Basic Pattern Connections

The simplest kind of connection relates two basic patterns. When an event matches the left–hand side basic pattern, the right–hand side will be executed. In this case the right–hand side is also a basic pattern. The execution of the right–hand side consists of generating the event matching to the right–hand side basic pattern. If this pattern does not contain parameters of the action, then the parameters of the triggering event are taken as default parameters of the action call expression.

Depending on the kind of connector used, the relationship between the events generated may vary. If the connector is *basic* (denoted by **to**) the generated event is causally and temporally equivalent to the event that matched the left–hand side. The *pipe* connection (denoted by =>) will result in the generated event causally following the triggering event, and causally following all events generated by previous executions of the connection. An *agent* connection (denoted by ||>) will also result in the generated event causally following the triggering event, but the connection doesn't causally relate the generated event to the events generated by previous executions of the connection.

For example,

connect AP.Read_call **to** Obj.Read_call;

Commentary: This example illustrates the use of a basic connection. The left–hand side of the connection is a basic pattern that is only matched by Read_call events generated by the AP component. When such a match triggers the connection, a Read_call event will be generated and made available to the Obj component.

Function Connections

A *requires* function in one interface may be connected to a *provides* function in another interface.

For example,

connect AP.Write **to** Obj.Write;

Commentary: This example illustrates the use of function connections. The functions are declared in the interfaces of AP and Obj. The functions must be type-compatible [49]. This connection has the effect that whenever AP calls its *requires* function Write, the call is executed as a call to Obj's *provides* function with the same arguments; the return values are returned to AP as the result if its call.

Patterns in Connections

An architecture is not restricted to a static hardware paradigm. The use of patterns in connections provides a powerful feature for specifying both static and dynamic architectures. A static architecture has a fixed number of components and the conditions under which they communicate do not vary — the printed circuit board example. A dynamic architecture may contain varying numbers of components and the conditions under which they communication can also vary — a distributed transaction processing system is an example of a dynamic architecture, since it may have varying numbers of resources and communication conditions. Often, communication between sets of components in a system can be specified using a single RAPIDE connection.

For example:

```
connect (?D in Data) AP.Write_call(?D) to Obj.Write_call(?D);
```

Commentary: This example illustrates the use of patterns in connections. A *requires* action of the AP is connected to a *provides* action of the Obj. The syntax (?D **in** Data) declares the type of objects (Data) that may be bound to the placeholder ?D. In general, a pattern may be prefixed by a list of these placeholder declarations. The connection defines event flow between the AP and Obj; whenever the AP generates a Write_call event then the Obj will receive a causally and temporally equivalent Write_call event with the same data.

For example:

```
connect
    (?A in AP, ?D in Data) ?A.Write_call(?D)
        to (!O in Data_Object_TC ||) !O.Request_Receive(?D);
```

Commentary: The connection connects every AP to every Data_Object_TC. The connection triggers whenever any application program generates a Write_call event and results in generating Write_call events with the same Data (?D) at every module that is a subtype of Data_Object_TC. Each data object's Request_Receive event is dependent upon the application program's Write_call event, but is independent of any of the other data object's Request_Receive events.

3.4.3 Service Connection

Often related constituents in an interface can be grouped into disjoint sets. For example, a transaction manager in the X/Open DTP industry standard has a set of constituents for interacting with applications (the TX set, see Section A.2.1) and another set for interacting with resource managers (the XA set, see Section A.2.2). Interface constraints relate members within a set, but the two sets are almost entirely independent. It is convenient to structure such interfaces into separate sets of constituents so that it is clear how interface constraints apply. But more importantly, this structuring of the interfaces can be used to define large numbers of connections correctly in large architectures.

Services

Complex interfaces can be structured into related sets of constituents called *services*. Services provide the ability to encapsulate a related set of constituents of the enclosing interface and to allow large numbers of connections between components to be defined by a single connection rule.

The syntax for services is:

service_declaration_item ::= *basic_name_list* ':' [**dual**] *interface_type_expression* ';'

Consider,

type Resource **is** **interface**
  service DO : Data_Object_TC(Integer);
end **interface** Resource;

The DO service in the Resource interface denotes all of the actions, functions and nested services (if any) constructed from applying Integer to the Data_Object_TC type constructor. To name them, the name "DO" is appended with the usual "." notation before the name of the constituent. If the service were **dual**, the same constituents would be denoted except that the *modes* of the service's constituents are reversed; *provides* (*requires*) functions become *requires* (*provides*) functions, and in (*out*) actions become out (*in*) actions.

Service Connection

An architecture can connect together dual services in component interfaces. Such a connection denotes sets of basic connections, one for each constituent in the service. A service and a dual service of the same type may be connected together since they have complimentary *provides* and *requires* constituents.

For example,

```
type Application is interface
   service DO : dual Data_Object_TC(Integer);
end interface Application;

architecture DTP_Architecture() is
   A : Application; RM : Resource;
connect
   A.DO to RM.DO;
end architecture DTP_Architecture;
```

Commentary: Here the dual DO services of an Application and a Resource are connected by a single rule. This connection denotes the following set of basic connections between each pair of constituents with the same name in the two services:

```
A.DO.Read to RM.DO.Read;
A.DO.Write to RM.DO.Write;
P.DO.Read_call() to RM.DO.Read_call();
(?i : Integer) RM.DO.Read_retn(?i) to A.DO.Read_retn(?i);
(?i : Integer) A.DO.Write_call(?i) to RM.DO.Write_call(?i);
RM.DO.Write_retn() to A.DO.Write_retn();
```

Service Set Connection

A service set declaration declares a set of services of the enclosing interface, each of which has the type of the interface expression (or its dual). Each service of the set can be named by indexing the service set name with a literal of the range or enumeration type used as the service set index.

The syntax for a service set is:

service_set_declaration ::=
  basic_name_list '(' { *service_set_index* } ')' ':' [**dual**] *interface_type_expression* ';'

Service sets are connected to other services using connection generators.

The syntax for connection generators is:

connection_generator ::=
  generation_scheme **generate**
    connection
  end [**generate**] [(**if** | **for**)] ';'

generation_scheme ::=
    if *expression*
  | **for** *expression* **in** *expression* **next** *expression*
  | **for** *identifier* ':' *type_expression* **in** *expression*

For example,

type Application_Program(NumRMs : Integer) **is interface**
  service Rsrcs(1 .. NumRMs) : **dual** Data_Object_TC(Integer);
end interface Application_Program;

architecture DTP_Architecture(NumRMs : Integer) **is**
  AP : Application_Program(NumRMs);
  Rs : **array**[Integer] **of** Resource;

```
connect
  for i : Integer in 1 .. NumRMs generate
    AP.Rsrcs(i) to Rs[i].AP;
  end generate;
end architecture DTP_Architecture;
```

Commentary: This example illustrates the expressive power of service sets and service set connections. The application program interface type constructor has a set of NumRMs dual data object services for communication with the resources labeled Rsrcs, and Rsrcs(i) is the ith data object service in the set. The architecture connects the application program AP to the array of resources Rs with a connection generator.

3.5 Modules

Modules are defined by a set of processes (possibly empty) that observe events and react to them by executing arbitrary code that in turn may generate new events. Modules are a general construct for encapsulating the implementation of a component. Consequently, modules can be either values of a "small" passive type such as Integer or values of a "large" active type such as a multi–threaded subsystem.

A module must define the *provides* constituents of its interface and may define additional constituents, called *internal* constituents, not found in its interface. These internal constituents may themselves be other modules (or children).

Rapide defines the conformance rules (by means of interface types and constraints) that determine which modules may be associated with which interfaces.

3.5.1 Process

A *process* is an independently executing single thread of control. Processes are constructed from sequential statements, including action calls that generate events and reactive statements that react to events. Since processes are single–threaded, each event a process generates is dependent on the preceding event generated by the process. Thus, all events generated by a process form a total dependency ordering.

The syntax of processes is:

process ::= *statement_list* | *generate_statement*

For example,

for I : Integer **in** 1 .. NumObjs **do** DO(I).Write_call(0); **end** **do**;

Commentary: This process consists of a single for statement. It makes Write_calls to several DO service set members, and then terminates. The Write_calls are all causally related in a total order.

Event Generation, Availability and Observation

The rules for event communication are given in the Executable Language Reference Manual [84] which describes the process of event generation, availability and observation in detail. In summary, at run–time RAPIDE processes *generate* events through the execution of action and function calls. When an event is generated, it is immediately made *available* to other processes or connections. Constituents whose events are available to a process of a particular module are as follows: the in, *private* and internal actions of the module, and the out actions of constituent modules of the module. Events become unavailable to a process after they have participated in the triggering of that process, see Section 3.5.2.

Once an event becomes available to a process, it will subsequently be *observed* by the receiving process. A consistency relation holds between the observation and dependency orders, called the orderly observation principle: events are observed by a process one by one in some total order that is consistent with the dependency partial order.

Timing Clause Statements

The generation of an event may be delayed for some time period after the execution of an action call statement via *timing clauses*. As stated before in Section 3.1.2, time is modelled in RAPIDE as a partial order; activities that occur before other activities are given lesser timestamps, while later activities are given greater timestamps, and (possibly) concurrent activities are given the same timestamp. The purpose of a timing clause is to model activities that have duration or occur some time in the future. Therefore, an event, which models a particular activity, has two timestamps: its start and finish times. A timing clause modifies the generation of an event by determining the start and finish timestamps.

A RAPIDE clock is a monotonically increasing counter. The rate at which it increases is not related to any physical unit, but rather is controlled by the program execution. This kind of clock provides what is oftern referred to as *simulation* time, as opposed to real–time. A clock's counter value is a timestamp, and an increase in time is referred to as a tick. A clock ticks when there are no more events to be generated with the current timestamp. A single RAPIDE architecture may contain multiple clocks, and an event may be timed with respect to one or more clocks.

The syntax for timing clauses is:

timing_clause ::=
    action_call **pause** *timing_expression*
  | *action_call* **delay** *timing_expression*
  | *action_call* **after** *timing_expression*

timing_expression ::= *expression* | *type_expression*

A timing clause may only be applied to action calls, and it must include an expression. The timing expression may denote a single object or type. When an object is given, the object must be of type C.Ticks (Integer), and if a type is used, then it must be a subtype of C.Ticks, where C is called the named clock of the timing clause.

Timing clauses that use **pause** t will slow the execution of the action call for the t ticks of the clock. That is, if the action call begins with the named clock's current counter value is n, then the call will complete when the counter value is $n+t$. The event will be generated and made available at timestamp $n + t$, and the start timestamp of the event will be n, while the finish timestamp will be $n + t$. Thus, the **pause** timing clause can be used to model the occurrences of activities that have duration.

A **delay**ed action call is equivalent to the same call, substituting **delay** for **pause**, with the following exception. The action call will generate an event e with start timestamp n and finish timestamp $n + t$, and any event that is or becomes available to the process at timestamp s relative to the named clock, where $n \leq s \leq n + t$, will never be observed by the process executing the timing clause.

Timing clauses that use **after** t schedule the generation of the event for t ticks in the future. The start and finish timestamp of the event will both be $n + t$.

If any of the timing clauses is parameterized with a value that is less than or equal to zero, it is equivalent to the same call without a timing clause. If a type expression is given, then it must be a subtype of C.Ticks for some clock C. Such a timing clause is specifies an arbitrary value in the range of the type expression. That is an arbitray value in the range is selected, and if the range is empty, the predefined exception Timing_Error is raised.

For example,

Write_retn() **after** C.Ticks **range** 1 . . 3;

Commentary: This example statement schedules the generation of a Write_retn event 1, 2 or 3 ticks (non–deterministically chosen) in the future.

Note: Timed statements are very similar to timing clauses and enable a module to allow time to pass without generating an event. The effect of executing a timed statement is the same as executing an action call with an identical timing clause, except that no event is generated.

The syntax for timed statements is:

timed_statement ::=
  pause *timing_expression* ';'
| **delay** *timing_expression* ';'

Reactive Statements

Reactive statements are used to respond to events observed by a process.

The syntax of the reactive statements is:

await_stmt ::=
  await
    pattern ['=>' { *stmt* }
    pattern_choices
  end [**await**] ';'

pattern_choices ::= { **or** *pattern_choice* }

pattern_choice ::= *pattern* '=>' { *stmt* }

when_stmt ::=
  when *pattern* **do** *stmt_list* **end when** ';'

The fundamental reactive statement is the *await* statement; the *when* statement is a commonly used form of the await statement that has a special syntax. An await statement includes one or more patterns that may be matched by observed events; when a match of a pattern is observed, a (possibly null) body of statements associated with that pattern, called its *alternative*, is executed. If more than one pattern is matched by the observed events, an arbitrary choice is made among the them. The match of the pattern chosen is said to *trigger* the execution of its alternative. The scope of placeholders in an await statement's pattern extends to its alternative. One can think of an await statement alternative as a parameterized list of statements, such that upon observing a triggering match, an instance of the alternative is execution, where the placeholder occurrences in the alternative have been replaced by the substituting values determined in matching.

For example,

```
await
    Read_call =>
        Read(Action_t, $val, $ver) ≡ Read_retn($val);
or  (?v in Data) Read_retn^(~ *) ~ Write_call(?v) =>
        val := ?v;
        ver := $ver + 1;
        Write(Action_t, $val, $ver, False) ≡ Write_retn();
end await;
```

Commentary: This example consists of one await statement. In brief, the above statement waits for either a single, available and observed Read_call event or any number of available and observed Read_retn events and a Write_call event. If the Read_call event triggers, then a Read and a Read_retn event will be generated. The values of the parameters will be determined by dereferencing the (assumed) global val and ver reference objects. If the Read_retn and Write_call events trigger, then the two state assignments will be executed and a Write and a Write_retn event will be generated. The value assigned to val in the first assignment is equal to the value given the Write_call event that was bound in matching the pattern.

When statements are a special syntactic form of await statements; they model "rules" that fire repeatedly upon observation of a triggering match. The following *when* statement

when *pattern* **do** *stmt_list* **end when** ';'

is equivalent to:

loop do await *pattern* => *stmt_list* **end when** ';' **end loop**;

For example,

when (?v **in** Data) Read_retn^(~ ∗) ~ Write_call(?v) **do**
  val := ?v;
  ver := $ver + 1;
  Write(Action_t,$val,$ver,False) ≡ Write_retn();
end when;

Commentary: This example consists of one *when* statement. It repeatedly waits for any number of available and observed Read_retn and a Write_call event, performs two state assignments, and generates a Write and a Write_retn event in response to them.

3.5.2 Triggering

When a process needs to match a pattern[3] before it can continue processing, the process will look through its *pool* of available, observed events to find such a match. Upon finding such a match, the process is said to be *triggered*, and the events that participated in the triggering are then unavailable to that process. If no match is found, the process is blocked until one is found.

If the triggering process finds more than one match in the pool, three relations on the matches are used to select one:

Maximal: Given two matches A and B, match A is *maximal* if and only if match B is contained in match A and there are elements in match A that are not contained in match B.

Earlier: Intuitively, given two matches A and B, match A is *earlier* if there is a *consistent cut* through the execution poset where match A occurs above the cut, but match B does not. The term *earliest* refers to the transitive of earlier.

First: Given two matches A and B, match A is *first* if and only if any event in match A was observed before every event in match B.

[3]The pattern being matched is either a particular pattern or one of a group of patterns.

For modules (see Section 3.5.3), the *earliest, maximal* match (with that priority) among the pool will be used, and for behaviors (see Section 3.5), the *first, earliest, maximal* is used. If given all these rules, there more than one match, an arbitrary choice is made.

Definition 3.5.1 *Given a set S and a partial order $<$ on S, a* consistent cut *of $<$ is a partial order $<'$ on S' such that $S' \subseteq S$, $<' \subseteq <$, and $e' \in S' \wedge e \in S \wedge e < e' \rightarrow e \in S'$.*

3.5.3 Module Constructor

Modules are generally created using a module generator. A module generator declares a function that returns a unique module every time it is called. Each call to the generator will generate a new module.

The syntax for module generators is:

module_generator ::=
   module *identifier* ‘(’ [*parameter_list*] ‘)’ [**return** *interface_expression*] **is**
      [*module_declaration_list*]
      [**constraint** *module_pattern_constraint_list*]
      [**connect** *module_connection_list*]
      [**initial** *module_statement_list*]
      [(**parallel** | **serial**) *process* ‘||’ process]
      [**final** *module_statement_list*]
   end [**module**] [*identifier*] ‘;’

For example,

```
module Simple(type Data; init : ref(Data) is nil(Data))
 return Data_Object(Data,init) is
  val  :  var Data;
  ver  :  var Integer := 0;
  Read :  function() return Data is
            begin
              Read(Function_t,$val,$ver);
              return $val;
            end function Read;
  Write : function(value : Data) is
            begin
              val := value;
              ver := $ver + 1;
              Write(Function_t,$val,$ver,False);
            end function Write;
 connect
  (?v in Data) Read(Action_t,?v) to Read_retn(?v);
  (?v in Data, ?u in Integer) Write(Action_t,?v,?u,False) to Write_retn();
 initial
  if not ( init.Is_Nil() ) then
      val := $init;
      Write(Action_t,$init,$ver,True);
  end if;
 parallel
  when Read_call do Read(Action_t,$val,$ver); end when;
  ||
  when (?v in Data) Read_retn^(~ *) ~ Write_call(?v) do
      val := ?v;
      ver := $ver + 1;
      Write(Action_t,$val,$ver,False);
  end when;
 end module Simple;
```

Commentary: This example illustrates a module generator for the Data_Object_TC interface type declared in Section 3.2.1. Simple defines two local variables, val and ver, that are not visible outside of this module generator. It also defines two functions that were declared *provides* in its interface. Functions can generate events as well as return values, and in this case, the functions generate events that are *private* — because the associated actions were defined *private* in the interface.
Simple defines two connections. The first connection will generate a Read_retn event (via an out action call) every time a *private* Read event with an action_t Call_type is generated. Similarly, the other connection generates Write_retn events when Write events are generated.

When a module is initialized just after being created by an application of the module generator Simple, the module will test whether it has been given an initial value (init). If it has (init will not be Nil), the initial value will be store in the local variable val and a *private* Write event signifying this occurrence will be generated.

During execution the module may receive Read_call and Write_call events. Upon observing a Read_call event, the module will generate a *private* Read event, and upon observing a Write_call event, the module will update its local variables representing its value and version and generate a *private* Write event. its state will be updated appropriately

A possible execution of a module generated from an application of Simple is given in Figure 3.9. This execution features the execution of an application that initially issues a Read_call event. This requests were observed by the data object after the initial part of the object was executed and generated the Write(function_t,init_val,0,True) event. The data object's processing of the Read_call event causes the generation of a Read and a Read_retn event. Upon receiving the Read_retn event, the example application generates a Write_call and two more Read_call events. The data object observes and processes the events in the following order: *i*) one of the Read_call events, *ii*) the Write_call event, and *iii*) the other Read_call event. Every data object execution will produce the characteristic write, followed by a set of reads, followed by another write, as the object goes through its sequence of versions.

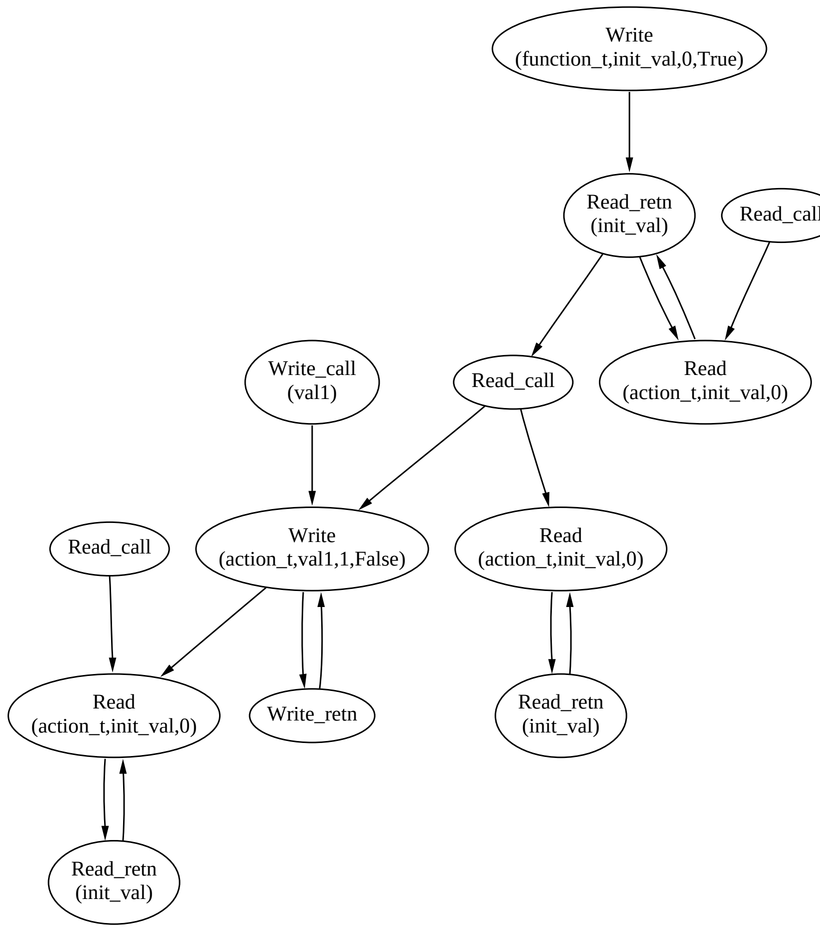

Figure 3.9: An execution of a module constructed from Simple.

3.5.4 Behavior

An interface type may contain a *behavior* part that is an abstract definition of the behavior of conforming modules to the type. If a module is not supplied for a component of an interface type, a default module will be generated from this behavior part. The behavior part of an interface consists of a set of types, objects, and transition rules. The objects model abstract state, and the transition rules specify how modules of the interface type react to patterns of observed events by changing their states and generating events. State transition rules are simple, finite state machine–like constructs that are more declarative than the procedural module generators.

The syntax for module generators is:

behavior ::= [*declaration_list*] **begin** { *state_transition_rule* }

state_transition_rule ::= [*pholder_decl_list*] *pattern* *op* *body* ';'

op ::= '=>' | '||>'

body ::= [{ *state_assignment* }] [*poset_generator* ';']

poset_generator ::= *restricted_pattern*

A state transition rule consists of an optional list of placeholder declarations followed by a *pattern*, an op symbol, and a *body*. The pattern found on the left side of the op symbol is called the state transition rule's *trigger*. A transition rule defined using the => op symbol is called a *pipe*, and one defined using the ||> op symbol is called an *agent*.

A body consists of an optional set of statements followed by an optional restricted pattern that describes a set of posets (called a poset generator). The scope of the outermost placeholder declarations in the trigger extends throughout the body.

The default module is generated from an interface behavior via a fairly straight–forward translation with only minor semantic changes. Each state transition rule is translated into a repeating process that awaits the generation of a match for the trigger, and executes the body. As a generated module's processes execute, they are synchronized such that only one process may trigger and execute at a time. To determine which process to trigger, all of the matches for all of the pattern triggers are collected from the processes pools of available, observed events, and the first, earliest, maximal match (with that priority) is chosen (see Section 3.5.2). Once an event participates in the triggering of a process (state transition rule), it becomes unavailable to that process, but will remain available to all of the others. Thus, an event may take part in triggering a given rule at most once, although it may participate in triggering several rules.

An action call statement in the body of a transition rule may include the *after* timing clause, but not the other timing clauses. A further restriction is that the pattern in the body may not contain the ≡ operator. These restrictions

For example,

```
behavior
  val   :  var Data;
  ver   :  var Integer := 0;
  Read  :  function() return Data is
              begin
                 Read(Function_t,$val,$ver);
                 return $val;
              end function Read;
  Write :  function(value : Data) is
              begin
                 val := value;
                 ver := $ver + 1;
                 Write(Function_t,$val,$ver,False);
              end function Write;
begin
  Start =>
        if not ( init.Is_Nil() ) then
           val := $init;
           Write(Action_t,$init,$ver,True);
        end if;;

  Read_call ||>
        Read(Action_t,$val,$ver) ≡ Read_retn($val);;

  (?v in Data) Read_retn^(~ *) ~ Write_call(?v) =>
        val := ?v;
        ver := $ver + 1;
        Write(Action_t,$val,$ver,False) ≡ Write_retn();;
```

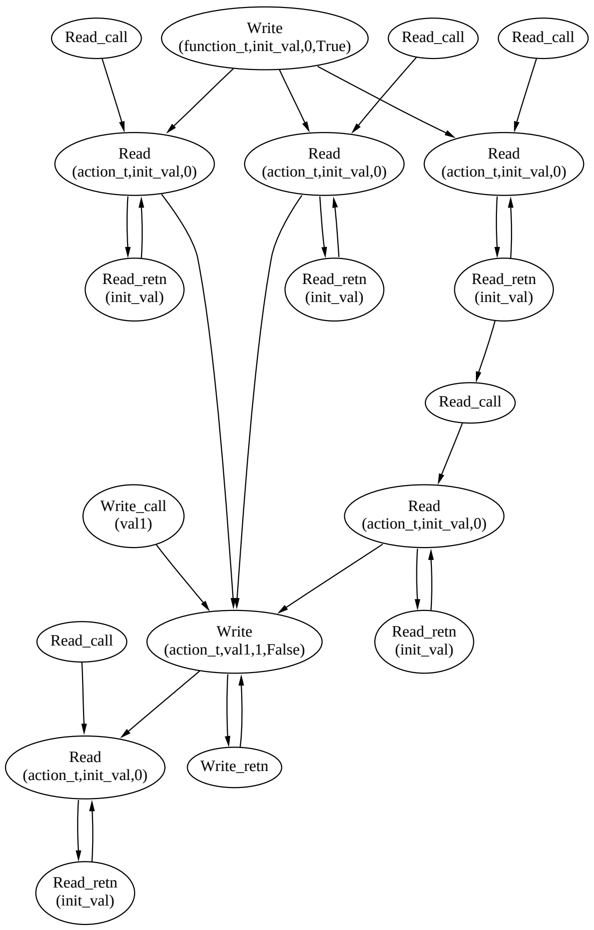

Figure 3.10: An execution of a module generated from the behavior.

3.6 Constraints

A good constraint language is a powerful specification weapon, particularly when supported by useable and efficient tools. In developing distributed systems to meet given requirements, a first step is a capability to specify intended behavior. Constraints may involve very simple assertions about the input and output values of a function or complex results of scenarios of activity between communication, but independently executing, systems of objects. Typically the latter might be communication protocols that are part of the specification of reliability, security, or time–critical requirements. A constraint language must be able to express these various kinds of specifications. In the case of distributed systems, constraints on causal event histories provide a powerful new kind of specification language. We argue, in fact, that this kind of specification language is necessary for expressing detailed requirements on distributed object systems.

Prior experience with the Rapide constraint language has shown that constraint checking tools are effective and necessary in analyzing complex simulations for properties that are often too complex for the human viewer. Constraints may be utilized either to specify the behavior of single components, in which case they are part of interface definitions, or they can specify communication between objects, in which case they apply to architecture connections.

The behavior of Rapide program may be constrained by *pattern constraints* that are based upon pattern specifications. Pattern constraints denote the patterns of events and state that are acceptable and unacceptable. This dissertation only uses the simplest kind of Rapide pattern constraints, the **never** constraint that disallows any observation of the given pattern. Other more complex constraints are described in the Rapide Constraint Language Reference Manual [87].

Every pattern constraint is associated with a Rapide module and constrains only those events that were made available to it. For a pattern constraint occurring in an interface definition, each module of the interface must satisfy that constraint. For a pattern constraint occurring in a module generator, each module created by a call to the generator must satisfy that constraint.

3.6.1 Never Constraints

A **never** constraint defines a pattern that should never occur.

The syntax of a **never** constraint is:

pattern_constraint ::= [*label*] **never** *pattern* ';'

No match of the given pattern should occur in the visible poset. If any such match exists, the constraint is violated.

3.6.2 Match Constraints

match constraints are used to specify event patterns that should occur during the execution of a program.

The syntax of a **match** constraint is:

pattern_constraint ::= [*label*] **match** *pattern* ';'

A **match** constraint constrains a subset of the visible computation to match the given pattern. The computation constrained is all of the events associated with basic patterns occuring in the given pattern. This subcomputation must be an exact match of the pattern for the constraint to be satisfied.

3.6.3 Data Object Model Constraints

Several example constraints from the interface declared in Section 3.2.1 and implemented in Sections 3.5.4 and 3.5.3 include:

Initialization. The example below illustrates the **never** constraint. The constraint requires that two Write events whose initial parameters are true may not exist in an execution. This constraint expresses that a data object can only be initialized once.

never Write(initial **is** True) ~ Write(initial **is** True);

One–To–One Correspondence of Private and Out Events. There should be an equivalence between *private* and out events.

```
match ((?val in Data) Read(?val) → Read_retn(?val))^(~ *);
match (Write(initial is False) → Write_retn)^(~ *);
```

Version Semantics. Read returns are not necessarily ordered, while write returns are totally ordered. This is because writes must create a new version that must be causally after the preceding version, while there is no such constraint on reads. Note: versions are generated in a sequence.

```
match Write_retn^(→ *);
```

A key property is that version numbers increase monotonically. Any two write returns must be causally ordered such that the version number of the dependent write return (?v2) must be greater than the version number of the preceding write return (?v1).

```
never (?v1, ?v2 in Integer) Write(version is ?v1) → Write(version is ?v2)
    where ?v1 >= ?v2;
```

If initially each object starts with a version number of 1 and each write increments the version number by 1, then at the leading edge of any consistent cut in the computation, the object's version number will be equal to the number of writes of that object before that edge.

```
match Write(version is 1, initial is True) → Write^(→ *);
match (?v in Integer) Write^(→ ?v) ≡ (Write^(→ *) → Write(version is ?v));
```

Single Version Semantics. Reads must return the last value written, and the versions must occur in sequences where a write occurs and is followed by any number of read returns that are followed by another version.

match ((?d **in** Data)(?v **in** Integer) Write(?d,?v) $\rightarrow$ Read(?d,?v)^($\sim$ $*$))^($\rightarrow$ $*$);

3.7 Event Pattern Mappings

Event pattern mappings (or maps) provide a powerful mechanism for defining relationships among architectures. The main purpose for maps is to define how executions of one architecture may be interpreted as executions of another architecture. In many cases, there is quite a wide ranges of differences among how a system can be viewed architecturally. For example, when two architectures of a system are at different levels of abstraction, many events in one may correspond to just one event in the other (as is often the case in hierarchical design). Patterns provide the necessary expressive power to define hierarchical as well as non–hierarchical kinds of mappings.

More generally, a map can interpret the executions of several architectures taken together (called *domains*) as executions of another *range* architecture. A significant benefit from interpreting domain executions as range executions is that the range execution generated by a map must satisfy the constraints of the range. Moreover, maps are composable; maps can be domain parameters in the definition of other maps. Thus, maps can be used as constraints on their domains and as domain parameters of other maps.

3.7.1 Map Generators

Maps are created by calling map generators.

The syntax of a map generator is:

map_generator ::=
  map *identifier* '(' [*formal_parameter_list*] ')' **from** *domain_list* **to** *range* **is**
    { *declaration* }
  [**constraint** { *constraint* }]
  rule { *agent_state_transition_rule* }
  end [**map**] [*name*] ';'

agent_state_transition_rule ::= *pattern* '||>' { *state_assignment* } [*poset_generator*] ';'

The actual parameters of a map generator invocation must correspond to the formal parameters of the map generator. These parameters serve the same purpose as formal parameters of function calls, to pass data values to the map generator. The data values passed are intended to be non–active modules, i.e., modules that neither receive nor generate events.

While formal parameters are used to pass non–active modules, the list of domain indicators is used to pass active modules or maps. If the domain indicator is a type expression, then the actual domain must be a module (or map) whose interface type (or range type) must be a subtype of the domain indicator's type. If the domain indicator is a module generator name, then the actual domain must be a module generated from that module generator or a map whose range indicator is the same module generator.

Since there are no real range constituents in a map, there is not an explicit range constituents' state. The map must provide the state by defining of constituents with the same name of the same type inside the map. If the constraints need values from a range constituent's state, the map will return the corresponding state from within the map.

Declaration of the range may be repeated in the map declarations. Maps may declare objects, and such objects are called *state components*. A map generator has visibility to the declarations in the interfaces of its domains and range, as well as the internal declarations of the module generators if the module generator name is the domain indicators. It can therefore name the actions of the interfaces of the domains and range as well as actions of their components.

3.7.2 Transition Rules

The principle feature of maps is the agent state transition rule that also occurs in interface behavior parts (Section 3.5.4). The agent state transition rule provides the necessary expressive power needed to define correspondences between architectures. Each rule observes patterns of events in the domains, triggers on them, and generates a poset in the range. The range poset could be a possible behavior of the range archicture, but it is not *requires* to be. It is generated solely by the map rather than by any execution of the range architecture.

The events that are made available to the map are as follows. If an actual domain of the map corresponds to an indicator that is a type expression, then the map observes the events of the *requires* and *provides* parts (but not a behavior part) of the domain's interface. If an actual domain of the map corresponds to an indicator that is a module generator, then the map observes the events generated in the domain if that event is generated by a call to an action or function of the domain's interface or its components' interfaces – again only the *requires* and *provides* parts (not behavior part). If the actual domain is a map, then the map observes all events generated by the domain. Thus, a map can see inside (one level) of its domain objects, and all the way down inside another map.

The range actions called are limited to those in the range's interface type. That is, a map may call *provides*, *requires* and *private* actions of the range interface type, as well as *provides*, *requires* and *private* actions of modules declared in the range interface type. Maps may also explicitly call the implicitly declared **call** and **return** actions of functions.

3.7.3 Induced Dependencies

A natural question to ask is whether there are any dependencies between: *i*) the events produced in the range, and *ii*) between domain events and range events. The answer to the second part of the question is no. Events of the range poset are causally independent to events of the domain poset. And for the answer to the former question, the range events generated by each triggering of a map rule have at least the dependency order defined by the poset generator in the body of the map's agent state transition rule. Additionally, events generated by separate triggerings of the rules can have a dependency order defined in terms of the dependencies between the domain events that triggered the rules. This is called an *induced* dependency; dependencies among events of the range poset are *induced* from the dependencies among the events of the domain poset.

The user has a choice from several induced orderings as defined below:

For all events e, f generated in the range of some map, f depend upon e if and only if *i*) e and f were generated by the same rule, by the same triggering, and the poset generator in the body of the agent state transition rule specified f to depend upon e, or *ii*) $T \rightsquigarrow U$, where T is the domain poset that triggered e and U similarly triggered f. The following are the definitions of $\rightsquigarrow$ that users may choose from:

none: There is never an induced ordering between map events, i.e., $T \rightsquigarrow U$ iff false.

strong: There is an induced ordering iff all events that triggered e precede all events that triggered f, i.e., $T \rightsquigarrow U$ iff $\forall a \in T,\ b \in U : (a \rightarrow b)$.

maxima: There is an induced ordering iff all events in the maxima (defined below) of the trigger of e precede all events in the maxima of the trigger of f, i.e., $T \rightsquigarrow U$ iff $\forall a \in Maxima(T),\ b \in Maxima(U) : (a \rightarrow b)$.

dominance: There is an induced ordering iff for all events a in the trigger of e, there exists an event b in the trigger of f such that a precedes b, i.e., $T \rightsquigarrow U$ iff $\forall a \in T,\ \exists b \in (U - T)(a \rightarrow b)$.

diff: There is an induced ordering iff all events in the trigger of e precede all events in the trigger of f that aren't also triggering events of e, i.e., $T \leadsto U$ iff $\forall a \in T,\ b \in (U - T) : (a \rightarrow b)$.

Definition 3.7.1 *An event a that is a member of poset P is in the* maxima *of P if and only if there isn't another event b that is also a member of P and causally precedes a, i.e.,* $a \in Maxima(P)$ *iff* $a \in P \wedge \nexists b \in P : b \rightarrow a$.

3.7.4 Conformance to Range Constraints

The range poset generated by a map must satisfy the map's constraints that are derived from the range's constraints. If the range indicator is an interface type, the map's constraints will include that interface's constraints, and if the range indicator is a module generator name, the map's constraints include both the constraints in the interface type and in the module generator's body.

Maps are intended to be used as an architecture analysis tool. An applications of maps has been to runtime compare an architecture with a standard (or reference) architecture as will be shown in Section 5. Comparison of architectures is accomplished by mapping the posets of the domain architecture(s) into behaviors of the range architecture and checking them for consistency with the constraints of the range architecture.

Maps have also been established experimentally to reduce the complexity of large simulations by mapping posets of events in a detailed low–level simulation into single events in a higher level simulation. When a hierarchical design methodology is used to develop a low level detailed architecture from a highly abstract specification, maps relating architectures at successive levels of abstraction can be composed transitively to define maps across several levels.

Maps support consistency checking of architectures at different levels in a design hierarchy, and automated runtime comparison of an architecture with a standard (or reference) architecture. Comparison of architectures is accomplished by mapping the behaviors of the domain architecture(s) into behaviors of the range architecture and checking them for consistency with the constraints of the range architecture. In Chapter 5, maps are used to relate a particular system to the X/Open reference architecture and to check that the system satisfies the formal constraints of X/Open.

3.8 Tool Suite

There are several tools to assist programmers who are using Rapide. The tools include a compiler, a constraint checking runtime system, a graphical poset browser, and an animation facilities for producing, viewing, and animating posets generated by Rapide computations.

3.8.1 Compiler (Rpdc)

The compiler[4] parses Rapide source code, reports syntax and semantic errors, and generates an executable. Executing the executable will generate a log file that is the event log of the computation. Other programs (most notably Pob described in Section 3.8.3) can be used to view the poset of the computation.

3.8.2 Constraint Checking Runtime System (RTS)

Constraint checking runtime system handles the resource allocation during program execution as well as verifying that the computation produced does not violate the constraints of the model.

3.8.3 Partial Order Browser (Pob)

Pob is a tool for graphically browsing the partially ordered event traces generated by Rapide computations. When it is executed, the pob will create three windows. One window contains a menu bar and an area in which graphs will be drawn; we call this the graph window. Another window, called the options window, contains just toggle windows. The third window, architecture, contains the components of the computation that generated the log file.

[4]This tool is actually a translator that rewrites Rapide source into Ada.

3.8.4 Simulation Animator (Raptor)

Animation is the depiction of the event activity in a system on a picture of the system. Movement of messages on a box and arrows diagram is a common animation style. Different graphical animation styles are appropriate for different systems.

Animation is an aid to human understanding. It is a powerful tool in architecture prototyping. Often, in our experience with RAPIDE, animation of a simulation provides the easiest way for a user to assess what a system is doing. Only then is it possible to embark on a more formal process to modify the system, express constraints on its behavior, and so on. Also, animation of distributed systems is aided by causal histories, because the causal dependencies between events imply simple rules about the order in which events should be depicted to give an accurate animation.

The current simulation RAPIDE animator, called Raptor, consists of an active architecture–graphical event player that produces cartooned scenarios of a program execution. It provides a powerful demonstration facility to illustrate and communicate executions of the architecture. The event player depicts the architecture (currently statically) and gives a picture of who can communicate with whom. The tool is active in the sense that it is able to play back an execution of a program. It has been developed with an interface to causal event histories generated from any system, not only the RAPIDE simulator.

Chapter 4

DTP Domain

This chapter introduces the effects of introducing event dependency into the definition and formal specification of transaction processing system architectures, properties, and protocols. Transactions form the basis for these definitions and specifications, and they typically exhibit the following properties [31]:

Atomicity. A transaction's changes to state are atomic: either all happen or none happen.

Consistency. A transaction is a correct transformation of the state. It does not violate any of the integrity constraints associated with the state.

Isolation. Even though transactions execute concurrently, it appears to each transaction, T, that the others executed either before T or after T.

Durability. Once a transaction completes successfully, its changes to state will persist despite failures.

These properties are known by their initials as the ACID properties.

This chapter examines the properties atomicity, isolation and durability, their corresponding system architectures and protocols. Each property and corresponding exempliary implementation is specified via a RAPIDE reference architecture, complete with component interfaces, connections, constraints and behaviors.

Reference Architecture

Architecture, when used in this context, deals with the composition of software components that implements the engine. Each component has an interface that defines the ways the component can communicate with other components. An architecture definition consists of a set of component interfaces and a set of connections between those interfaces. Architecture definitions should be represented formally in a machine–processable reference architecture description language.

The architecture alone is not sufficient to describe a system; the behavior of the system is needed as well. Behavior can be described in two ways, either as a constraint or as executable code. Executable code is useful, because understanding what the system is doing requires a control flow abstraction of the system. Execution of a system can provides the control flow abstraction if tools permit the programmer to observe what happened during the execution. Constraints can also assist the programmer in further understanding the system by describing properties of *all* conforming executions.

Thus, these requirements can be abstracted to define a *reference architecture*, a clear, precise, executable, and testable specification that is how a standard should be represented. A reference architecture contains formal, machine–processable definitions of the component interfaces, their behaviors, and how the components may be connected together into architectures, as well as formal constraints on those behaviors and on the communication protocols between the components. It can be interpreted as both a goal and a yardstick. As a goal, it is clear description of the wanted behavior. As a yardstick, it must be precise and testable.

Views

Each reference architecture presented in this chapter is a *view*, or level of abstraction of distributed transaction processing system. What kinds of things exist in a good view of a system? A good view includes three models of the system: program, situation, and domain models.

Program Model. When code is completely new to the programmer, Pennington [78] found that the first mental representation that programmers build is a *program model* consisting of a control flow abstraction of the program. This abstraction reflects what the program is doing, statement by statement or in a control flow manner. Note that a functional understanding of the program is not yet accomplished.

The RAPIDE animator provides the program model. It is an architecture–based event history player that depicts the execution's control flow as the movement of events between the modules. An event being sent from one module to another in the animation represents the passing of control between those modules. Concurrent activities, i.e., two control flows, is represented in the tool suite as multiple events in motion.

Situation Model. Programmers commonly use a *situation model* to understand the behavior of programs. The situation model extends the information provided via the program model to create a data–flow, functional abstraction. The situation model is a way of tracing why values are generated and what the basis is for those values to be computed.

In RAPIDE the poset browser, interface behaviors, and architectures may each contribute to the creation of a situation model. The filtering capabilities of the poset browser enable the creation of functional abstractions. The dependencies amoung events are based upon the control–flow and data dependencies that occurred during the execution.

Domain Model. An additional mental representation common to programmers is the *domain model.* The domain model is a top–down understanding of how one module consisting of other modules. This model is built from understanding how modules are composed hierarchically; one module consists of additional submodules.

This representation is directly modelled in RAPIDE through the syntactic structure of architectures (and modules). Architectures consists of sets of constituent modules and connection rules.

The three models reflect mental representations of code at different abstractions. Formally, a view of code may include one, several or all of the models.

Evolutionary Prototyping

This chapter develops several reference architectures for formally specifying commitment protocols, concurrency control mechanisms, and recovery strategies. It presents a methodology to analyze these protocols and prototype them on exemplary system architectures. Prototype models are built, because the kinds of systems being modelled are not practical starting points for analysis or experimentation, usually because they are too expensive to build, or inaccessible when built, or simply too complex. The prototyping methodology used in this dissertation is an example of evolutionary prototyping.

Evolutionary prototyping is the process of developing a prototype for a system gradually, satisfying some requirements before attempting to improve its capabilities to satisfy others. There are several reasons to do this. The most important is to understand the requirements themselves. Information gained from the prototype should help uncover inconsistencies, incompleteness and inaccuracies. If we simply try to build a program to satisfy everything a once, we will be in danger of getting one big mess. This is another reason for formalizing programs, because the programs are too complex to reason about in their natural, unified state.

4.1 Transaction Implementation and Execution

This section introduces a model of transaction processing with event dependencies. Event dependencies allow a transaction to be defined as a set of subtransactions T, two partial orderings on those subtransactions (I – an implementation and R – an execution), and a mapping X from T to the version state. Transactions are computer programs, i.e., program code that may be executed. The set of subtransactions is the collection of operations that are contained within the program code. The structuring of that code determines the implementation ordering, and the run time behavior of the program determines the execution ordering. The values read or written by each

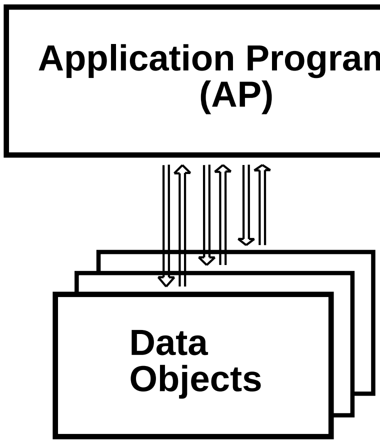

Figure 4.1: Simple Transaction Architecture

operation during the execution determine the version state.

4.1.1 Architecture

A minimal architecture of a transaction processing system that will be used for the purpose of defining transaction implementation, execution, and version state is shown in Figure 4.1. This architecture consists of an application program and a set of data objects. In general, the application program executes transactions that consist of many work requests for the data objects.

Interfaces

Without loss in generality, the work requests made by the application program and handled by the data objects can be modelled as reads and writes. This dissertation models read and write operations on the data objects as four asynchronous actions: read call, read return, write call, and write return. This type of modelling, without loss of generality, further assumes each data object has a single value. A read call event causes a data object to generate a read return event parameterized with the value of the data object. Similarly, a write call event is parameterized with the next value for the data object.

```
type Data_Object(type Data; init : ref(Data) is nil(Data)) is
   interface
      service
         AP : Data_Object_Service(Data);
end interface Data_Object;
```

Figure 4.2: Data Object Interface Type Constructor

```
type Application_Program(type Data; NumObjs : Integer) is
   interface
         service
            DOs(1 .. NumObjs) : dual Data_Object(Data);
end interface Application_Program;
```

Figure 4.3: Application Program Interface Type Constructor

The interface type constructor for data objects may be found in Fig. 4.2, and the interface type constructor for the application program is given in Fig. 4.3. Both interfaces are based upon the Data_Object_Service interface type constructor given in Fig. 4.4. The Data_Object interface type constructor declares one such service, labeled AP, and the Application_Program interface type constructor declares a (*dual*) service set of them, labeled DOs.

Constraints on the interfaces derived from these type constructors will be presented later in this dissertation. Several of these constraints will be faciliated by the *private* read and write actions of the Data_Object_Service interface type constructor.

Connections

The Rapide architecture for this view is given in Fig. 4.5. The architecture declares an application program AP and an array of data objects D. The application program and data objects are connected through their data object services via a connection generator. Note: Only AP and D components with *dual* services can be connected together by this connection generator. Thus, both interface types must be derived from the same data type, which in this case is the type Integer.

```
type Data_Object_Service(type Data) is
   interface
      action
         in     Read_call();
         out    Read_retn(value : Data);
         in     Write_call(value : Data);
         out    Write_retn();
      private
         action
            Read(value : Data; version : Integer);
            Write(value : Data; version : Integer; init : Boolean);
end interface Data_Object_Service;
```

Figure 4.4: Data Object Service Interface Type Constructor

```
architecture AP_DO_Only_Architecture() is
   NumDOs : Integer;
   AP   : Application_Program(Integer, NumDOs);
   D    : array[Integer] of Data_Object(Integer);
connect
   for i : Integer in 1 .. NumDOs generate
        AP.DOs(i) to D[i].AP;
   end generate;
end architecture AP_DO_Only_Architecture;
```

Figure 4.5: AP–Data Object View

```
architecture AP_DO_Only_Architecture() is
   NumDOs : Integer is 2;
   AP  : Application_Program(Integer, NumDOs);
   D   : array[Integer] of Data_Object(Integer)
            is (1 .. NumDOs, default is Single_Version_Object(Integer, ref_to(Integer,0)));
connect
   for i : Integer in 1 .. NumDOs generate
       AP.DOs(i) to D[i].AP;
   end generate;
end architecture AP_DO_Only_Architecture;
```

Figure 4.6: Fully Instantiated Architecture

4.1.2 Behaviors

The architecture of Fig. 4.5 can become executable if its components are associated with modules. A fully instantiated architecture is given in Fig. 4.6. This architecture declares and connects up two data objects to the application program. Each data object is created from the Single_Version_Object module generator given in Fig. 4.7. This module generator connects *private* AP.Read and AP.Write events to (respectively) *out* AP.Read_retn and AP.Write_retn events. Therefore, whenever a module created from this module generator generates a AP.Read (or AP.Write) event, a corresponding AP.Read_retn (or AP.Write_retn) event will also be generated. The module will also react to a AP.Read_call event by generating a Read event with the current value val of the integer object and the current version number ver. AP.Write_call events will generate AP.Write events and update the current value of the integer object and version number.

A default module for the application program is generated from its interface type's behavior. An exemplary application program behavior is given in Figure 4.8. This behavior starts by generating AP.Read_call events for all of the data objects. For each AP.Read_retn the application program receives from a data object, it makes a AP.Write_call to the next data object with the value read as the parameter.

```
module Single_Version_Integer() return Data_Object(Integer, ref_to(Integer,0))
   is
      val    : ref(Integer);
      ver    : ref(Integer) is ref_to(Integer,0);
   connect
      (?va in Integer)(?ve in Integer)
            AP.Read(?va, ?ve) to AP.Read_retn(?va);

      (?va in Integer)(?ve in Integer)
            AP.Write(?va, ?ve, False) to AP.Write_retn();
   parallel
      when AP.Read_call do AP.Read($val, $ver); end when;
   ||
      when (?va in Integer) AP.Read_retn^(~ *) ~ AP.Write_call(?va) do
            AP.Write(?va, $ver+1, False);
            val := ?va;
            ver := $ver + 1;
      end when;
end module Single_Version_Integer;
```

Figure 4.7: Single Version Integer Object Implementation

```
begin
   Start
   =>
         (!i in 1 . . NumObjs ||) DOs(!i).Read_call();;

   (?i in Integer)(?v in Integer) DOs(?i).Read_retn(?v)
   =>
         DOs(?i mod NumObjs + 1).Write_call(?v);;
```

Figure 4.8: Application Program's Behavior

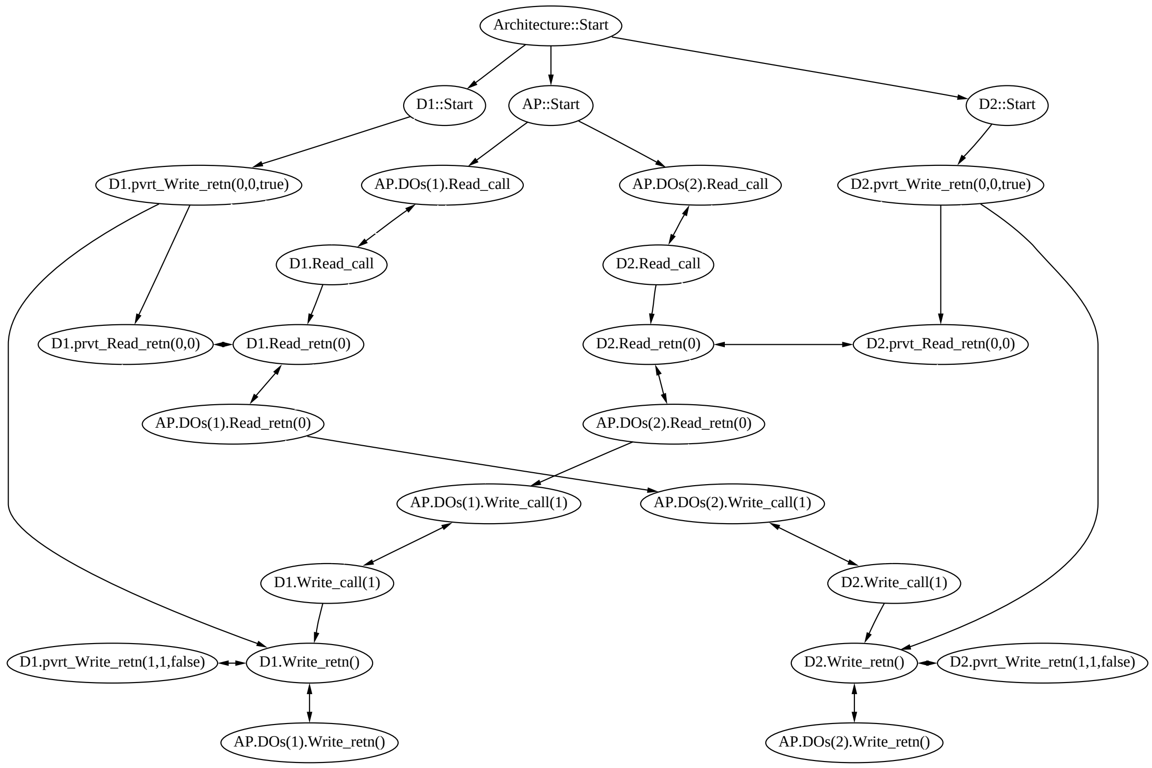

Figure 4.9: An Execution

4.1.3 Execution

An example execution of the architecture instantiated with two data objects is depicted in Figure 4.9. Notice the execution poset is consistent with the implementation, i.e., only new dependencies have been added.

The implementation reflects the relationships intended by the programmer. The partial order is therefore potential causality and refers to the structure of the transaction implementation. This relation, R, reflects the potential causal ordering of the subtransactions that occurred in the execution. These orderings may vary depending upon the architecture of the TP system upon which the transaction was executed. A single–processor, single–process, single–threaded system will have a much different ordering than a multi–processor, multi–process, or multi–threaded system.

Note: in the standard model there is no distinction between a read call and its associated read return, just a single read action. This is possible since the events are totally ordered and they must occur in call/return pairs, simple function call semantics.

From a single data object's perspective during an execution, it may receive read and write calls with arbitrary orderings from its environment. The data object reacts to these calls by generating return events. The relationships and parameters of these events is constrained as per the associated semantic model. There are two primary semantic models, single–version and multi–version. We consider only the single–version semantics in this discussion for brevity.

Single–Version Semantics

In the execution of Figure 4.9, each object goes through a sequence of versions as it is written and read by these pairs of events. Reads do not change the object version, but each time an object is written it gets a new version. Constraints on data objects under a single–version semantics are as follows:

- Returns are generated because of call events.

  match (Read_call $\rightarrow$ Read_retn)^($\sim$ $*$)
  match ((?d **in** Data_Type) Write_call(?d) $\rightarrow$ Write_retn(?d))^($\sim$ $*$)

- Read calls may only return values that were written by the latest write (return).

  match ((?d **in** Data_Type) Write_retn(?d) |> Read_retn(?d))^($\sim$ $*$)

- Object go through sequences of versions.

  match (Write_retn $\rightarrow$ Read_retn^($\sim$ $*$))^($\rightarrow$ $*$)

These dependencies are imposed by the data objects, and additional dependencies may be imposed by the environment.

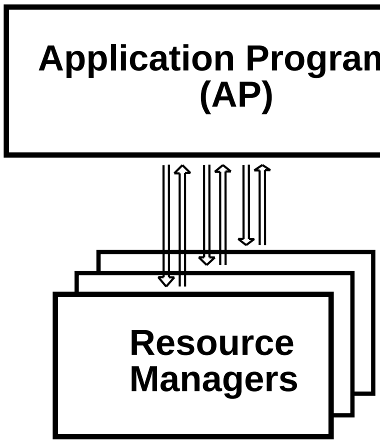

Figure 4.10: Atomicity View Architecture

4.2 Atomicity

Transactions are not guaranteed to execute to completion, but instead are assured to satisfy a weaker but still sufficient property. Transactions are performed entirely or not at all; they cannot be only partially done at termination. The execution of an *atomic* transaction will leave the system in either the state derived from all of the instructions or the initial state.

4.2.1 Architecture

The architecture (see Figure 4.10) for this view includes application program and resource managers. The application program makes the requests that implements the transaction. The architecture encapsulates data objects in resource managers that perform the work requests of the application program. The resource managers are more than data objects; their interfaces must include constituents to allow the work to be committed or aborted.

```
type AP_RM_Work_Request_Service(type Xid) is
  interface
    action
      out Work_Request (x : Xid; p : Parameters);
      in  Work_Results (x : Xid; r : Results);
    requires
      Work_Request : function(x : Xid; p : Parameters) return Results;
  end interface;
```

Figure 4.11: AP an RM Work Request Service

Interfaces

Transaction Identifier. How is the work performed by different resource managers (called branches) tied together into a single transaction? The answer is through an identifier, called the *transaction identifier* or xid. The work requests made by the application program to the resource managers are parameterized with an xid. All the work associated with a particular transaction is requested to the resource managers with the same xid.

Work Request Service. A service that exemplifies what the signature such work requests should have is shown in Fig. 4.11. The application program and resource managers will have dual work request services. Both interfaces will have to share a common Xid type.

Transaction Commitment Service. When the work is finished, the application program is required to explicitly request the transaction to be committed or rolled back by the resource managers, depending on the work results received or user abortion.

Application Program Interface. The application program's interface type contains both the work request and atomicity services. It is dependent upon the xid type and the number of resources it will communicate with.

```
type Transaction_Commitment_Service(type Xid) is
   interface
      action
         in   Commit_call (x : Xid);
         out  Commit_retn (x : Xid; committed : Boolean);
         in   Rollback_call (x : Xid);
         out  Rollback_retn (x : Xid; rolledback : Boolean);
   end interface Transaction_Commitment_Service;
```

Figure 4.12: Transaction Commitment Service

```
type Application_Program(type Xid; NumRMs : Integer) is
   interface
      service
         A(1 .. NumRMs) : AP_RM_Work_Request_Service(Xid);
         B(1 .. NumRMs) : Transaction_Commitment_Service(Xid);
      private
         New_Xid : function() return Xid;
end interface Application_Program;
```

Figure 4.13: Application Program Interface

```
type Resource_Manager(type Xid) is
  interface
    service
      A : dual AP_RM_Work_Request_Service(Xid);
      B : dual Atomicity_Service(Xid);
end interface Resource_Manager;
```

Figure 4.14: Resource Manager

```
architecture AP_RMs_Only(NumRMs : Integer) is
  type Xid is ...
  AP   : Application_Program(Xid,NumRMs);
  RMs  : array[Integer] of Resource_Manager(Xid);
connect
  for i : Integer in 1 .. NumRMs generate
        AP.A(i) to RMs(i).A;
        AP.B(i) to RMs(i).B;
  end generate;
end architecture;
```

Figure 4.15: System Architecture

Resource Manager Interface. The resource manager interface is similar to the interface of the application program, except the services are dual.

Connections

The architecture of Figure 4.15 simply connects the dual services of the resource managers and the application program together.

Behavior

The structure of transactions is modified such that a transaction always generates a new xid before it requests any work from the resource managers. For now, let's assume the application program is able to generate these xids. The only requirement is that each transaction is associated with exactly one xid, and each xid is associated with exactly one transaction; xids are unique and used for just one transaction. This

```
never (?x in Xid; ?i, ?j in Integer)
      ( RMs(?i).B.Commit_retn(?x,True) ~ RMs(?j).B.Rollback_retn(?x,True) );
```

Figure 4.16: Atomicity Constraint

unique xid is used in all subsequent work requests to associate them with this transaction. Upon making requests and perhaps receiving responses to those requests, the transaction is completed with an explicit Commit_retn or Rollback_retn event.

4.2.2 Atomicity Constraint

A transaction identifier should never be the argument of a committed event and an aborted event. As a consequence, any transaction is either committed by all the resource managers or none. The atomicity constraint[1] is written in Figure 4.16.

4.2.3 Two–Phase Commit Protocol

The most common approach to achieving atomicity is the *two–phase commit protocol* [32, 57]. It is so called, because the commitment protocol is divided into two phases. Commitment refers to whether the transaction can end successfully, i.e., can do what it was requested to do.

In Phase 1, the polling phase, the manager for each resource is asked whether it *can commit* its part of the transaction, if it is requested to do so. If a resource manager can commit its work, it replies affirmatively. This positive response is a promise, guaranteeing that if asked to commit, it will. A negative reply reports failure for any reason. Thus, the poll asks whether the resource can commit its transaction branch, and if so, *prepare* the branch for commitment.

When all the resource managers have responded, the decision phase, Phase 2, is entered. If all resource managers responded affirmatively, then all of the resource

[1]This constraint, as written, is outside the current Rapide 1.0 compiler subset for an obscure reasons. Specifically, the placeholder ?i and ?j must be bound by a parameter of the basic patterns. For such a constraint to be checked, the signatures of the commit return and rollback actions must be modified to include the a parameter referring to which resource manager generated the event.

```
type Transaction_Commitment_Service_for_2PC(type Xid) is
  interface
    action
      in   Prepare_call (x : Xid);
      out  Prepare_retn (x : Xid, promised : Boolean);
      in   Commit_call (x : Xid);
      out  Commit_retn (x : Xid; committed : Boolean);
      in   Rollback_call (x : Xid);
      out  Rollback_retn (x : Xid; rolledback : Boolean);
  end interface Transaction_Commitment_Service_for_2PC;
```

Figure 4.17: Transaction Commitment Service for the Two–Phase Commit Protocol

managers are requested to commit (otherwise they are all requested to *undo*) their parts of the transaction thereby restoring the database to a consistent state. Thus, the entire transaction is ensured of being either atomically committed or undone.

Assuming the application program is chosen to generate the poll and collect the responses from the RM's, the interface shared by the application program and resource managers is extended to include the prepare request and reply. Thus, the protocol uses two actions in addition to the atomicity service: Prepare_call and Prepare_retn (see Fig. 4.17).

If the behavior of the application program is extended with the state transition rules found in Fig. 4.18, an execution may be obtained. The execution, of which Fig. 4.19 is a subset, was obtained by instantiating the AP_RMs_Only architecture with the number of resource managers equal to two. The application program begins by generating a new transaction identifier, xid1. This identifier is used by the application program as a parameter of two work requestion; one request to each resource manager. Upon receiving the favorable results from the resource managers, the application decides to attempt to commit this transaction. At this point, it starts executing the two–phase commit protocol. The application polls the two resource managers with the prepare to commit request. The two replies from the prepare requests are both promises, and the decision is made by the application to commit the transaction. At this point the application issues the two commit calls to the resource managers. They both commit their transaction branches, and finally the application

```
(?x : Xid) AP.commit_call(?x)
=>
  (!i : Integer in 1 .. NumRMs ~) RM(!i).prepare_call(?x);;

(?x : Xid)((!i : Integer in 1 .. NumRMs ~) RM(!i).prepare_retn(?x,True))
=>
  (!i : Integer in 1 .. NumRMs ~) RM(!i).commit_call(?x);;

(?x : Xid)((prepare_ret(?x)^(~ *) ~ prepare_retn(?x, False))
≡ ((!i : Integer in 1 .. NumRMs ~) RM(!i).prepare_retn(?x)))
=>
  (!i : Integer in 1 .. NumRMs ~) RM(!i).rollback_call(?x);;

(?x : Xid)((!i : Integer in 1 .. NumRMs ~) RM(!i).commit_retn(?x, True))
=>
  AP.commit_retn(?xid, committed);;

(?xid : Xid) ((!i : Integer in 1 .. NumRMs ~) RM(i).rollback_retn(?xid, rollback))
=>
  AP.commit_retn(?xid, rolledback);;
```

Figure 4.18: Two–Phase Commit Behavior

acknoledges the commitment of the transaction.

The independent, and possibly concurrent execution of the two resource managers is explicitly visible by the two sides of the poset. Width in a poset expresses independence. Dependencies between the two sides indicate synchronization or coordination between the components occurred. The executions of the resource managers was synchronized two times in the execution. First, when the application decided to *attempt* to commit the transaction. This decision was based upon the good results of both resource managers. Presumably, if either of the transaction branches' work was not favorable, the decision would have been made to abort the transaction.

The second time the execution of the resouce managers was synchronized occurred when the application program made the decision to actually commit the transaction. This is easily visible in the poset where the dependencies criss–cross near the bottom of the poset. This criss–crossing is quite typical of the two–phase commit protocol, and is an indication of a complex coordination between the resource managers.

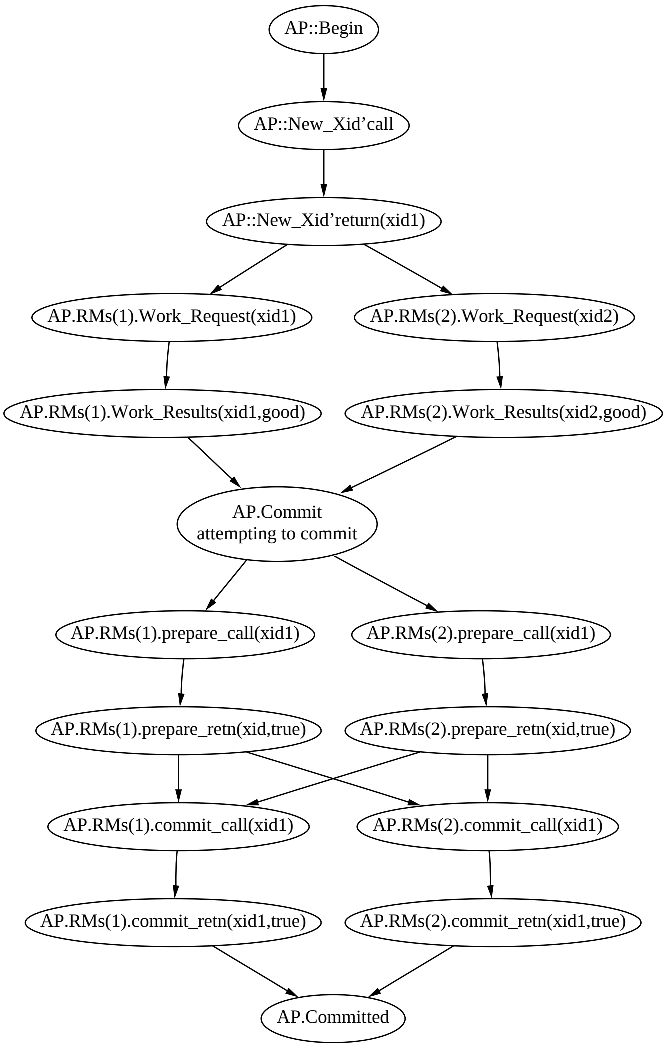

Figure 4.19: Example Execution of Two–Phase Commit Protocol

Coordination

Atomicity when implemented via a two–phase commit protocol implies the following coordination constraint:

Any request to commit a transaction branch must depend upon all of the resource manager's responses to the prepare poll.

It is an important property that the decision to commit must be based upon all of the responses to the prepare poll. The decision to commit cannot be based upon only a few of the responses.

This constraint does not apply to the decision to abort the transaction. The abort decision can be reached after only the first negative response. (In fact, it can be reached even before the polling phase, phase 1, is reached, if the application does not like the results of the work requests.) Thus, the coordination constraint is a constraint on just the decision to commit the transaction and the responses to the prepare poll.

The coordination constraint is expressed in our RAPIDE model with the **never** constraint of Fig. 4.20. This constraint expresses that the execution can never have a Prepare_retn event independent from a Commit_call event where both are parameterized with the same transaction identifier, ?x.

```
never (?x : Xid)( Prepare_retn(?x) || Commit_call(?x) );
```

Figure 4.20: Coordination Property

4.3 Isolation

Isolation answers the question, "How do we define the correct execution of concurrent transactions?" Since it is always possible that transactions will execute one at a time (serially), it is reasonable to assume that the normal, or intended, result of a transaction is the result obtained when it is executed with no other transactions executing concurrently. Thus, the concurrent execution of several transactions[2] is assumed to be correct if and only if its effect is the same as that obtained by running the same transactions serially in some order.

Alternatively this principle can be viewed as a property of transactions. That is, a transaction shall not make updates to the database which affect any other transactions happening at the same time. This defines isolation; transactions are isolated when they do not overlap other transactions, where overlap is with respect to time or database state.

4.3.1 Architecture

The architecture encapsulates data objects by resource managers. The resource manager accepts work requests. In doing the work request, each resource manager may interact with one or more data objects. The reads and writes made on the data objects are associated with the transaction identifier.

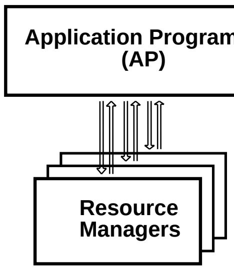

Figure 4.21: Isolation View Architecture

[2]This section only refers to committed transactions.

```
type Resource_Manager(type Xid; type Data) is
   interface
      action
         in     Work_Request(x : Xid; p : Parameters);
         out    Work_Results(x : Xid; r : Results);
      private
         action
            in Read(x : Xid; o : Data_Object);
            in Write(x : Xid; o : Data_Object);
end interface Resource_Manager;
```

Figure 4.22: Isolation View of Resource Manager Interface

```
architecture Example_Resource_Manager(type Xid; type Data)
   for Resource_Manager(type Xid; type Data)
   is
      DOs : array[Integer] of Data_Object(Data);
   connect
      (?x in Xid; ?p in Parameters) Work_Request(?x, ?p) where f(?x,?p)
         to DOs[g(?p)].Read_call();
      ...
end interface Example_Resource_Manager;
```

Figure 4.23: An Example Resource Manager Implementation

4.3.2 Isolation Constraints

Serial Executions

An execution is *serial* if each transaction runs to completion before the next one starts. Specifically, no beginning event of a transaction can be found temporally within any other transaction's begin and end events in the execution. This is the opposite of concurrent execution, where the transactions overlap in time; when the execution of any transaction's events overlaps with another transaction's events. Thus, a serial execution is one in which all of the transactions do not overlap in time.

The during predicate of RAPIDE events is used to define a temporal overlap. The pattern, denoted by A **during**$(t1, t2)$, binds the minimal timestamp of the events contained in A to $t1$ and the maximal timestamp of the events contained in A to $t2$. [3] Thus, two events, A and B, overlap in time if $A_{t1} \leq B_{t1} \leq A_{t2}$ or $A_{t1} \leq B_{t2} \leq A_{t2}$ or vice versa.

However, timestamps are not necessarily comparable. They are definitely comparable if they are associated with the same clock, i.e., if all events share the same clock, then all of the timestamps are totally ordered (comparable). But there may be more than one clock in a RAPIDE model. If there are more than one clock and those clocks are mutually independent, their timestamps will not be directly comparable to each other.

In some cases, a temporal ordering can be inferred between two events, even when they come from independent clocks. A temporal ordering can sometimes be inferred, because causal and temporal properties are related via a consistency property: if event A is in the causal history of event B, then event A must temporally precede B. This property defines $\prec$, the precedence relation between events, and using this relation we can define a serial execution.

Theorem 4.3.2.1 *A property of serial executions, which is a requirement for transaction processing applications, is that each transaction executes upon a consistent*

[3] $t1$ is always less than or equal to $t2$.

```
-- ≺ is a user defined pattern macro. It determines the precedence relation-
-- ships either directly from the temporal relationships or infers them from the
-- causal relationships and transitivity.
 pattern ≺( pattern a, b ) is
    (( ?t1,?t2,?t3,?t4 : Time)
          a during(?t1,?t2) and b during(?t3,?t4) where ?t2<?t3 )
      or ( a → b )
      or ((?c : Event) a ≺ ?c and ?c ≺ b );

-- A serial execution orders all of the transaction executions.
 match ((?t in Xid) Trans_Exc(?t))^(prec *);
```

Figure 4.24: Serial Execution Constraint

database state. This property holds assuming i) the initial database state c_0 is consistent, ii) if a transaction is executed in isolation, it will preserve the database consistency, and iii) the system won't allow arbitrary states to be created after or before the executions of transactions.

Proof: A consistent database state c_n will be obtained from the initial, consistent database state c_0 if the transactions are executed serially, since there will be an ordering of the transactions, $T_1, T_2, \ldots, T_n$, such that T_1 sees the initial state c_0, for each $i > 1$ transaction T_i sees the consistent state, c_i, that will be produced by running $T_1, T_2, \ldots, T_{i-1}$ sequentially to completion, and c_n is the state produced by T_n.

Serializable Executions

Forcing the primitive transactions to obey a serial execution will lead to poor performance, because serial executions do not take advantage of possible concurrency. To permit greater concurrency while preserving consistency, a transaction system's execution of a transaction may be serializable. An execution is *serializable* (SR) if it is "equivalent" to some serial execution of those same transactions. In other words it must be possible to order the actions of the transactions $T_1, T_2, \ldots, T_n$ such that T_1 "sees" the initial database, and T_2 "sees" the database that would have been produced if T_1 had run to completion, etc. The execution produced by this sequential view of the transactions' executions is called a *serialization* or *linearization* of the transactions.

Since serializability is based on equivalence to a serial execution, the question of defining serializability transforms into defining this equivalence. Three common definitions of equivalence can be found in the literature [77]. These are final-state equivalence, view equivalence and conflict equivalence. Each of these definitions describe progressively smaller classes of correct executions, but each smaller class has advantages which offset the reduction in potential concurrency.

Final–state Equivalence. An execution is *final–state* equivalent to a serial execution if it leaves the database in a state which could be reached by a serial execution. Intuitively, two histories are final–state equivalent if they produce the same final state of the database for all interpretations of transactions and all initial database states.

The problem with this equivalence is the database states generated during the execution are unimportant, in that a transaction could execute on an inconsistent database state. Since it is important that each transaction executes upon a consistent database state and final–state serializability allows executions which violate this property, final–state serializability is not suitable for validating executions of the programs used in transaction processing.

View Equivalence. *View* equivalence is a subset of final–state equivalence, thus an execution which is view serializable ends in a state which could be reached by a serial execution. An execution is view equivalent to a serial execution if every transaction executes upon a database state that could be generated by a serial execution of the same transactions. Thus, every transaction executes on a consistent database since every transaction in a serial execution must execute upon a consistent database.

In order for an execution to be view serializable, their must exist a serial execution in which every read in both executions returns the value written by the same write action (same version). A major problem with VSR is that it is an NP–complete problem to determine if an execution is view serializable [77].

Conflict Equivalence. *Conflict* equivalence is a proper subset of view equialence, thus an execution which is conflict equivalent ensures all transactions to view consistent databases. Conflict equivalence constrains that every pair of conflicting operations between transactions should be ordered the same as in a serial execution. Conflict refers to the ability of transaction's effect on the objects to adversely affect another transaction's effect on the objects. If such interference is possible, we say those transactions are in conflict. This definition is used commonly used as "the" definition of serializability, since it can be recognized in polynomial time, and there exist efficient algorithms to ensure that an execution is conflict serializable.

A transaction's effect on the objects can be restricted by limiting the number of objects accessible to it. For this reason, every transaction is associated with a set of objects, called its *view set*. The view set contains all the objects and their values in the database potentially accessible to the transaction. If two transactions have view sets which have an empty intersection then they do not overlap, and they cannot conflict. This reflects overlap with respect to database state. However, we can do better than that! Just because two the view sets of two concurrent transactions intersect, that does not mean the transactions will adversely affect one another.

Of course, the atomicity property implies the effects of a transaction on the objects are conditional upon the outcome of the transaction. When an object in the view set of a transaction is accessed by the transaction, the object and its value becomes a member of the transaction's access set. Each write action overwrites its value in the access set. Upon abortion, the object in a transaction's access set is restored back to the view set. Upon commitment, the access set is written to both the view set and the persistent database. Using the objects the transactions actually access is much more accurate then the objects in the view set. But the intersection of two access sets is still too restrictive to use for defining database state overlap.

The access set is really the union of two simpler sets, the *read set* and the *write set*. The read set of a transaction is the set of objects the transaction reads, and the write set is the set of objects the transaction writes. Thus, two transactions *conflict* if the read set or write set of one intersects the write set of the other. If two transactions have one or more conflicts, all of the conflicts must be ordered the same as in a serial execution.

Two read actions by two different transactions to the same object cannot violate consistency, because reads do not change the object state. Thus, only write actions may create violations. Two write actions to an object by the same transaction do not violate consistency because we assume that the transaction knows what it is doing to its data; we assume that if a transaction runs in isolation, it will correctly transform the system state. Consequently, only write–related interactions between two concurrent transactions can create inconsistency or violate isolation. (See Figure 4.25.)

```
-- Two events conflict if they are from two different transactions and at least
-- one of them is a write.
pattern Conflict(pattern ?e1, ?e2) is
    (?o : Data_Object; ?x1, ?x2 : Xid)
    ( (?e1@Write(?x1,?o) ~ ?e2(?x2,?o))
          or (?e2@Write(?x1,?o) ~ ?e1(?x2,?o))) where ?x1/=?x2;

-- Observe pairs of conflicting events from two transactions. Note: The two
-- pairs must differ.
observe ((?x1,?x2 in Xid; ?e1,?e2,?e3,?e4 in action(x : Xid; d : Data_Object))
      (( ?e1(?x1) and ?e2(?x2) and ?e3(?x1) and ?e4(?x2) )
            where ?x1/=?x2 and (?e1/=?e3 or ?e2/=?e4))
         <-> ( Conflict(?e1,?e2) and Conflict(?e3,?e4)))
-- These pairs must be expressible / mappable into conflicting pairs where the
-- conflicts must be ordered the same.
match (?x1,?x2 in Xid; ?e1,?e2,?e3,?e4 in action(x : Xid; d : Data_Object))
      (( ?e1(?x1) → ?e2(?x2) and ?e3(?x1) → ?e4(?x2) )
        <-> ( Conflict(?e1,?e2) and Conflict(?e3,?e4)));
```

Figure 4.25: Isolation – Conflict Serializability Constraint

Theorem 4.3.2.2 *An execution is a conflict serializable execution, denoted CSR–execution, iff* $\prec^*$ *is an acyclic relation, where the relation* $\prec^*$ *is defined for an execution E as follows: i) if* Write(?i,?o) $\prec$ A(?j,?o) **where** ?i /= ?j *occurs in E then* $(t_i, t_j) \in \prec^*$*, and ii) if* A(?i,?o) $\prec$ Write(?j,?o) **where** ?i /= ?j *occurs in E then* $(t_i, t_j) \in \prec^*$*.*

This is fairly intuitive: if the relation has no cycles, then the transactions can be topologically sorted to make an equivalent execution history in which each transaction ran serially, one completing before the next began. If there is a cycle, such a sort is impossible, because there are at least two transaction, such that T1 ran before T2, and that T2 ran before T1.

Theorem 4.3.2.3 *An execution E of transactions T is a CSR–execution iff there exists a serial execution S of T such that every conflicting pair of events in E is ordered the same as that same pair in S.*

Proof: Part 1: A CSR–execution is conflict equivalent to a serial execution.

Assume that E is a CSR–execution with a corresponding $\prec^*$. Select a serial execution S and corresponding $\prec$ such that $\prec^* \subseteq \prec$. Since in E all conflicts are ordered the same, the executions of T may be topologically sorted, and such a S and $\prec$ must exist.

Without loss in generality, assume that Write(?i,?o) $\rightarrow$ Read(?j,?o) is in S, but Read(?j,?o) $\rightarrow$ Write(?i,?o) is in E. By definition of a CSR–execution $t_j \prec t_i$. However, $t_j \not\prec t_i$, since $t_i \prec t_j$ is already assumed and that would make $\prec$ contain a cycle. This violates the hypothesis that $\prec^* \subseteq \prec$. Therefore, Write(?i,?o) $\rightarrow$ Read(?j,?o) must be in E.

The other five cases are identical.

Therefore, there exists a serial execution which satisfies the conditions for E.

Part 2: If an execution E is conflict equivalent to a serial execution then that execution is a CSR–execution.

Given transactions T and an execution E on T. Assume there exists a serial schedule S on T such that the conflict properties hold. We must show that E is a CSR–execution.

Assume E contains a cycle of the form, $t_i, t_j, t_k, \ldots, t_m, t_i$. By definition, there exist relationships in E of the form A(?i,?o) $\rightarrow$ A(?j,?o), where A(?i,?o) conflicts with A(?j,?o), A(?j,?o) $\rightarrow$ A(?k,?o), where A(?j,?o) conflicts with A(?k,?o) and so on all the way through A(?m,?o) $\rightarrow$ A(?i,?o), where A(?m,?o) conflicts with A(?i,?o). Note that the ?o in the A(?x,?o) only have to be the same object in each pair of conflicting steps, but different pairs may conflict on different objects.

By the hypothesis, these relationships must be in S. Therefore, by the definition of a serial execution, $\prec$ contains the same cycle as $\prec^*$. However this contradicts the hypothesis which states that S is a serial execution. Therefore, E has a $\prec^*$ which is acyclic. Therefore, E is a CSR–execution.

The various ways isolation can be violated are characterized by the cycles in $\prec^*$. Cycles take one of only three generic forms [31]. Each form of cycle has a special name: lost update, dirty read, or unrepeatable read.

Lost Update. Transaction t1's write is ignored by transaction t2, which writes the object based on the original value.

never (Read(?t2,?o) → Write(?t1,?o) → Write(?t2,?o)) **where** ?t1/=?t2;

never (Write(?t2,?o) → Write(?t1,?o) → Write(?t2,?o)) **where** ?t1/=?t2;

Dirty Read. Transaction t1 reads an object previously written by transaction t2, then t2 makes further changes to the object. The version read by t1 may be inconsistent, because it is not the final (committed) value of o produced by t2.

never (Write(?t2,?o) → Read(?t1,?o) → Write(?t2,?o)) **where** ?t1/=?t2;

Unrepeatable Read. Transaction t1 reads an object twice, once before transaction t2 updates it and once after committed transaction t2 has updated it. The two read operations return different values for the object.

never (Read(?t1,?o) → Write(?t2,?o) → Read(?t1,?o)) **where** ?t1/=?t2;

4.3.3 Two–Phase Locking Protocol

The role of the transaction system can be viewed as determining when an object can be promoted from a transaction's view set to its access set. It is not the case that entitles can always be promoted. A concurrency control is used to control this promotion and guarantee the correctness of the responses to the concurrent requests. The concurrency control is that portion of the system that is concerned with deciding which operations should be taken in response to requests by the individual transactions to read and write into the database. Every transaction system that allows queries and updates to be executed concurrently must restrict the interleaving of the read and write transactions of different users, if these users are always to see the database that they share in a consistent state.

A useful tool for restricting execution to only easily recognizable serializable executions is to use one or more protocols, which all transactions must follow. A *protocol*, in its most general sense, is simply a restriction on the sequences of actions that a transaction system may perform.

There are two types of protocols (or concurrency control approaches) for ensuring correctness of concurrent transactions: optimistic and pessimistic protocols. In the optimistic protocol, transactions are permitted to execute in a private buffer at will. All updates are tentative. At commit time, correctness checks are made.

Pessimistic protocols assume a transaction depends on only a small portion of the database state. Therefore, conflict can be avoided by partitioning the objects into disjoint classes. Transactions depending on a common portion of the state must still be serially executed. Unfortunately, it is usually impossible to examine a transaction and decide exactly what portion of the state it will use. For this reason pessimistic protocol are generally restricted to a more flexible scheme, which requires that the transaction acquire locks for objects it will be using. The transaction then operates on the actual database. When a transaction aborts or fails, the system must restore the state for all the objects the transaction has modified. Let's examine pessimistic protocols in more detail.

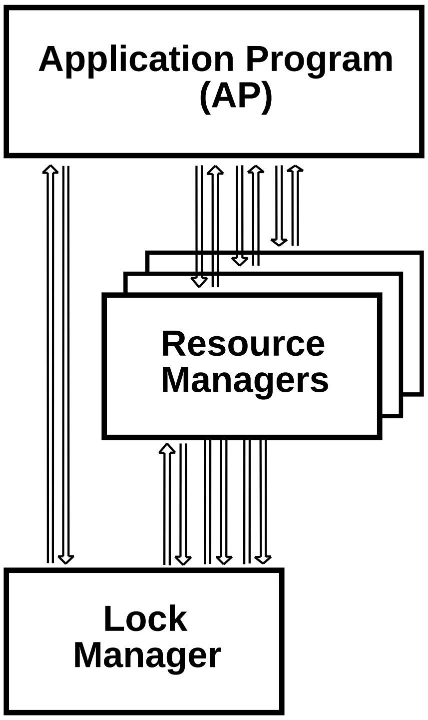

Figure 4.26: Lock–based Pessimistic Protocol Architecture

Locks

Pessimistic protocols based upon *locks* can ensure that if every transaction adopts the protocol, then every transaction is assured to never access data that is temporarily inconsistent. Locks are a synchronization method by which shared resources can be partitioned between transactions. A *lock* is an access privilege associated with a single data object that can be granted or denied for a transaction.

The architecture, sketched in Fig. 4.26, for lock–based pessimistic protocols adds to the isolation architecture a special kind of resource manager, called a lock manager, that the other resource managers must call to dynamically reserve the shared data objects.

The lock manager accepts two types of requests, a lock and an unlock request. Each request contains the name of the data object being locked and the identifier

```
type Lock_Service is
  interface
   action
    in   lock_call(x : Xid; o : Data_Object);
    out  lock_retn(x : Xid; o : Data_Object);
    in   unlock_call(x : Xid; o : Data_Object);
    out  unlock_retn(x : Xid; o : Data_Object);
end interface Lock_Service;

type Lock_Manager(NumRMs : Integer) is
  interface
   service
    Rs(1 .. NumRMs) : Lock_Service;
  constraint
   match (?o in Data_Object) ((?x in Xid; ?i, ?j in Integer)
                    Rs(?i).lock_retn(?x,?o) → Rs(?j).unlock_retn(?x,?o)^(→ *))^(~ *);
end interface Lock_Manager;
```

Figure 4.27: Lock Manager Interface

of the transaction making the request. The lock manager generates a corresponding reply when the request is fulfilled. These requests can be viewed as functions or pairs of actions, and an example of the pairs of actions interface is presented in Fig. 4.27. If transaction x holds a lock on object o, then certain guarantees are made to x with respect to o; x will be guaranteed that no concurrent transaction will be able to acquire a lock on o until x releases its lock. This type of lock is commonly called, the *exclusive lock*.

A module generator that produces exclusive lock managers is given in Fig. 4.28. It uses a process generator to generate a separate process for each data object o being protected. Each process in turn accepts lock_call event for a particular xid ?x, generates a lock_retn event granting a lock for that transaction, waits for an unlock_call event by the same transaction, generates an unlock_retn event, and waits for another lock_call event.

```
module Locker_gen(NumRMs, NumObjs : Integer) return Lock_Manager(NumRMs) is
  parallel
    for o : Integer in 1 .. NumObjs generate
      when (?i in Integer; ?x in Xid) Rsrcs(?i).Lock_call(?x, o) do
            Rsrcs(?i).Lock_retn(?x, o);
            await (?j in Integer) Rsrcs(?j).Unlock_call(?x, o);
            Rsrcs(?j).Unlock_retn(?x, o);
         end when;
      end generate;
end module Locker_gen;
```

Figure 4.28: Lock Manager can be implemented with a process generator.

Well–formed Transactions

Consider programs that interact with the database not only by reading and writing objects but also by locking and unlocking them. Lock–based pessimistic concurrency control protocols require the transactions to dynamically reserve data before reading or writing them, and that the operation of locking acts as a synchronization primitive. These protocols further require that each transaction will unlock any object it locks, eventually; if a transaction aborts without releasing the locks, the system itself must release the locks held by the aborted transaction. This defines well–formed transactions. A transaction is *well–formed* (see Fig. 4.29) if the read sets and the write set of each of its access statements are contained in the collection of parts of the database locked (and not yet unlocked) by the transaction.

```
-- Well-formed Transactions lock before and unlock after accessing an object.
match (?x in Xid) ((?o in Object)
   lock(?x,?o) ≺ (Read(?x,?o) or Write(?x,?o))^(~ *) ≺ unlock(?x,?o))^(~ *);
```

Figure 4.29: Well–formed Transaction Constraint

The Two Phases

A less obvious fact is that consistency requires that a transaction be divided into a growing and a shrinking phase. In the growing phase a transaction only acquires (does not release) new locks, and in the shrink phase the action only releases (does not acquire) locks. Two–phase locking works because it insures that the order in which any two transactions access the same object is that same as the order in which those actions access any other object. The underlying assumption is that if two executions result in the same order of access at each object then the executions are conflict equivalent. Given the conflict equivalence relation on executions, it has been proven that if the two–phase locking protocol is used then any actual execution is equivalent to at least one serial execution [23]. That is, two–phase locking insures serializability.

```
-- Growing and Shrinking Phases.
match (?x in Xid) lock(?x)^(≺ *) ≺ unlock(?x)^(≺ *);
```

Figure 4.30: Two–Phase Locking Constraint

4.4 Durability

It is not enough to set the system to just any consistent database state whenever a transaction is stopped. The system should not erase effects that have been guaranteed to be permanent for other transactions. If a transaction is durable, then its changes will persist. However, durability in this ideal form is impossible to implement because of processor failure, transmission failure, disk crash or even catastrophic failures like natural disasters. In each of these cases, information concerning the database is lost.

Once such a loss is detected, the system restores the lost database state to the state that existed prior to the occurrence of the failure. Detecting the loss is not very difficult, and the primary concern of the database designers is the restoration. Performing this restoration is usually accomplished through the initiation of various backup and crash recovery procedures depending on the type of storage that failed.

4.4.1 Architecture

Storage is traditionally viewed as a hierarchy. The lowest level of storage is volatile storage, like main and cache memory. This data is transient and easily lost due to system crash or power failure. Information residing in nonvolatile storage usually survives system crashes but may be lost. Typical nonvolatile storage devices are disks and magnetic tapes, which have failures like media failures or head crashes. Stable storage is the highest level of the storage hierarchy, and information residing in it is "never" lost. Theoretically, this cannot be guaranteed, because stable storage media are still susceptible to catastrophic failures like fires, earthquakes, floods, wars or acts of God. The design of such media involves using many nonvolatile storage devices distributed (perhaps geographically) in many locations so that the data is as durable as the real world.

Several strategies have been used to handle the many different types of failure on the above types of storage:

Failures without loss of information: In this case, all information in storage is still available. These failures include discovering a software error condition, like an arithmetic overflow, a division by zero or deadlock. *Deadlock* is when a partially executed transaction may not be able to continue, because it requires resources held by other waiting transactions. This would present a situation where they will remain waiting forever. The traditional solution for handling deadlock is to fail one of the transactions, undo its effects, and then release its resources. Ensuring atomicity of transactions in the presence of transaction failures without loss of information is called *fault tolerance*.

Failures with loss of volatile storage: The most common tool for protecting against loss of data stored in volatile storage in the face of power or other system failures is to maintain a copy in nonvolatile storage. The difficulty in handling this approach is to optimize the performance benefits of storing data in volatile storage with the durability benefits of nonvolatile storage. Ensuring atomicity of transactions in the presence of system crashes is called *crash recovery*.

Failures with the loss of nonvolatile storage: Ensuring atomicity of transactions in the presence of media failures or head crashes involves periodically dumping or archiving the entire contents of the database to stable storage. This assumes the duplicate copy of the data will have an independent failures from

```
type Durability_Service() is
  interface
    type Storage_Level is enum Volatile, Nonvolatile, Stable end enum;
    provides
      level : Storage_Level;
    type Failure_Type is
      enum
        Without_Loss_of_Information,
        With_Loss_of_Volatile_Storage,
        With_Loss_of_Nonvolatile_storage
      end enum;
    action
      out Failure (t : Failure_Type);
  end interface Durability_Service;
```

Figure 4.31: Durability Service

the primary copy on stable storage. To implement an approximation of stable storage media, the information must be distributed (perhaps geographically in many locations) among many nonvolatile storage devices with independent failure modes. Ensuring atomicity of transactions in the presence of nonvolatile (and stable) storage failures involves maintaining (perhaps several) redundant backups.

All of the designs for the recovery strategies and algorithms are based upon redundancy with independent failure modes.

Durability Service. The durability service shown in Fig. 4.31 is an extension to the interface of the resources. Each resource will have a specific storage level and have various failures.

4.4.2 Durability Constraint

The durability of transactions is assured using the various storage devices via recovery strategies and protocols. The difficulty in designing them is in the methodology of optimally using the storage hierarchies such that modifications to the database persist only if the associated transaction commits. All of the designs for the recovery strategies and algorithms are based upon redundancy with independent failure modes. The durability constraint is written in Fig. 4.32.

```
-- Values are written multiple times on medium with different levels of protection.
match (?rm, ?rm2, ?rm3 in Resource_Manager; ?o in Data_Object; ?x in Xid)
      (?rm.Write(?x,?o)   where ?rm.level  = Volatile ~
       ?rm2.Write(?x,?o)  where ?rm2.level = Nonvolatile ~
       ?rm3.Write(?x,?o)  where ?rm3.level = Stable )
```

Figure 4.32: Durability Constraint

4.4.3 Write–Ahead Logging protocol

Recovery protocols control the redundant copies of the database's resources to ensure durability.

Log–based Recovery

Database systems usually maintain a history of the transactions' executions, and this history is called a *log*. Logs generally reside on at least nonvolatile storage devices, preferably stable storage devices.

A log is often viewed as a table or sequence of log records. The transaction and other resource managers use the log to record all of the changes that are made to the database as well as status information about the transactions. The log provides read and write access on the log table to the transaction and resource managers.

Begin, write, commit and abort events are appending to the end of the log where their associated events happen. The connections of such an architecture are:

```
Begin(?t)'return => Log.Append(Begin,?t);
Write(?t,?o,?new_value,?old_value)'return =>
      Log.Append(Write,?t,?o,?new_value,?old_value);
Commit(?t)'return => Log.Append(Commit,?t);
Abort(?t)'return => Log.Append(Abort,?t);
```

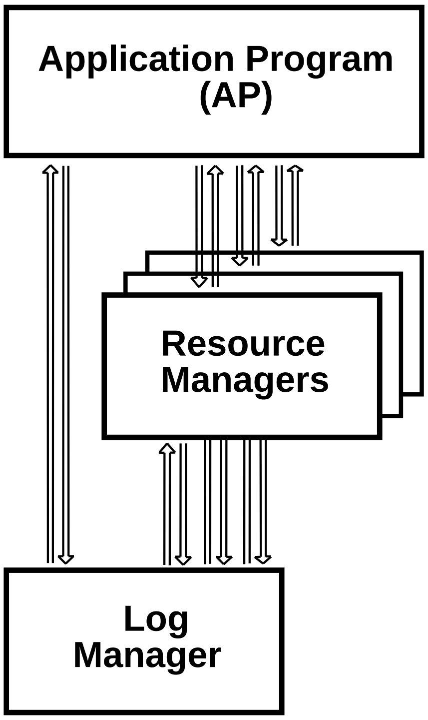

Figure 4.33: Log–based Protocol Architecture

Write–Ahead Constraint

We will discuss one log–based recovery protocol, called *write–ahead logging.* The write–ahead logging protocol requires the changes to the database be recorded in the log prior to updating the database state. This constraint is written in Fig. 4.34.

```
-- Changes to the database must be recorded in the log (a nonvolatile storage
-- device) prior to updating the database state.
match (?rm, ?rm2 in Resource_Manager; ?o in Data_Object; ?x in Xid)
    (  ?rm.Write(?x,?o)  where ?rm.level  = Nonvolatile ≺
       ?rm2.Write(?x,?o) where ?rm2.level = Volatile );
```

Figure 4.34: Write–Ahead Constraint

Write–Ahead Behavior

The fundamental concept for write–ahead logging is to record enough information in the log such that upon failure the transactions can be executed in reverse, thereby restoring the database state. This is like leaving a trail of breadcrumbs such that our steps can be reversed (undone) and we can return to our original starting point.

Note: we only want to restore the failed transactions. The committed transactions' affects must still persist. Thus, the write–ahead logging protocol also requires the status of the transactions to be recorded into the log. Using this status information, specifically whether a transaction has been completed or is still executing, the protocol can recover the persistent database state.

Undo protocol. Transactions that are executing when a failure occurs are generally aborted. Transaction abortion is performed by *undoing* or *rolling back* the actions of the transaction. The transaction manager performs rollback by reading the transaction's log backwards (most–recent–first order). Each log record carries the name of the resource manager that wrote it. Thus, the transaction manager calls the "undo" function of that resource manager and passes to that function the log record. Upon calling a resource manager's undo function with the log record, the resource manager will undo the action that wrote the log record, returning the data object to its old state. If this is repeated for each action

This rollback algorithm is a palindrome. It's behavior can be specified in RAPIDE using the following pattern macro:

pattern Rollback(?x : Xid) **is**
  (?d **in** Data) Action(?x,?d) $\prec$ Rollback(?x) $\prec$ Undo(Action(?x,?d))
  or
  Failure;
end Rollback;

Chapter 5

X/Open Reference Architecture: Case Studies

5.1 Introduction

Software developers have begun to recognize the importance of studying the design of the structure of large and complex systems. As the size of software systems increases, the design problem goes beyond the algorithms and data structures of the computation. *Software architecture* is concerned with the design of the gross overall system structure.

Concentrating on this area of software engineering is important, because an architecture improves at least four aspects of software systems development:

Understanding: Software architecture facilitates our ability to comprehend large systems by presenting them at a higher level of abstraction. Architecture is not concerned with the details of *how* communication occurs, but rather concentrates on effectively describing the structure of communication — the kinds and numbers of components and protocols through which they communicate.

Interoperability: Software architecture can be an effective means of establishing a common framework (or standard) across various domain–related products.

Standard architectures specify systems that can be built by composing individual components, similarly to computer chips on a printed circuit board. Such architectures are especially useful where each software component may be coded in a different programming language.

Analysis: Open systems whose interchangeable components must share a common software architecture are highly dependent that architecture, where subtle changes can greatly influence system response–time, throughput and data–flow bottlenecks. Architectural descriptions provide new opportunities for analysis of system properties, including advanced forms of system correctness, consistency and conformance checking, as well as improved performance prediction.

Evolution: Architectural descriptions separate concerns of the functionality of a component from the ways in which that component communicates with other components. This allows one to efficiently evolve a system's design by reusing most, if not all, of the system's components, and only redesigning the communication paths or by reusing the paths and replacing the components.

Much of architectural description in practice is largely an ad hoc "boxology": drawings in which boxes represent components, and arrows represent unspecified interactions among those components. Architectures are often nested where a component in one architecture is described by a hierarchy of (sub)components or architectures. Practicing engineers interpret the drawings with respect to common families of architectures or *architectural styles*, e.g., pipe–and–filter, blackboard, and client–server. Consequentially, such architectural designs are: *i*) handcrafted, imprecise, unmaintainable, and only vaguely understood by developers, *ii*) not analyzable for consistency or completeness, *iii*) not enforced as a system evolves allowing multiple and perhaps unintended interpretation, and *iv*) supported by virtually no tools to help the architect.

The recent trend toward standardization, e.g., Microsoft's OLE [8, 13], OSF's DCE [34, 76], KQML [75], X/Open and OMG's CORBA [33], has produced a growing recognition for the importance of software architecture and has lead to much more explicit use of architectural design. Such standards attempt to specify a framework for building "open" software systems. Open in the sense that the software system's architecture is sufficiently specified by the standard's documentation so that multiple vendors can produce interoperable components. This obviously requires considerable precision and completeness in the architectural description. However, it is common for such standards to be specified entirely in English. Occansionally, the English descriptions are augmented with state tables. While state tables are a step in the right direction, they are not enough, and researchers have developed other technologies to go further.

Architecture description languages (ADLs) are an improvement over current practice in describing software architectures. Generally, they are expressive notations for representing architectural designs. An architecture should precisely and explicitly describe the number of components, their interfaces, and how those components are connected together (communicate) into larger systems. The architecture should also describe the features required of the target system, as well as the features *not* allowed to appear in the target system.

For example, one type of architectural feature is inter–components communication. In particular, an architecture often constrains the components a particular component can communicate with. This concept is called *communication integrity* [68]. A precise architecture description will describe for each component (or component type) specifically what components it can *and* can't communicate with.

ADLs are able to capture various features of software architecture that have been hereto only vaguely understood by developers. They provide an opportunity to go beyond architectural description and into architectural analysis. This opportunity is maximized if the ADL has formal, mathematical foundation. A formalization gives us semantic precision and a common basis for formal analysis. The basic requirement for a formal ADL is its ability to formally capture design requirements, e.g., modularization and system composition, characterizations of system properties, and theories of inter–component communication.

Rapide [9, 67, 83, 84, 85, 86, 87, 88, 89] is an example of a formal ADL. It is an event–based, concurrent, object–oriented language with formal constraints specifically designed for prototyping system architectures. A standard is represented in

RAPIDE by a reference architecture. RAPIDE reference architectures facilitate accurate description and modeling of an architecture standard.

Reference architectures contain the architectural constraints of the software standard. These constraints are of the following two kinds: *i*) *never* constraint that constrain the execution to never produce a pattern of behavior, or *ii*) *protocol* constraint that constrain the execution to be a repeating pattern of behavior. While expressing a specific constraint in RAPIDE is not difficult, uncovering all of the constraints about a system often is. The first formalization of a constraint is commonly not quite right, often because the designer's understanding of the system is inaccurate or incomplete. RAPIDE uses simulation techniques to prototype and animate the behavior of reference architectures.

Prototyping and animation assists in teaching the designer about the system, and helps uncover and test the correctness of the constraints. They promote *creeping formalization*, a process of incrementally introducing formalization into a model. This methodological strength lies in not requiring a complete formalization at and below each step in the design process. Constraints can be added at any time in the standardization process. As test scenarios are developed, they can be executed by the reference architecture whose behavior can be animated and run–time checked with the constraints. A reference architecture also suitable for successive refinement into the actual system, and at each point, the system can be executed, animated, and automatically run–time checked for conformance to the architecture's constraints. This has great appeal for industrial customers, since they prefer methodologies that can be done by "mortals" (not just experienced software architects).

We sought to our claims through the application of RAPIDE to specify and test conformance to a distributed transaction processing industry standard, the X/Open DTP reference model [101]. We conducted four case studies. The first case study was conducted to capture the architectural and high–level transaction constraints found in the X/Open industry standard. The second case study was conducted to add typical behaviors to the components of the architecture, and to check that the behaviors satisfied the system constraints. The third case study was conducted to use the specifications of the second case study as a reference architecture and test the conformance of a exemplary system, thereby using the reference architecture as a standard. The fourth case study was conducted to extend the reference architecture with several additional constraints and prototypical behaviors to see how well a RAPIDE architecture evolves.

5.2 Background

Many organizations are developing standard protocols that allow transaction processing systems to interoperate [38, 40, 43, 45]. These standards capture the core algorithms implemented by most transaction processing systems. One emerging standard that is the focus of this chapter is from X/Open, an independent, open systems consortium — the X/Open standard for distributed transaction processing [101]. This chapter investigates the influences of the atomicity, isolation, consistency and durabiilty properties on the X/Open standard, and uses RAPIDE, a new prototyping language, that allows the definition, specification, execution, and analysis, especially the architectural aspects, of such systems.

Distributed transaction processing (DTP) provides mechanisms to combine multiple software components into a cooperating unit that can maintain shared data. This enables construction of applications that manipulate data consistently using multiple products and that scale by adding additional hardware and software components. This construction is facilitated by the development of portable application program interfaces and system–level interfaces that enable the portability of application program source code and the interchangeability and interoperability of system components from various vendors.

The X/Open DTP standard is a framework for building systems that implement the atomicity, isolation, consistency and durability properties. It describes protocols, integrity constraints, and state transition diagrams that constrain the legal sequences of routines that may be executed. The standard is still evolving and currently only defines a commitment protocol for ensuring atomicity, and even in this definition no guarantees are made in the presence of failures. In the case of isolation, only very brief descriptions are given that relate to only the most basic forms, even though more relaxed forms of isolation are viewed as a significant advance over SQL2 [44].

5.2.1 Open Architecture Systems

A basic premise of the X/Open Company Limited is that compliant systems can be built with composable parts from more than one software vendor at relatively low cost by "instantiating" an "open" shared design. The X/Open DTP shared design is a software architecture — that is less concerned with the algorithms and data structures used *within* components than with the overall system structure. Structural issues include: gross organization and global control structure; assignment of functionality to system components; protocols for communication and synchronization; scaling and performance; and selection amoung design alternatives. The X/Open DTP architecture defines constraints on the structure of compliant system instances [79, 2] and is therefore a *standard*. A standard in the sense that it is a constraint on compliant systems.

Codification of such architectural standards can be critically important in assuring that the various components of a system are integrated correctly. Architectural requirements and behavioral protocols are typically expressed by X/Open in scenario form. That is, required behaviors of the software were specified by sequences of events in structured English. While this language of events is shared by large groups of developers, it is an informal language. We speculated that significant benefits could accrue from formalization of the scenarios.

5.2.2 Branding

The added complexity and design options for distributed systems have resulted in greater demand for various kinds of systems development assistance. Prototyping provides assistance to ensure that the system architecture is viable before embarking on the costly task of implementation. However, once a viable prototype's software architecture has been developed, the role of the prototype changes. It becomes a standard (or a reference architecture) to which the actual implementations must adhere, since the implementations will not necessarily share the same architecture as the prototype. Thus, another kind of assistance that is desirable for complex distributed systems is automated testing that an implementation conforms to the software architecture and constraints of the reference architecture.

The importance of conformance testing is well recognized and practiced by organizations that develope standards. One function X/Open Ltd. performs is the *branding* of compliant components. X/Open has introduced a trademark to identify products that conform to its specifications. Branding promises the user that the product works as outlined by the standard. When branding works (and X/Open hasn't yet got it working across the board), it ensures that information systems buyers can choose platform vendor, or even application vendor, without fear of interoperability problems.

At present, X/Open endorses 750 products that carry the X/Open brand. To obtain an X/Open branding certificate, the software vendors must follow the specifications of X/Open. This assures that the vendors use the relevant X/Open specification, national or international standards. Next, the software vendor performs a verifiable test and applies to X/Open with the results of the test.

There is no direct relationship between the branding process' test suites and the specifications. It would be preferable if the tests were derived directly (even automatically) from the specification. Testing can be done from proving code correct (highly unlikely) or from verifying via testing conformance of executions.

X/Open's test suites are *end–to–end* tests that provide inputs to and examine results of a system's execution. This type of testing only treats the system as a "black box" with inputs and outputs. However, it is impossible to determine using such a method if a system conforms to a standard that requires a particular protocol for a system's execution (like the X/Open DTP requirement of the two–phase commit protocol). This is because the execution is internal to the black box and is unobservable (and therefore uncheckable) from outside the box. X/Open's branding process would be improved if it broke through the black box paradigm and was able to test the internal execution of the system. Again, these internal tests only apply when the architecture standard codifies the protocols used within the components.

5.2.3 Implementation

It is advantageous if the codification of an architecture standard does not restrict the system to be implemented in only one programming language but rather allows many programming languages to be used to implement the system. In fact, one of the X/Open DTP documents [100] describes a portion of the standard in both C [41, 53] and COBOL [97]. Unfortunately, the rest of the X/Open DTP documents [102, 99, 101, 98] are quite C specific. For example, since C does not have explicit constructs to model asynchronous communication,[1] the standard uses "quick" returning function calls to model issuing an event. Similiarly, the reactive nature of receiving events is performed by repeated polling the senders. These and other C–specific techniques limit the languages that could be used to implement systems compliant to the standard.

While it is advantageous to have specifications that can be implemented in and does not favor any programming language or style of programming, it is still important, as a first step, to accurately respecify *exactly* the same standard as described in the X/Open DTP documents. Such a respecification seeks to minize resistance to adoption of this form of representation of their standard. Thus, we have adopted a strategy of specification that would require as little change in the behaviorial model as necessary. This means that any system that conforms to our reference architecture would also conform to the specification described in the X/Open DTP documents. Ideally, future versions of the standard would be more general and allow implementations in different programming languages.

5.2.4 Why Rapide?

The following case studies were an experiment to test the Rapide language and tool suite. Rapide was designed to specify concurrent, time–sensitive protocols, and we

[1]By *asynchronous communication* we mean the issuing of an event by one process to another process. It is an important semantic property that the issuance does not stop either process (like an Ada [94] rendevous). The sending process sends, and at some later time, the receiving process receives the event.

wanted to test its ability to model transaction processing protocols, especially distributed transaction processing (DTP) protocols. RAPIDE specifications have precise semantics that can represent synchronous and asynchronous communication — both event and functions call semantics. RAPIDE models explicitly encodes concurrent behavior, and thus can accurately model distributed behavior.

Since many DTP protocols are difficult to understand, they can benefit from RAPIDE's prototyping and animation capabilities. The RAPIDE team had recently developed a compiler and a set of analysis tools. We felt that execution or analysis of the X/Open specifications (a reference architecture) would be valuable approach to understand the standard. In particular, it was initially difficult for us to understand the X/Open documents' single thread of control constraints, and we felt there would be a great benefit in representing and animating these constraints.

More generally, we wanted to test RAPIDE's ability to encode constraints for the varied transaction processing properties. Different styles may be used in writing RAPIDE prototypes. Of particular interest to us was the constraint–oriented style that allows one to write several independent constraints that may be easily composed to constrain the complete system. Of course, once these constraints were written, we immediately wanted to test the feasibility of automatically checking them.

Finally, we wanted to develop a methodology for using the RAPIDE technology. A RAPIDE prototype can be developed through stepwise refinement. An architecture–oriented style might be adopted first, encoding the gross overall structure of the system without the specifics of the algorithms or data structures. A constraint–oriented style might be adopted next, to express the requirements as constraints without biasing the implementation. The reference architecture may then be refined and elaborated, perhaps transforming the style of specification into a state–oriented style. Once refined and elaborated, the reference architecture can be executed, animated, and tested to satisfy its constraints. This process facilitates experimentation of the effects of various changes to the reference architecture.

Once the reference architecture is developed, it might be used as a standard to test actual systems for conformance. In our view, designing with RAPIDE can be viewed as similar in nature to programming; the author of a RAPIDE reference architecture can get the same type of immediate feedback (and often more information) from the simulation than the author of a program gets from compiling and early testing [63].

5.3 Description of the Case Studies

We conducted four case studies that used RAPIDE to specify and test conformance to the X/Open DTP industry standard.The standard is specified by X/Open with English text, state–tables, and exampliary execution traces. We translated these specifications into RAPIDE reference architectures. Each reference architecture was reflective of particular study of the standard.

The first case study was conducted to capture the architectural and high–level transaction constraints found in the standard. This case study is described in Section 5.3.1, while Section 5.3.2 describes the second case study that was conducted to add typical behaviors to the components of the architecture, and to verify (or check) that the behaviors satisfied the system constraints. Specifically, the transaction initiation, and completion – via the two–phase commit protocol. The third case study (Section 5.3.3), was conducted to use the reference architecture of the second case study to test the conformance of a exempliary concrete system to the standard. Since the X/Open DTP standard is constantly under revision, we examined in the last case study the result of extending the standard with several isolation constraints and prototypical behaviors. This case study, described in Section 5.3.4, adds the property of isolation and the two–phase locking protocol into the standard.

5.3.1 Architecture Study

The first study emphasized the architectural aspects of the X/Open industry standard. By architecture, we mean the interfaces of the components of compliant systems, their communication paths and protocols. The standard is a software architecture that allows multiple application programs to share resources provided by multiple resource managers and allows their work to be coordinated into global transactions.

The components of the architecture, e.g., application programs (APs), resource managers (RMs), and transaction managers (TMs), may be combined in many architectures depending on which components are used and how they are connected.

A local instance of the standard is obtained by combining an AP and one or more RMs coordinated by a TM. Another DTP architecture may consist solely of TM code calling a set of RM libraries. Such an architecture is useful, for example, during transaction completion and recovery.

Method

The reference architecture is intended to supply significant semantic content that informs implementors about the kinds of properties and protocols compliant systems must have. Thus, it provides a reusable framework for a product family.

We used many features of the Rapide language. The component interfaces were divided into services, as per the X/Open documents. Each service includes type and function declarations. Those services include various formal constraints. Pattern macros were used to model threads of control. These pattern macros made extensive use of dependency relationships.

Results

The Rapide reference architecture for the X/Open DTP industry standard (see Appendix A) is about XXXX lines long. This includes many lines of formal constraints. The Rapide technology provides an extensive pattern language, but most of the patterns used were very simple ones.

We used the Rapide 1.0 Compiler to static–semantically check the architecture and constraints for various properties, e.g., for preservation of communication integrity.

Experience

Dynamic Architectures. Instances of the reference architecture may differ with respect to the number and kinds of components, especially the RMs, and the connections among them. This is especially true when a component can take on more than one role in a system. This occurs when a resource manager acts like an application program and calls another resource manager. RAPIDE's type system, pattern language, and architectural features are used to specify such a dynamic architecture.

Flags. The X/Open documents use C integers to represent flags rather than the enumeration types used in the RAPIDE reference architecture. Representing all of the flags as integer values has several problems. First is type checking; one can't static semantically distinguish between the flags used solely by one routine, and flags used by the other routines. Also, sets of flags have to be created by adding the flags' integer values together. This creates a limit (# of bits in a C integer) on the number of flags. Also, they had to explicitly define the constant TMNOFLAGS (0 integer value) to denote that there is no flag set.

The RAPIDE reference architecture represented the flags as enumeration values. It grouped the enumeration values into enumeration types as per their functional grouping in the standard. The RAPIDE compiler's static semantic checking assures the components that share a service must define the same enumeration types and values.

However, our reference architecture's approach does not explicitly define the bit–level data structure chosen to represent the flags. If there is an advantage to X/Open's approach, it is that their approach strictly specifies a bit–level data structure. They define every flag's bit length to be the size of a C integer.[2]

Naming Conventions. The X/Open documents commonly use naming conventions to declare interfaces, functions, and constants. Examples of these conventions include:

[2]Defining an equivalent data type to X/Open's flag can be done in RAPIDE with an interface type and a module generator for each constant value.

- Functions in the XA (service) interface that are prefixed with ax_ are the routines that allow an RM to call a TM, and correspondingly, all TMs must provide these routines. Similarly, xa_ prefixed functions are supplied by RMs and called by TMs.
- Negative values, which alway imply an error condition, returned by xa_ functions all begin with XAER_. Their non–negative return values all begin with XA_.
- Names of flags passed to XA routines begin with TM.

The naming conventions are desirable, because they facilitate understanding and (human) semantic checking. This is especially critical, since C does not have constructs that can automatically check these architectural constraints. However, the RAPIDE service construct provides additional semantics. 0) Interfaces are divided into named services and the service name is used to prefix constituents of the service where they are used. This is essentially equivalent to their prefix notation. 1) Services replaces pointers to functions improving the semantic checking that can be performed.

Inconsistency. Our attempt to capture the standard uncovered several inconsistencies or incompletenesses. One example is related to the setting the commit return point. The X/Open documents specify that the application program can modify when the tx_commit() function returns. It can return either when the two–phase commit protocol is completed, or when the decision to commit is logged but prior to completing the second phase. However, logging is not described in the standard.

5.3.2 Protocol Requirements Study

In the second study we modeled prototypical behaviors for the components of the standard. Since the standard concentrates on the coordination between the components and the TM is primarily responsible for the coordination, we coded a realistic implementation for the TM and toy implementations for the AP and RMs.

Method

We used the formal constraints of the architecture developed in the previous study and the state–tables and English descriptions found in the X/Open documents to build our prototypical behaviors for the AP, TM, and RM components. Since most of the constituents for the component interfaces are specified to be functions, they were implemented as functions. Since the current RAPIDE compiler does not implicitly generate the events that represent the call and return of the functions, our reference architecture explicitly declares and generates events for those activities. These explicitly declared and generated events were used by the Partial Order Browser (POB) to visualize the executions and by the constraint checker to verify the runtime behavior.

Results

The prototype implementations of the component behaviors were XXX pages. They were coded as RAPIDE behaviors. Quite a bit of the code dealt with storing and accessing the state of the component (similarly to state–tables of the X/Open documents). Also, the standard defines many failure conditions and a hierarchy of those conditions. These semantics were quite lengthy to express in the RAPIDE, and quite a bit of the behaviors' code handles the conversion of the failure conditions through the hierarchy.

The TM specifications and consequently our implementation are centered around the two–phase commit protocol. The two–phase commit protocol assures the transaction property of atomicity, an architectural constraint. The commit protocol is so named, because it is divided into two phases. The first phase of the protocol the TM polls the RMs to prepare to commit the transaction. A positive response to the poll guarantees that the RM will commit its part of the transaction if requested. A negative response, for any reason, indicates that part of the transaction cannot be committed. The second phase begins by collecting all of the responses to the poll and deciding whether to commit the transaction. If all of the RMs respond positively, the decision will be to commit, and if any of the RMs respond negatively, the decision will be to abort. Upon reaching the decision, the RMs are informed and they each commit or abort as per the decision.

Theoretical analysis of this protocol determined that all correct executions of the commit protocol imply not only the atomicity constraint but also a coordination constraint. The coordination is constrained such that a commit function call must never be causally independent from a prepare function return for the same transaction. As a positive result of this study, this constraint was easily captured by the constraint language and run time checked by the constraint checker. However, some of the other X/Open DTP constraints were very computational expensive to check and were outside the implemented subset of the RAPIDE 1.0 Compiler. Therefore, not all of the constraints of the reference architecture (presented in Appendix A) were checked during this case study.

Experience

Understandability. The X/Open documents give several exempliary execution traces that we interpreted as constraints for the proper sequences of system calls. The pattern language captured those constraints very well and in a manner that for the most part was easy to understand. The RAPIDE tool suite was able to simulate and animate those execution traces. The animation was very useful in validating the accuracy of prototypes.

Debugging with the extra information provided by the poset was quite easy. The dependency relationships found within the poset quickly answers a common question asked during debugging like "why did this activity occur?"

Functions vs. Events. The commit protocol allows for polling and informing the RMs to be done concurrently if the call and return of the activities is done asynchronously, i.e., are not implemented as functions. However, since the activities are implemented with functions, an ordering must be chosen, possibly nondeterministically. In either case, RAPIDE has constructs that can either constrain such executions or generate them. A nondeterministic ordering was generated by collecting the indices of the RMs as a set, and nondeterministically selecting a member of that set.

Scalability. The Rapide simulator was adequate for small executions but bogged down on large ones, especially large ones with the constraint checking. The constraint checking problems are two–fold. First, the current constraint checker was designed and implemented very quickly using very inefficient algorithms. The checker was designed more to test the constraint language than to actually check constraints. It performs exponentially worse as the program executes.

Second, the current constraint checker was designed to only check a small subset of the constraint language. This subset was a commonly occurring type of constraint for this study; the only implemented feature for constraint checking is the **never** constraint that constrains the execution to never have a particular pattern of behavior within it. The usefulness of this type of constraint is quite apparent as is efficient techniques to check them. However, the other types of constraints do not lend themselves to efficient checking algorithms.

Input and Output. An additional subset limitation of the Rapide 1.0 Compiler is input and output (I/O). Currently, the only run time I/O is the passing of the command line arguments to the program. While this limitation does not theoretically restrict the behaviors that can be dynamically selected at run time, it does create a considerable burden for every reference architecture with an extensive suite of scenarios.

It is quite time–consuming and tedious to encode every possible alternative as some sort of command line argument. However, for small suites of scenarios, command line arguments works quite well. We occasionally modified the code directly rather than encoding a command line argument. These modifications were quite simple to make, and the compile and execution times were fairly short.

Scenario Sizes. There are not many scenarios given in the X/Open DTP documents, and the ones that are there are more like execution fragments than complete scenarios. On the other hand, the Rapide scenarios were complete executions. As one would expect, the X/Open fragments are only about dozen lines of text, while the RAPIDE scenarios were fairly large execution histories. Of course, the extra information provide by the dependency relationships encoded in a RAPIDE execution is the cause of this difference.

5.3.3 Conformance Testing Study

In the third case study we tested a system for conformance to the reference architecture developed in the second case study. The exempliary system was a bank transaction processing system that accepts deposit, withdrawal, and transfer transactions. It is a good example domain, because much of the X/Open standard centers around the error conditions that occur during a transaction's execution, and banking transactions can exhibit many different failure modes, e.g., user abort, account overdrawal, and hardware failure.

Method

Determining whether an application conforms to a standard is a challenging undertaking, and one technique for doing so is *comparative validation.* Comparative validation checks the consistency between two or more different systems, e.g., our banking system and our reference architecture, by testing them with respect to each other under a particular set of circumstances. In effect, by having several algorithms that implement the same problem and by giving each algorithm the same input data, we can check whether their outputs are equivalent.

In RAPIDE, one or more systems (the domain) are provided with a stimulus (test data) and their behaviors produce an execution. This execution is mapped into the universe of the other system (the range), and the mapped execution is checked for equivalence against the range's constraints. Determing equivalence involves detecting differences between the domain executions and the range's execution model. Repeating the comparison under different stimuli increases the confidence that the systems conform to each other.

Current approaches for defining the relationships among several systems are limited to trace–based methodologies. A trace of events does not record enough information to detect all violations of transaction processing constraints, e.g., a violation of the two–phase commit protocol that can't be detected with a trace of events occurs when a TM's commit call that was made in response to the prepare–to–commit poll is based upon only some (not all) of its inputs. However RAPIDE's execution model is based upon partially ordered sets of events (posets), and it has an explicit construct used to defining the relationships among architectures, the **map**. A domain system or architecture conforms a range architecture if and only if the mapped execution of the domain architecture satisfies all of the constraints of the range architecture. Essentially, the poset produced by the domain architecture is interpreted via the map as if it were produced by the range architecture.

Results

The bank transaction processing system was written in approximately XXX lines of RAPIDE. We chose to implement our banking system with an architecture very closely related to the architecture described in Gray and Andreas Reuter's book [31] to test the complexity of writing maps. The map was XXX lines long, and its code was surprisingly simple to write.

Experience

Interoperability. We chose to implement the banking system, in RAPIDE, because any other language would require additional code to be written to generate the events. The RAPIDE project has plans to develop libraries of code to assist developers in generate the events in several languages.

Defining Conformance. An interesting question that always occurs when one is trying to test systems for conformance to a standard: "how is conformance with a standard defined?" For example, the data values transmitted between the components have to exactly match (in both type and value) the values defined in the standard or they can be mapped into the standard's value by the map. Specifically, the X/Open documents specify constants, return values, and flags with integer values. The advantage of X/Open's approach is that the standard exactly specifies the value (data value) transmitted to other components. Another approach, which was used in the RAPIDE reference architecture, is to use enumeration types (and values) to represent the constants, return values, and flags. This approach enables automated type checking to reduce errors, but requires a mappping to translate from the enumeration values to their corresponding integer values.

Fundamentally, a definition of conformance to a reference architecture can be based upon either the architectural topology, behavior, or a combination of the two. Behavior–based conformance involves checking that the computation produced by executing the system is equivalent to a computation producable by the standard. A standard maker may also want to define exactly how constituents are allocated to the components, or she may not care where they are allocated as long as they exist. Topology–based conformance involves checking that system's components, the constituents of the components, and how they are connected together are equivalent to the architecture of the standard. These two metrics, topology and behavior, are not mutually exclusive but represent a wide spectrum of architectural criteria from which conformance can be defined.

Checking Conformance. Topological conformance views the standard's specification as a roadmap for the system. This type of conformance ensures the functionality of the components is preserved by the system implementation. Continuing this philosophy, the topology is viewed as a bound on the possible system behavior and therefore bounds conformance.

Theoretically a static–semantic checking tool can be used to test topological conformance. This tool would examine the architectures of both the standard and the system and attempt develop a unique one–to–one mapping of the components and connections. If such a mapping was obtainable, then the system conforms. However if no such mapping could be found, the system would not conform.

Topological conformance can be also be checked while the system executes like behavioral conformance. Topological and behavioral conformance can be tested by extracting constraints from the standard and checking that the system's execution satisfies those constraints. This can be done in RAPIDE by mapping up the execution's events into the domain of the standard's constraints or by mapping the constraints of the standard down onto the domain of the system execution. In either case, a map from the standard to the system must be created.

Violation Detection. A pratical question for using the tools is what happens when a violation is detected. Specifically what kind of interface is provided by the tools to allow the system architect or implementor to pinpoint why the system doesn't conform. RAPIDE's constraint checker produces an event for each constraint violation that is dependent upon the mapped events participating in the violation of the constraint. Further study is needed to determine if a facility for mapping the mapped events in the range back to its triggering events in the domains.

5.3.4 Isolation Extension Study

Many of the X/Open standards are constantly under revision, and the DTP standard is no exception. In the last of our case studies, we examined the effects of changing the standard to include the property of isolation and prototype behaviors that satisfy the property. Specifically, we extended the reference architecture with several constraints that limit every transaction's execution to be isolated from the other concurrently executing transactions and with prototypical behaviors that implement the two–phase locking protocol that ensures isolation.

Method

We used **never** constraints to specify the three conditions that occur when isolation is violated: lost update, dirty read and unrepeatable read. These constraints are the three architectural constraints that limit the collective execution of the resource managers. The constraints monitor the read and write events generated by the resouce managers and check that the transactions don't exhibit the dependencies that are characteristic of the three conditions (and violate isolation). Therefore, the resource managers were modified to generate a read (and a write) event every time a data object is read (and written).

We also modified the system architecture by adding a specific type of resource manager, the lock manager, a behavior for the lock manager, and modified the behaviors of the other resource managers to use the lock manager — the resource managers were coded to use the two–phase locking protocol.

Tested the reference architecture by simulating the reference architecture as well as with the banking system.

Results

Performing this study required very little additional RAPIDE code to be written. The coding the constraints for isolation required only 8 lines, and only XX lines were needed to specify and implement the lock manager. The resource managers were only slightly modified; the implementation of isolation only required a few calls to lock and unlock the resources as per the two–phase locking protocol. Again these modifications were extensions to the X/Open standard and were not present in the X/Open documents.

Experience

Isolation Constraint. There are several ways to specify the property of isolation. Since many of these protocol specifications are, in theory, very expensive to check and are, more importantly, not implemented in the current compiler, we rewrote them into several **never** constraints that when taken collectively imply isolation. That presents an interesting question as to whether it is always possible to rewrite a constraint into an equivalent set of **never** constraints and is this set finite and countable. Also, can it be automatically generated from the original constraint.

These are very interesting theoretical questions, but from a more practical perspective, we have found that all of the constraints we wrote were able to be converted manually into a very small set of **never** constraints.

Lock Manager Behavior. The implementation of the lock manager exemplified a very powerful rapid prototyping construct in RAPIDE, the process generator. It creates a process for each object that can be locked. Each process repeatedly waits for a lock call, responds with a lock return, waits for a unlock call, and responds with an unlock return. RAPIDE provides convenient constructs to declare any number of processes that execute concurrently.

Chapter 6

Conclusions

We have argued in Chapter 1 that formal methods, prototyping, and architecture definition technologies can improve the state of practice in software system standardization. The rest of the chapters have presented and used RAPIDE to verify this hypothesis on a particular standard, the X/Open distributed transaction processing industry standard. In this chapter, we first look back and summarize the RAPIDE technology. We then succiently describe the original contributions found in this dissertation for formally specifying and developing prototypes of system standards. Finally, we look forward into the future directions that this research has opened up.

6.1 RAPIDE Summary

6.1.1 Architecture Definition Language

RAPIDE is an executable architecture definition language (ADL) and a suite of tools based on that language. That language has many features in common with other ADLs such as interfaces and connections. It also goes beyond current ADL's in providing several new features. In summary, RAPIDE provides:

Interfaces: Component interfaces in RAPIDE specify *both* what the components demand of their context and what they provide to it; interface features define a component's features for both synchronous and asynchronous communication, its reactive behavior to inuts, and (runtime checkable) constraints on its behavior. Interfaces in RAPIDE are types. Dynamic subtype substitution is supported to allow flexible rapid modification of architectures by replacing old components with new ones.

Connections: Architecture connections between the components; that is, rules that describe how components interact with each other in a particular architecture.

Constraints: Formal constraints on the architecture connections, expressing, for example, communication protocols between components. Architecture constraints are supported by dynamic conformance checking tools.

Mappings: Event pattern mappings between architectures; a powerful capability to define relationships between (possibly widely differing) architectures. Mappings are supported by runtime rules that map the behavior of one architecture (the domain of the map) into that of another architecture (the range of the map). Maps may be used to check conformance of the domain to the range architecture, e.g., checking conformance to reference architectures.

Hierarchy: Recursive component hierarchy, in which component structures are defined as architectures of constituent components.

Genericity: Genericity, so that architectures of product instances and product families can both be represented. This is key to achieve a reuse–based development process.

RAPIDE does these things in an unique way. The underlying semantic base of RAPIDE is the notion of an event–based model of the computation. The events abstract subcomputations that are of little interest in the architectural view. In any system, the events are not all independent; in RAPIDE the event–based model includes partial orders representing dependency among the events. Two kinds of dependency are emphasized in RAPIDE: causal and temporal.

6.1.2 Rapide Computation

Causal and temporal event histories generated by simulation of a Rapide architecture provide the most detailed event–based representation of distributed/concurrent behavior. An event history is a partially ordered set of events (or poset) and can explicitly show which events caused which other events and which events happened independently, during execution. Causal information, inconjunction with timing, is critical in the analysis of distributed system behavior, in the ability to accurately express constraints and in constructing sensible animations of a system's behavior. The advantages of causal event histories is well documented and illustrated in this disseration[1] and in the Rapide literature (see references).

Posets are the basis of the Rapide notations and tools. The Rapide language provides ways to specify what events a component (interface) can generate and consume (using Rapide "actions"), as well as what services it provides and requires. The language provides a way to define, in executable form, the behaviors of components by simple reactive rules that trigger on, and generate, patterns of events (including the dependency relationships among those events). The language defines the interconnection rules for components of an architecture in terms of patterns of events that are generated and observed by the components (using Rapide "basic" and "guarded basic" connections). Alternate architectural models of a single system can be related (or "mapped") in the Rapide language, by maps that define corresponding computation patterns in the separate architectures.

Architectures can be exercised by executing them using a set of use–cases or scenarios. The results of the executions are computation posets that reveal both causal and timing/performance properties of the architecture. These posets form the basis for causal and temporal analysis tools to determine properties of architectures.

This usage model motivates the Rapide tools. A compiler/runtime system provides the capability to simulate Rapide architectures. Runtime constraint checkers

[1] For example the coordination constrint as a prerequisite for correct execution of the two–phase commit protocol for atomicity in the X/Open DTP standard.

detect and report the occurrence of prohibited patterns of events, i.e., constraint violations, in the computation scenario. Poset browser provide sophisticated pattern–based filtering of the posets so that human intuition can be used to discover important patterns in an architecture's dynamics. These discoveries can then be recorded as constraints — derived requirements — in the architectures. Human understanding is further enhanced by poset–driven animators that depict the architecture diagramatically and animate the event occurrences.

6.1.3 X/Open DTP Case Studies

The Rapide technology has matured to a point where the tools can support "real world" case studies of architectural modelling. In particular, this dissertation is an example of applying Rapide to the architectural modelling of an industry standard for distributed transaction processing from X/Open Ltd. Other studies are in progress to demonstrate the effectiveness of architecture simulation techniques in realistic performance analysis obtained from prototypes of proposed systems.

Before conducting the case studies several questions were raised about the potential benefits of using Rapide:

- Is Rapide difficult to learn and use?
- Can Rapide adequately express the X/Open DTP standard?
- What are the benefits of using Rapide to represent such a standard?
- What methodologies and tools support the use of Rapide?

This section will discuss some of our findings in attempting to answer these questions.

Learning and Using Rapide

Rapide is a complex language, and the lack of language reference manuals and examples [2] have made it difficult to learn. Rapide requires a paradigm shift from

[2]The Rapide language reference manuals and a book of examples are expected to be released for public review in early 1996.

conventional programming languages. However, once this shift is made and fluency in using the appropriate tool is reached, RAPIDE becomes much easier to use.

Expressibility of RAPIDE

RAPIDE was able to express all of the transaction processing and architectural properties of interest to us. This included the individual component interfaces, their behaviors, and constraints, as well as the global properties of atomicity and isolation. However, many of these global properties were originally expressed as RAPIDE protocol constraints that are, in theory, very expensive to check and are, more importantly, not implemented in the current tool suite. Surprisingly, these complex constraints could be rewritten into several simpler constraints that are easier to check.

These rewrites were not difficult to perform because of the large body of related work [31] that had already discovered them. In general, enabling the RAPIDE compiler to automatically rewrite (or optimize) constraints is not trivial [21]. Automatic event pattern–based constraint optimization would require proof system for the constraint language.

Benefits of Using RAPIDE

The X/Open documents protray the standard in a very C programming language–specific manner. For example, they define a "resource manager switch" as a collection of pointers to functions and pass such a switch between a RM and TM to faciliate communication. Passing collections of pointers to functions is a very C–specific way to connect components together. A RAPIDE specification is more language independent, and a language independent specification enables the implementors the greatest freedom in their implementation approach. Greater freedom brings X/Open closer to their goal of specifying open systems.

Clearly, formalism makes some types of analysis possible, where a lack of formalism prevents it. Because of RAPIDE's formal computation model, we were able to show that a coordination requirement that is a logical consequence of the specifications and protocols X/Open defined. Also, RAPIDE's architectural language enables formal syntacic and static–semantic checking of a system's architecture.

An additional feature that faciliates design evolution is RAPIDE's ability to easily combine constraints. Constraints can be incrementally added at various locations in a reference architecture: in interfaces, architectures, modules and maps. In general, the RAPIDE technology enables systematic design and evolution of systems and system standards via reusable and composable reference architectures.

RAPIDE Tools

Since the RAPIDE computational model provides much more information than traditional programming languages, the tools to make use of this extra information are critically important. The first tool is the syntax and static–semantic checker. This tool provides an initial examination for architectural consistency and communication integrity. The code generator and run–time system are required to produce executions, and those executions are visualized with the partial order browser and the animator. The constraint checker is built into the code generator and run–time system. These tools provide a comprehensive suite that supports the entire system development process.

6.2 Original Contributions

Not all of the work described in this dissertation was performed entirely by the author. As discussed in Chapter 1, the RAPIDE technology is the product of approximately 50 man–years of effort, and this has made it difficult to specify individual contributions. In general, the author has participated in this effort from the beginning — from the language conceptualization, design, and an associated tool suite implementation. One specific contribution made by the author was the design and implementation of the pattern matching component of the compiler. Other specific contributions to which the author is primarily responsible include:

X/Open DTP Reference Architecture: A formalization of over 200 pages of X/Open documents into a simple 20 page RAPIDE reference architecture that is executable and testable. The reference architecture formally defines component interface types, behaviors, and architectures. It also formally defines constraints and protocols on the executions of the architectures. The approach formalizes many of the documents informal ideas in terms of RAPIDE's computational model — partial orders of event sets (posets).

DTP Reference Architectures: An environment for the specification and analysis of DTP standards. The analysis includes execution, animation, and testing. Several views of DTP are used to define additional concepts, e.g., isolation, not present in the X/Open DTP industry standard.

Note: This includes knowledge gained from representing DTP standards in a formalism like RAPIDE, i.e., several original constraints like the coordination constraint, as well as the influences this example has had on the development of RAPIDE.

Influences on X/Open standard: Dependency based partial orders of events allows the formal specification of: *i*) the concept of "thread of control," *ii*) the intended parallelism of the system execution, and *iii*) the property isolation, not in X/Open documents. These results haven't been presented to X/Open, but when they are, they are expected to have a significant impact.

Influences on RAPIDE: Added syntactic sugar to the constraint language, the **never** constraint. This form of constraints is a very useful restriction from an implementation perspective. This form is easier to implement and quite useful for specification.

Conformance Testing: An application of a methodology for testing applications for conformance to reference architectures that involves taking (or mapping) a pattern of behavior in one or more (domain) architectures and producing another pattern of behavior in the (range) architecture. The mapped behavior is checked against the constraints of the range architecture.

6.3 Future Research

Many of the shortcomings described in this dissertation are currently under investigation. These shortcomings serve as subjects for future work as well as additional projects that are identified in this section. The possible future work directions can be formulated along three lines: extensions to improve the RAPIDE tool suite and practical applications of formal methods. It is hoped that the experience of using the evolving RAPIDE technology to develop standards will result in the design of better standards as well as improved testing of conforming systems to those standards.

6.3.1 RAPIDE Improvements

Documentation and Teaching. It has been extremely difficult for new users to learn RAPIDE without access to manuals and a compiler. The RAPIDE language reference manuals and compiler are expected to be publically released in early 1996. Several additional case studies, including Sparc V9 Processor [90], DMSO High Level Architecture [60], and several security–based standards [1], are currently under investigation. From these experiences teaching methods will be better understood and new users should have a much easier time learning and building prototypes.

Optimize Compiler and Runtime System. The current RAPIDE compiler and runtime system have a lot of room for improvement. The Stanford Program Analysis and Verification Group are continuing to develop new algorithms to improve performance of the RAPIDE simulations. Among the choices for improvement, there is special interest in optimizing the pattern matching parts of the compiler and runtime system [50]. Another interesting project is to distribute the runtime system's (and therefore RAPIDE programs) among several processors to improve the performance of the simulations.

Porting RAPIDE Tool Suite. Currently the RAPIDE tool suite executes only upon Sun's Sparc 2 hardware running version 4.1 of Sun's UNIX operating system. There has been considerable interest in porting the tools to Intel Pentium and other hardware running Linux.

Constraint Checker. The current **never** constraint checker is scheduled to be rewritten by the end of 1996 to more efficiently check **never** constraints as well as other types of RAPIDE constraints. Note: the constraint language is currently being re–examined to clarify and polish its syntax and semantics.

6.3.2 Practical Applications of Formal Methods

RAPIDE Proof System. Once the refinement of the constraint language is finished, a proof system to automate the checking of a set of constraints for satisfiability would also be a tremendous benefit. Such a proof system would also be extremely useful for checking the RAPIDE constraint checker's performance optimizations.

Architectural Refinement. An interesting question is whether a tool can be built to guarantee the preservation of properties throughout architecture evolution similar to the methodology of Moriconi and Qian [73, 74].

Appendix A

X/Open DTP Reference Architecture

This appendix presents the Rapide reference architecture for the X/Open distributed transaction processing (DTP) industry standard. This reference architecture is a formalization of the standard that is executable and testable. It formally defines the component interfaces, behaviors, and architecture. The reference architecture also formally defines constraints and protocols on the execution of the architecture.

The X/Open DTP standard is a software architecture that allows multiple application programs to share resources provided by multiple resource managers, and allows their work to be coordinated into global transactions. The architecture comprises three kinds of software components:

- an application program (AP) that defines transaction boundaries and specifies the actions that constitute a transaction,
- resource managers (RMs, such as databases or file access systems) that provide access to shared resource, and
- a transaction manager (TM) that assigns identifiers to transactions, monitors their progress, and takes responsibility for transaction completion and for failure recovery.

These terms are formally defined in this appendix.

A.1 Types

A.1.1 Global Types

These types, for portability reasons, must be common among the APs, TMs and RMs.

Transaction Branch Identifier

The X/Open standard uses transaction branch identifiers (or xids) to associate individual operations with a global transaction. An xid is a string that identifies a global transaction and a specific transaction branch that is the set of operations between one TM and one RM for that global transaction. Each xid is generated by a TM and given to a RM. The RM may use this information to optimize its use of shared resources and locks.

The structure of xids is specified in the RAPIDE code below via the Xid_t interface. An xid contains a format identifier, two length fields and a data field. The data field comprises at most two contiguous components: a global transaction identifier (gtrid) and a branch qualifier (bqual). The gtrid_length field specifies the number of bytes that constitute gtrid, starting at the first byte of the data field (that is, at *data[0]*). The bqual_length field specifies the number of bytes that constitute bqual, starting at the first byte after gtrid of the data field (that is, at *data[gtrid_length]*). A value of -1 in FormatID means that the xid is null.

```
XIDDATASIZE    : Integer is 128;      -- size in bytes
MAXGTRIDSIZE   : Integer is  64;      -- maximum size in bytes of gtrid
MAXBQUALSIZE   : Integer is  64;      -- maximum size in bytes of bqual

type Xid_t is
  record
     formatID,                         -- format identifier
     gtrid_length,
     bqual_length : ref(Integer);
     data         : String;
```

```
axiom
   -- The value in formatID should be greater than or equal to -1.
      (formatID >= −1);

   -- A value of -1 in formatID means that the XID is null.
      (formatID  = −1) −> ( gtrid_length = 0 and bqual_length = 0 );

   -- A value greater than -1 in formatID means that the gtrid and bqual lengths
   -- have a reasonable value .
      (formatID  > −1)
           −> ( 1 <= gtrid_length and gtrid_length <= MAXGTRIDSIZE and
                1 <= bqual_length and bqual_length <= MAXBQUALSIZE );

   -- The length of the data must be properly bounded.
      data.Length() <= XIDDATASIZE;
      data.Length()  = gtrid_length + bqual_length;
end record; −− Xid_t;
```

Commentary: The X/Open documents discuss the passing of pointers to xids, since many of the functions pass them. These pointers are valid only for the duration of the call. If the xid is needed after it is returned from the call, a local copy must be made before returning. In the reference architecture, the passing of references to xids has the same restriction.

Thread of Control

A thread of control (or a *thread*) is an important concept in the X/Open DTP standard; it is the concept that associates RM work with the global transaction.

```
type Tid_t is integer;
```

```
type Operating_System is
  interface
    provides
      -- returns a new thread identifier
        Create_Thread : function() return Tid_t;
  end interface Operating_System;

module Make_OS() return Operating_System is
    Last_Tid   : var Tid_t := 0;
    function Create_Thread() return Tid_t is
      begin
        Last_Tid := $Last_Tid + 1;
        return $Last_Tid;
      end Create_Thread;
  end module Make_OS;

OS : Operating_System is Make_OS();
```

Commentary: The X/Open documents make explicit use of thread without defining exactly what a thread is. Instead they give multiple definitions, including:

- the entity, with all its context, that is currently in control of a processor, and
- an operating–system process.

The assumptions X/Open makes about threads include:

- a thread identifier is an implicit parameter of the communication between the X/Open components,
- "coupling" techniques exist by which sets of threads can be associated together.

We have modelled the thread type, Tid_t, as an integer generated by an operating system object, OS.

Causal Chains. One property of threads is that all events generated by a thread are causally ordered.

```
match ( (?t in Tid_t) ( (?e in Event(t : Tid_t)) ?e(?t) )^(→ *) )^(~ *);
```

Coordination Assumption. The thread concept is central to the TM's coordination of RMs. APs call RMs to request work, while TMs call RMs to delineate transaction branches. The way the RM knows that a given work request pertains to a given branch is that the AP and the TM both call it from the *same thread*. This protocol can be expressed in RAPIDE as the following constraint:

```
match ( (?t in Tid_t)
  ( AP.TX.tx_begin(?t) → TM.XA.xa_start(?t)^(→ *) → AP.AR.Request(?t)^(→ *)
    → (AP.TX.tx_commit(?t) or AP.TX.tx_rollback(?t))) )^(~ *);
```

A.2 Services

Interfaces for AP, TM and RMs are structured into services. The TX service (Figure A.1) is "shared" between the TM and the AP in the sense that their interfaces contain respectively provided and required services of type TX, and the architecture connects them. Similarly the AP shares an AR service with each of the RMs, and the TM shares an XA service with each of the RMs.

A.2.1 TX (Transaction Demarcation) Service

TX is the name of the service shared by the AP and the TM. The TX functions are provided by TMs and are called by APs. APs demarcate global transactions via the TX interface and perform recoverable operations via RMs' native interfaces.

```
type TX_Service is
  interface
    provides
      -- Begin a global transaction: Note begin is a keyword.
        function tx_begin(t:Tid_t) return TX_Return_Code;
      -- Close the AP's resource managers.
        function close(t:Tid_t) return TX_Return_Code;
      -- Commit a global transaction.
        function commit(t:Tid_t) return TX_Return_Code;
      -- Obtain current transaction information.
        function info(t:Tid_t; info:ref(TXInfo)) return TX_Return_Code;
      -- Open the AP's resource managers.
        function open(t:Tid_t) return TX_Return_Code;
      -- Roll back a global transaction.
        function rollback(t:Tid_t) return TX_Return_Code;
      -- Set return point of commit.
        function set_commit_return(t:Tid_t; when_return:Commit_Return)
            return TX_Return_Code;
      -- Select chaining mode.
        function set_transaction_control(t:Tid_t; control:Transaction_Control)
            return TX_Return_Code;
      -- Set transaction timeout value.
        function set_transaction_timeout(t:Tid_t; timeout:Transaction_Timeout)
            return TX_Return_Code;
end interface TX_Service;
```

Transaction Characteristics

The state of an application thread of control includes several characteristics. The AP specifies these by calling tx_set_*() functions.

The commit_return characteristic determines the stage in the commitment protocol at which the tx_commit() call returns to the AP.

```
type Commit_Return is
   enum
      TX_COMMIT_COMPLETED,           -- tx_commit() return when the two-phase
                                     -- commit procedure is completed.
      TX_COMMIT_DECISION_LOGGED      -- tx_commit() returns at the point when
                                     -- the decision to commit is logged but prior
                                     -- to completing the second phase.
   end enum; -- Commit_Return;
```

The transaction_control characteristic determines whether the completion of one transaction automatically begins a new transaction (called *chained* mode).

```
type Transaction_Control is
   enum
      TX_CHAINED,                 -- completion begins a new transaction.
      TX_UNCHAINED                -- completion does not begin a new transaction.
   end enum; -- Transaction_Control;
```

The transaction_timeout characteristic specifies the time period in which the transaction must complete before becoming susceptible to transaction timeout. The interval is expressed as a number of seconds.

```
type Transaction_Timeout is Integer;
```

Transaction Information

The TXInfo record is used to return information about the thread state, including the state of all characteristics, the thread's association, if any, to a global transaction, and transaction state information.

```
type TXInfo is
   record
      xid           : ref(Xid_t);
      when_return   : ref(Commit_Return);
      control       : ref(Transaction_Control);
      timeout       : ref(Transaction_Timeout);
      state         : ref(Transaction_State);
   end record; -- TXInfo;
```

Transaction State. The AP may call tx_info() to obtain information regarding the state of the transaction it is in; that is, to determine whether the transaction is alive, has timed–out (and been marked rollback–only), or has been marked rollback–only (for a reason other than transaction timeout).

```
type Transaction_State is
   enum
      TX_ACTIVE,
      TX_TIMEOUT_ROLLBACK_ONLY,
      TX_ROLLBACK_ONLY
   end enum; -- Transaction_State;
```

Return Codes

The values returned by the TX routines.

```
type TX_Return_Code is
   enum
      TX_NOT_SUPPORTED,     -- option not supported
      TX_OK,                -- normal execution
      TX_OUTSIDE,           -- application is in an RM local transaction
      TX_ROLLBACK,          -- transaction was rolled back
      TX_MIXED,             -- transaction was partially committed and partially
                            -- rolled back
```

```
    TX_HAZARD,               -- transaction may have been partially committed and
                             -- partially rolled back
    TX_PROTOCOL_ERROR,       -- routine invoked in an improper context
    TX_ERROR,                -- transient error
    TX_FAIL,                 -- fatal error
    TX_EINVAL,               -- invalid arguments were given
    TX_COMMITTED,            -- transaction has heuristically committed
    TX_NO_BEGIN,             -- transaction committed plus new transaction could
                             -- not be started
    TX_ROLLBACK_NO_BEGIN,    -- transaction rollback plus new transaction could not
                             -- be started
    TX_MIXED_NO_BEGIN,       -- mixed plus new transaction could not be started
    TX_HAZARD_NO_BEGIN,      -- hazard plus new transaction could not be started
    TX_COMMITTED_NO_BEGIN    -- heuristically committed plus new transaction could
                             -- not be started
  end enum; -- TX_Return_Code;
```

Commentary: The X/Open documents define these return codes as integer values, where errors are denoted by negative return values. Thus, the AP may regard non–negative return codes as denoting success, but these return codes may convey additional information. The Rapide code does not explicitly define the denotation, only that the AP and TM components must share the same denotation.

A.2.2 XA Service

XA is the name of the services shared by the TM and the RM. XA consists of two subservices, *ax* and *xa*. The XA subservices are provided by RMs and are called by TMs, while the AX subservices are provided by TMs and are called by RMs.

```
type XA_Service is
  interface
    service
      ax_sub : dual AX_Subservice;
      xa_sub :      XA_Subservice;
  end interface XA_Service;
```

XA Subservice

Each RM provides a XA_Subservice that gives the TM access to the RM's xa_ routines. This is called the RM Switch in the X/Open documents.

```
RMNAMESZ : Integer is 32;

type XA_Subservice is
  interface
    provides
      Name    : String;
      Flags   : ref(Set(RM_Flag));  -- options specific to the resource manager
      Version : Integer;
  axiom
    Name.length() <= RMNAMESZ;
    Version =   0;                  -- must be 0.
    provides
     -- Terminate the AP's use of an RM.
     function close(t:Tid_t; xa_info:Info_Type; rmid:Integer; flags:Set(XA_Flag))
        return XA_Return_Code;
     -- Tell the RM to commit a transaction branch.
     function commit(t:Tid_t; xid:ref(Xid_t); rmid:Integer; flags:Set(XA_Flag))
        return XA_Return_Code;
     -- Test an asynchronous xa_ operation for completion.
     function complete(t:Tid_t; handle, retval, rmid:Integer; flags:Set(XA_Flag))
        return XA_Return_Code;
     -- Dissociate the t from a transaction branch. Note: end is a keyword.
     function xa_end(t:Tid_t; xid:ref(Xid_t); rmid:Integer; flags:Set(XA_Flag))
        return XA_Return_Code;
```

```
    -- Permit the RM to discard its knowledge of a heuristically-completed trans-
    -- action branch.
     function forget(t:Tid_t; xid:ref(Xid_t); rmid:Integer; flags:Set(XA_Flag))
        return XA_Return_Code;
    -- Initialise an RM for use by an AP.
     function open(t:Tid_t; xa_info:Info_Type; rmid:Integer; flags:Set(XA_Flag))
        return XA_Return_Code;
    -- Ask the RM to prepare to commit a transaction branch.
     function prepare(t:Tid_t; xid:ref(Xid_t); rmid:Integer; flags:Set(XA_Flag))
        return XA_Return_Code;
    -- Get a list of Xids the RM has prepared or heuristically completed.
     function recover(t:Tid_t; xids:Set(Xid_t); count, rmid:Integer; flags:Set(XA_Flag))
        return XA_Return_Code;
    -- Get a list of Xids the RM has prepared or heuristically completed.
     function rollback(t:Tid_t; xid:ref(Xid_t); rmid:Integer; flags:Set(XA_Flag))
        return XA_Return_Code;
    -- Start or resume a transaction branch - associate an Xid with future work
    -- that the t requests of the RM. Note: Compiler bug with functions in services
    -- named start!
     function xa_start(t:Tid_t; xid:ref(Xid_t); rmid:Integer; flags:Set(XA_Flag))
        return XA_Return_Code;
  end interface XA_Subservice;
```

AX Subservice

The TM provides an AX_Subservice that gives the RMs access to the TM's ax_ routines. All TMs must provide these routines. These routines let a RM dynamically control its participation in a transaction branch.

```
type AX_Subservice is
  interface
    provides
    -- Register an RM with a TM.
     function reg(t:Tid_t; rmid:Integer; xid:ref(Xid_t); flags:Set(XA_Flag))
        return AX_Return_Code;
```

```
  -- Unregister an RM with a TM.
    function unreg(t:Tid_t; rmid:Integer; flags:Set(XA_Flag))
      return AX_Return_Code;
constraint
 -- The flags argument of reg() and unreg() is reserved for future use and must
 -- be set to TMNOFLAGS.
  never( ?t in Tid_t, ?rmid in Integer, ?xid in ref(Xid_t),
         ?flags in Set(XA_Flag), ?ret in AX_Return_Code)
    ( reg'call(?t,?rmid,?xid,?flags) or reg'return(?t,?rmid,?xid,?flags,?ret)
     or unreg'call(?t,?rmid,?flags) or unreg'return(?t,?rmid,?flags,?ret) )
    where ?flags.Cardinality() /= 0;
end interface AX_Subservice;
```

Flag Definitions

The XA Service uses the following flag definitions.

RM Switch. The flag definitions for the RM switch.

```
type RM_Flag is
  enum
/*  TMNOFLAGS, */      -- no other flag being used
    TMREGISTER,        -- resource manager dynamically registers
    TMNOMIGRATE,       -- resource manager does not support association migration
    TMUSEASYNC         -- resource manager supports asynchronous operations
  end enum; -- RM_Flag
```

The xa and ax Routines. The flag definitions for xa and ax Routines.

```
type XA_Flag is
  enum
/*  TMNOFLAGS, */  -- no other flag being used
    TMASYNC,       -- perform routines asynchronously
    TMONEPHASE,    -- caller is using one-phase commit optimization
```

```
    TMFAIL,          -- dissociates caller and marks transaction branch rollback-only
    TMNOWAIT,        -- return if blocking condition exists
    TMRESUME,        -- caller is resuming association with suspended transaction branch
    TMSUCCESS,       -- dissocate caller from transaction branch
    TMSUSPEND,       -- caller is suspending, not ending, association
    TMSTARTRSCAN,    -- start a recovery scan
    TMENDRSCAN,      -- end a recovery scan
    TMMULTIPLE,      -- wait for any asynchhronous operation
    TMJOIN,          -- caller is joining existing transaction branch
    TMMIGRATE        -- caller intends to perform migration
  end enum; -- XA_Flag
```

Return Codes

The ax Routines. The `ax` routines' return codes.

```
type AX_Return_Code is
  enum
    TM_JOIN,           -- caller is joining existing transaction branch
    TM_RESUME,         -- caller is resuming association with suspended trans-
                       -- action branch
    TM_OK,             -- normal execution
    TMER_TMERR,        -- an error occurred in the transaction manager
    TMER_INVAL,        -- invalid arguments were given
    TMER_PROTO         -- routine invoked in an improper context
  end enum; -- AX_Return_Code
```

The xa Routines. The `xa` routines' return codes.

```
type XA_Return_Code is
  enum
/*    XA_RBBASE, */      -- the inclusive lower bound of the rollback codes
      XA_RBROLLBACK,     -- the rollback was caused by an unspecified reason
      XA_RBCOMMFAIL,     -- the rollback was caused by a communication failure
      XA_RBDEADLOCK,     -- a deadlock was detected
```

```
        XA_RBINTEGRITY,     -- a condition that violates the integrity of the resources was de-
                            -- tected
        XA_RBOTHER,         -- the resource manager rolled back the transaction branch for a
                            -- reason not on this list
        XA_RBPROTO,         -- a protocol error occurred in the resource manager
        XA_RBTIMEOUT,       -- a transaction branch took too long
        XA_RBTRANSIENT,     -- may retry the transaction branch
/*      XA_RBEND, */        -- the inclusive upper bound of the rollback codes
        XA_NOMIGRATE,       -- resumption must occur where suspension occurred
        XA_HEURHAZ,         -- the transaction branch may have been heuristically completed
        XA_HEURCOM,         -- the transaction branch has been heuristically completed
        XA_HEURRB,          -- the transaction branch has been heuristically rolled back
        XA_HEURMIX,         -- the transaction branch has been heuristically committed and
                            -- rolled back
        XA_RETRY,           -- routine returned with no effect and may be reissued
        XA_RDONLY,          -- the transaction branch was read-only and has been committed
        XA_OK,              -- normal execution
        XAER_ASYNC,         -- asynchronous operation already outstanding
        XAER_RMERR,         -- a resource manager error occurred in the transaction branch
        XAER_NOTA,          -- the XID is not valid
        XAER_INVAL,         -- invalid arguments were given
        XAER_PROTO,         -- routine invoked in an improper context
        XAER_RMFAIL,        -- resource manager unavailable
        XAER_DUPID,         -- the XID already exists
        XAER_OUTSIDE        -- resource manager doing work outside global transaction
  end enum; -- XA_Return_Code
```

A.2.3 AR Service

The AR Service is the service shared by the AP and RMs. It gives the AP access to shared resources.

This service is an RM–defined application programming interface by which an AP operates on the RM's resource. Since each RM defines its own interface, there may be many native interfaces. The RM may offer a standard interface, such as SQL or ISAM, in which case the AP may be portable to other RMs that use the same interface. The RM may, on the other hand, offer a proprietary interface specific to its services.

```
type AR_Service is
 interface
  function Request(t : Tid_t);
 end interface AR_Service;
```

Commentary: Since this interface is not defined by the standard, we have used an exempliary interface.

A.3 Interfaces

A.3.1 Application Program Interface

The interface of an AP component includes two services: a native RM service(s) through which the AP communicates with the RMs and a TX (transaction demarcation) service through which the AP communicates with a TM.

```
type Application_Program(NumRMs : Integer) is
  interface
    service AR(1..NumRMs) : dual AR_Service;
            TX            : dual TX_Service;
  end interface Application_Program;
```

A.3.2 Transaction Manager Interface

The interface of a TM component includes NumRMs XA services, one for each RM the TM may call, and a TX service that the AP may call.

```
type Transaction_Manager(NumRMs : Integer) is
  interface
    service XAs(1..NumRMs) : dual XA_Service;
            TX             :      TX_Service;
  constraint
  -- When the TM calls an RM's open() routine several times, once for each RM instance, it
  -- must generate a different RM identifier for each call.
   never (?i in Integer,  ?t in Tid_t,  ?n in Info_Type,  ?f in Set(XA_Flag),
          ?i2 in Integer, ?t2 in Tid_t, ?n2 in Info_Type, ?f2 in Set(XA_Flag))
      ( XA_Service::xa_sub.open'call(?t,?n,?i,?f)
        ~ XA_Service::xa_sub.open'call(?t2,?n2,?i2,?f2))
      where ?i = ?i2;

  -- The Atomicity Property: A global transaction identifier (i.e., a process identifier) should
  -- never be the argument of a commit event for one RM and an abort event for another RM.
  -- As a consequence, any transaction is either committed by all RM's or none.
   never (?x in ref(ref(Xid_t)), ?i in Integer, ?t in Tid_t, ?f in Set(XA_Flag),
          ?i2 in Integer, ?t2 in Tid_t, ?f2 in Set(XA_Flag) )
      ( XA_Service::xa_sub.commit'call(?t,?x,?i,?f)
        ~ XA_Service::xa_sub.rollback'call(?t2,?x,?i2,?f2));

  -- The Coordination Constraint: All commit call events from the TM to the RMs must
  -- depend upon all the prepare returns from the RMs.
   never (?x in ref(ref(Xid_t)), ?i in Integer, ?t in Tid_t, ?f in Set(XA_Flag),
         ?i2 in Integer, ?t2 in Tid_t, ?f2 in Set(XA_Flag) )
      ( XA_Service::xa_sub.prepare'return(?t,?x,?i,?f)
        || XA_Service::xa_sub.commit'call(?t2,?x,?i2,?f2));
  end interface Transaction_Manager;
```

A.3.3 Resource Manager Interface

The interface of a RM component includes a native service for the AP and a XA service through which the RM communicates with a TM.

```
type Resource_Manager is
  interface
    service XA : XA_Service;
            AR : AR_Service;
  constraint
   -- The calling RM must have TMREGISTER set in the flags element of its xa switch; reg()
   -- and unreg will return failure, [TMER_TMERR], when issued by a RM that has not set
   -- TMREGISTER.
    match ( ?t in Tid_t, ?i in Integer, ?x in ref(Xid_t), ?f in Set(XA_Flag),
            ?r in AX_Return_Code)
      ((XA.ax_sub.reg'return(?t,?i,?x,?f,?r) or XA.ax_sub.unreg'return(?t,?i,?f,?r) )
      where (($(XA.xa_sub.Flags).Is_Member(TMREGISTER)) or (?r=TMER_TMERR)));
end interface Resource_Manager;
```

A.4 Behaviors

In order to make the above interfaces executable, we will give default behaviors and modules for the AP, TM, and RM components. These implementations are not intended to be realistic, just simple examples of the behaviors required to generate one possible correct execution of the standard.

A.4.1 Transaction Manager Behavior

```
type TX_State_Table is
   enum
      S_0,               -- Non-existent Transaction
      S_1,               -- Active
      S_2,               -- Idle
      S_3,               -- Prepared
      S_4,               -- Rollback Only
      S_5                -- Heuristically Completed
   end enum;
TX_State   : array[Tid_t] of ref(TX_State_Table)
               is (1..10, default is ref_to(TX_State_Table,S_0));
Opened     : array[Integer, Tid_t] of ref(Boolean)
               is (1..10, default is (1..10,default is ref_to(Boolean,False)));
Associated : array[Integer, Tid_t] of ref(Boolean)
               is (1..10, default is (1..10,default is ref_to(Boolean,False)));
T_Xid      : array[Tid_t] of ref(Xid_t)
               is (1..10, default is ref_to(Xid_t,new(Xid_t)));
T_Control  : array[Tid_t] of ref(Transaction_Control)
               is (1..10, default is ref_to(Transaction_Control, TX_UNCHAINED));
T_Timeout  : array[Tid_t] of ref(Transaction_Timeout)
               is (1..10, default is ref_to(Transaction_Timeout, 0));
T_State    : array[Tid_t] of ref(Transaction_State);
```

```
function None_Opened(t:Tid_t) return Boolean is
begin
   for i : Integer in 1 .. NumRMs do
           if ( $(Opened[i,t]) ) then return False; end if;
   end do;
   return True;
end function None_Opened;

function next_gtrid() return Xid_t is
   x : Xid_t;
begin
   x.formatID        := 0;
   x.gtrid_length := 1;
   x.bqual_length := 1;
   x.data := $next_gxid;
   next_gxid := $next_gxid + 10;
   return x;
end function next_gtrid;

function Xid(gtrid : Xid_t; rm : Integer) return ref(Xid_t) is
   i : ref(Integer);
   x : var Xid_t := new(Xid_t);
begin
   xid_call();
   i := $(gtrid.data) + rm;
   ($x).formatID := 0;
   ($x).gtrid_length := 0;
   ($x).bqual_length := 0;
   ($x).data := $i;
   xid_return();
   return x;
end function Xid;

function More_Severe(X, Y : TX_Return_Code) return TX_Return_Code is
```

```
begin
   case X of
         TX_FAIL => return X;
   xor   TX_MIXED =>
            if (Y=TX_FAIL) then return Y; else return X; end if;
   xor   TX_HAZARD =>
            if (Y=TX_FAIL or Y=TX_MIXED) then return Y;
            else return X; end if;
   xor   TX_ERROR, TX_OUTSIDE, TX_ROLLBACK, TX_COMMITTED =>
            if (Y=TX_FAIL or Y=TX_MIXED or Y=TX_HAZARD) then return Y;
            else return X; end if;
   default => return TX_OK;
   end case;
end function More_Severe;

function TX.tx_begin(t:Tid_t) return TX_Return_Code is
   rc : var XA_Return_Code;
   f  : Set(XA_Flag) is {};
begin
   if ($(TX_State[t])=S_0 or $(TX_State[t])=S_3 or $(TX_State[t])=S_4) then
      -- Caller is not open or already in transaction mode.
         return TX_PROTOCOL_ERROR;
   elsif ($(TX_State[t])=S_1) then
         T_Xid[t] := next_gtrid();
         for i : Integer in 1 .. NumRMs do
            if ($(Opened[i,t]) and not $(XAs(i).xa_sub.Flags).Is_Member(TMREGISTER)) then
                rc := XAs(i).xa_sub.xa_start(t, Xid($(T_Xid[t]),i),i,f);
                Associated[i,t] := True;
            end if;
         end do;
         if ($(TX_State[t]) = S_1) then TX_State[t] := S_3;
         else assert( $(TX_State[t]) = S_2 );
            TX_State[t] := S_4;
         end if;
         return TX_OK;
   end if;
end; -- function TX.tx_begin;
```

```
function TX.close(t:Tid_t) return TX_Return_Code is
  f  : Set(XA_Flag) is {};
  rc : var XA_Return_Code;
  info : Info_Type;
begin
  if ($(TX_State[t])=S_0 or $(TX_State[t])=S_1 or $(TX_State[t])=S_2) then
    -- AP is not part of an active global transaction.
      for i : Integer in 1 .. NumRMs do
        if ( $(Opened[i,t]) ) then
           rc := XAs(i).xa_sub.close(t,info,i,f);
           Opened[i,t] := False;
        end if;
      end do;
      TX_State[t] := S_0;
      return TX_OK;
  else -- AP is in transaction mode.
      assert( $(TX_State[t])=S_3 or $(TX_State[t])=S_4 );
      return TX_PROTOCOL_ERROR;
  end if;
end; -- function TX.close;

function TX.commit(t:Tid_t) return TX_Return_Code is
  f  : Set(XA_Flag) is {};
  rc : var XA_Return_Code;
  committable : var Boolean := True;
  txrc : var TX_Return_Code := TX_OK;
begin
  if ($(TX_State[t])=S_0 or $(TX_State[t])=S_1 or $(TX_State[t])=S_2) then
    return TX_PROTOCOL_ERROR;
  else
    for i : Integer in 1 .. NumRMs do
      if $(Associated[i,t]) then
        rc := XAs(i).xa_sub.xa_end(t,Xid($(T_Xid[t]),i),i,f);
        rc := XAs(i).xa_sub.prepare(t,Xid($(T_Xid[t]),i),i,f);
        case $rc of
             XA_OK => txrc := More_Severe($txrc,TX_OK);
```

```
            xor  XA_RDONLY => txrc := More_Severe($txrc,TX_OK);
--          xor  XA_RBBASE .. XA_RBEND =>
            xor  XA_RBROLLBACK, XA_RBCOMMFAIL, XA_RBDEADLOCK,
                 XA_RBINTEGRITY, XA_RBOTHER, XA_RBPROTO,
                 XA_RBTIMEOUT, XA_RBTRANSIENT =>
                   txrc := More_Severe($txrc,TX_ROLLBACK);
                   committable := False;
            xor  XAER_NOTA =>
                   committable := False;
                   txrc := More_Severe($txrc,TX_ROLLBACK);
            xor  XAER_RMERR =>
                   committable := False;
                   txrc := More_Severe($txrc,TX_ROLLBACK);
            xor  XAER_RMFAIL =>
                   committable := False;
                   txrc := More_Severe($txrc,TX_FAIL);
            xor  XAER_INVAL =>
                   committable := False;
                   txrc := More_Severe($txrc,TX_FAIL);
            xor  XAER_PROTO  =>
                   committable := False;
                   txrc := More_Severe($txrc,TX_ROLLBACK);
            end case;
          end if;
        end do;
        for i : Integer in 1 .. NumRMs do
          if ( $committable ) then
            rc := XAs(i).xa_sub.commit(t,Xid($(T_Xid[t]),i),i,f);
            case $rc of
                 XAER_RMFAIL, XAER_INVAL, XAER_PROTO, XAER_NOTA =>
                   txrc := More_Severe($txrc,TX_FAIL);
            xor  XA_HEURMIX => txrc := More_Severe($txrc,TX_MIXED);
            xor  XA_HEURHAZ => txrc := More_Severe($txrc,TX_HAZARD);
            xor  XA_RETRY => txrc := More_Severe($txrc,TX_ERROR);
            xor  XA_OK, XA_HEURCOM =>
                   if (i /= 1 and $txrc /= TX_OK) then
                     txrc := More_Severe($txrc,TX_MIXED);
```

```
                else txrc := More_Severe($txrc,TX_OK);
                end if;
        xor  XA_HEURRB, XAER_RMERR =>
                if (i /= 1 and $txrc = TX_OK) then
                  txrc := More_Severe($txrc,TX_MIXED);
                else txrc := More_Severe($txrc,TX_ROLLBACK);
                end if;
        end case;
      else
        rc := XAs(i).xa_sub.rollback(t,Xid($(T_Xid[t]),i),i,f);
        case $rc of
             XAER_RMFAIL, XAER_INVAL, XAER_PROTO =>
                txrc := More_Severe($txrc,TX_FAIL);
        xor  XA_HEURMIX => txrc := More_Severe($txrc,TX_MIXED);
        xor  XA_HEURHAZ => txrc := More_Severe($txrc,TX_HAZARD);
        xor  XA_RETRY => txrc := More_Severe($txrc,TX_ERROR);
        xor  XA_HEURCOM =>
                if (i /= 1 and $txrc /= TX_COMMITTED) then
                  txrc := More_Severe($txrc,TX_MIXED);
                else txrc := More_Severe($txrc,TX_COMMITTED);
                end if;
        xor  XA_HEURRB, XAER_RMERR =>
                if (i /= 1 and $txrc = TX_COMMITTED) then
                  txrc := More_Severe($txrc,TX_MIXED);
        else txrc := More_Severe($txrc,TX_OK);
                end if;
        end case;
      end if;
    end do;
    if ($(TX_State[t]) = S_3) then
      TX_State[t] := S_1;
      return $txrc;
```

```
        else -- If the AP has selected chained mode, then begin a new global transaction.
          T_Xid[t] := next_gtrid();
          for i : Integer in 1 .. NumRMs do
            if ($(Opened[i,t]) and not $(XAs(i).xa_sub.Flags).Is_Member(TMREGISTER)) then
                rc := XAs(i).xa_sub.xa_start(t, Xid($(T_Xid[t]),i),i,f);
                Associated[i,t] := True;
            end if;
          end do;
          TX_State[t] := S_2;
          return $txrc;
        end if;
      end if;
    end; -- function TX.commit;

    function TX.open(t:Tid_t) return TX_Return_Code is
      f  : Set(XA_Flag) is {};
      rc : var XA_Return_Code;
      some_opened : var Boolean := False;
      info : Info_Type;
      txrc : var TX_Return_Code := TX_OK;
    begin
      if ( $(TX_State[t]) = S_0 ) then
        for i : Integer in 1 .. NumRMs do
          rc := XAs(i).xa_sub.open(t,info,i,f);
          -- RMs that are not open return RM-specific errors.
          case $rc of
              XA_OK =>
                Opened[i,t] := True;
                txrc := More_Severe($txrc,TX_OK);
          xor XAER_RMERR =>
                Opened[i,t] := False;
                txrc := More_Severe($txrc,TX_ERROR);
          xor XAER_INVAL =>
                Opened[i,t] := False;
                txrc := More_Severe($txrc,TX_FAIL);
```

```
        xor   XAER_PROTO =>
                Opened[i,t] := False;
                txrc := More_Severe($txrc,TX_FAIL);
        end case;
      end do;
      if ( None_Opened(t) ) then
        -- All RMs are closed.
        return TX_ERROR;
      else    -- At least one RM has been opened.
        TX_State[t] := S_1;
        return TX_OK;
      end if;
    end if;
    return TX_OK;
  end; -- function TX.open;

  function TX.rollback(t:Tid_t) return TX_Return_Code is
    f  : Set(XA_Flag) is {};
    rc : var XA_Return_Code;
  begin
    if ($(TX_State[t])=S_0 or $(TX_State[t])=S_1 or $(TX_State[t])=S_2) then
      return TX_PROTOCOL_ERROR;
    elsif ($(TX_State[t])=S_3) then
      for i : Integer in 1 .. NumRMs do
        rc := XAs(i).xa_sub.xa_end(t,Xid($(T_Xid[t]),i),i,f);
        rc := XAs(i).xa_sub.rollback(t,Xid($(T_Xid[t]),i),i,f);
      end do;
      TX_State[t] := S_1;
      return TX_OK;
    else assert( $(TX_State[t]) = S_4 );
      TX_State[t] := S_4;
      return TX_PROTOCOL_ERROR;
    end if;
  end; -- function TX.rollback;

  function TX.set_commit_return(t:Tid_t; when_return:Commit_Return)
    return TX_Return_Code is
```

```
begin
   case when_return of
         TX_COMMIT_DECISION_LOGGED => return TX_NOT_SUPPORTED;
   xor   TX_COMMIT_COMPLETED => return TX_OK;
   default => return TX_EINVAL;
   end case;
end; -- function TX.set_commit_return;

function TX.set_transaction_control(t:Tid_t; control:Transaction_Control)
   return TX_Return_Code is
begin
   if ($(TX_State[t]) = S_0) then return TX_PROTOCOL_ERROR; end if;
   case control of
         TX_UNCHAINED =>
            T_Control[t] := control;
            if ($(TX_State[t])=S_1 or $(TX_State[t])=S_2) then
               TX_State[t] := S_1;
            else assert($(TX_State[t])=S_3 or $(TX_State[t])=S_4);
               TX_State[t] := S_3;
            end if;
            return TX_OK;
   xor   TX_CHAINED =>
            T_Control[t] := control;
            if ($(TX_State[t])=S_1 or $(TX_State[t])=S_2) then
               TX_State[t] := S_2;
            else assert($(TX_State[t])=S_3 or $(TX_State[t])=S_4);
               TX_State[t] := S_4;
            end if;
            return TX_OK;
         default => return TX_EINVAL;
   end case;
end; -- function TX.set_transaction_control;

function TX.set_transaction_timeout(t:Tid_t; timeout:Transaction_Timeout)
   return TX_Return_Code is
```

```
begin
  if (timeout < 0) then return TX_EINVAL; end if;
  if ($(TX_State[t])=S_0) then return TX_PROTOCOL_ERROR; end if;
  T_Timeout[t] := timeout;
  return TX_OK;
end; -- function TX.set_transaction_timeout;

function TX.info(t:Tid_t; info:ref(TXInfo))
  return TX_Return_Code is
begin
  if ($(TX_State[t])=S_0) then return TX_PROTOCOL_ERROR; end if;
  if (not info.Is_Nil()) then
    if ($(TX_State[t])=S_1 or $(TX_State[t])=S_2) then
      $($info.xid).formatID := -1;
    else
      ($info).xid := $(T_Xid[t]);
      ($info).control := $(T_Control[t]);
      ($info).timeout := $(T_Timeout[t]);
      ($info).state := $(T_State[t]);
    end if;
  end if;
  if ($(TX_State[t])=S_3 or $(TX_State[t])=S_4) then
    -- Caller is in transaction mode.
    return TX_NOT_SUPPORTED;
  else
    -- Caller is not in transaction mode.
    return TX_OK;
  end if;
end; -- function TX.info;

function reg(t:Tid_t; rmid:Integer; x:ref(Xid_t); flags:Set(XA_Flag))
  return AX_Return_Code is
```

```
  begin
    Associated[rmid,t] := True;
    return TM_OK;
  end; -- function reg;

  function unreg(t:Tid_t; rmid:Integer; flags:Set(XA_Flag))
    return AX_Return_Code is
  begin
    Associated[rmid,t] := False;
    return TM_OK;
  end; -- function unreg;
connect
  for i : Integer in 1..NumRMs generate
    XA(i).reg to AX.reg;
    XA(i).unreg to AX.unreg;
  end generate;
```

A.4.2 Resource Manager Behavior

```
function XA.xa_sub.open(t:Tid_t; xa_info:Info_Type; rmid:Integer; flags:Set(XA_Flag))
 return XA_Return_Code is
begin
 return XA_OK;
end; -- function XA.xa_sub.open;

function XA.xa_sub.xa_start(t:Tid_t; xid:ref(ref(Xid_t)); rmid:Integer; flags:Set(XA_Flag))
 return XA_Return_Code is
begin
 return XA_OK;
end; -- function XA.xa_sub.xa_start;

function XA.xa_sub.xa_end(t:Tid_t; xid:ref(ref(Xid_t)); rmid:Integer; flags:Set(XA_Flag))
 return XA_Return_Code is
begin
 return XA_OK;
end; -- function XA.xa_sub.xa_end;

function XA.xa_sub.prepare(t:Tid_t; xid:ref(ref(Xid_t)); rmid:Integer; flags:Set(XA_Flag))
 return XA_Return_Code is
begin
 return XA_OK;
end; -- function XA.xa_sub.prepare;

function XA.xa_sub.commit(t:Tid_t; xid:ref(ref(Xid_t)); rmid:Integer; flags:Set(XA_Flag))
 return XA_Return_Code is
begin
 return XA_OK;
end; -- function XA.xa_sub.commit;
```

```
function XA.xa_sub.rollback(t:Tid_t; xid:ref(ref(Xid_t)); rmid:Integer; flags:Set(XA_Flag))
 return XA_Return_Code is
begin
 return XA_OK;
end; -- function XA.xa_sub.rollback;

function XA.xa_sub.close(t:Tid_t; xa_info:Info_Type; rmid:Integer; flags:Set(XA_Flag))
 return XA_Return_Code is
begin
 return XA_OK;
end; -- function XA.xa_sub.close;

function XA.xa_sub.recover(t:Tid_t; xids:Set(Xid_t); count, rmid:Integer; flags:Set(XA_Flag))
 return XA_Return_Code is
begin
 return XA_OK;
end; -- function XA.xa_sub.recover;

function XA.xa_sub.forget(t:Tid_t; xid:Xid_t; rmid:Integer; flags:Set(XA_Flag))
 return XA_Return_Code is
begin
 return XA_OK;
end; -- function XA.xa_sub.forget;

function XA.xa_sub.complete(t:Tid_t; handle, retval, rmid:Integer; flags:Set(XA_Flag))
 return XA_Return_Code is
begin
 return XA_OK;
end; -- function XA.xa_sub.complete;

function AR.Request(t:Tid_t) is
 xid : ref(Xid_t); f : Set(RM_Flag) is {};
begin
  XA.ax_sub.reg(t,$this_rmid,xid,f);
end; -- function AR.Request;
```

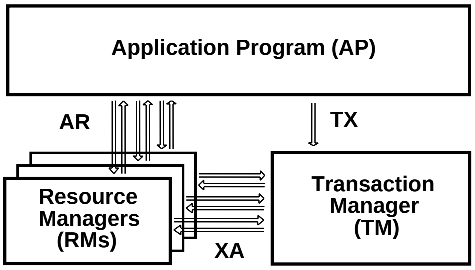

Figure A.1: Local Instance Architecture

A.4.3 Application Program Module Generator

```
module Simple_Application(NumRMs : Integer)
   return Application_Program(NumRMs) is
      thread : Tid_t is OS.Create_Thread();
      txrc : var TX_Return_Code;
   parallel
      txrc := TX.open(thread);
      txrc := TX.tx_begin(thread);
      for i : Integer in 1 .. NumRMs do
         AR(i).Request(thread);
      end do;
      txrc := TX.commit(thread);
      txrc := TX.close(thread);
   end module Simple_Application;
```

A.5 Architectures

A particular instance of the X/Open architecture [1] consists of one application program (AP), one transaction manager (TM), and one or more resource managers (RMs) connected together as shown in Figure A.1. The boxes indicate the generic component interfaces, and the lines indicate the communication between them. The

[1] X/Open calls this architecture a local instance architecture.

RAPIDE code for this architecture is:

```
architecture Local_Instance_Architecture(NumRMs : Integer)
  for DTP
  is
    AP   : Application_Program(NumRMs)
              is Simple_Application(NumRMs);
    RMs : array[Integer] of Resource_Manager
              is (1 .. NumRMs, default is new(Resource_Manager));
              --| RMs.length() = NumRMs;
    TM   : Transaction_Manager(NumRMs);
  connect
    AP.TX to TM.TX;
    for i : Integer in 1 .. NumRMs generate
      RMs[i].XA to TM.XAs(i);
      TM.XAs(i) to RMs[i].XA;
      AP.AR(i)   to RMs[i].AR;
    end generate;
  constraint
    -- Atomicity
      never (?thread, ?thread2 in PID_t; ?xid in ref(ref(Xid_t)),
               ?rmid, ?rmid2 in Integer; ?f, ?f2 in Set(XA_Flag))
        ( Resource_Manager::XA.xa_commit'return(?thread,?xid,?rmid,?f,xa_ok)
          ~ Resource_Manager::XA.xa_rollback'return(?thread2,?xid,?rmid2,?f2,xa_ok));

    -- Coordination
      never (?thread, ?thread2 in PID_t; ?xid in ref(ref(Xid_t)),
               ?rmid1, ?rmid2 in Integer, ?f, ?f2 in Set(XA_Flag))
        ( Resource_Manager::XA.xa_prepare'return(?thread,?xid,?rmid1,?f,xa_ok)
          || Resource_Manager::XA.xa_commit'call(?thread2,?xid,?rmid2,?f2));
  end architecture Local_Instance_Architecture;
```

Appendix B

Bank System

This chapter presents an executable application system that we will test for conformance with the X/Open DTP reference architecture presented in Chapter A. The domain of this application is banking. A bank transaction processing system is a good example, because much of the X/Open DTP standard centers around error conditions that occur during a transaction's execution, and banking transactions can exhibit many different failure modes, e.g., user abort, account overdrawal, and hardware failure.

The architecture of the banking transaction processing system is very similar to X/Open's. It has an application program, a transaction manager, and a set of resource managers. One of the resource managers is a lock manager that the other resource managers use to serialize their use of the resources.

B.1 Services

B.1.1 Transaction Identifier

```
type Xid is Integer;
```

B.1.2 Application Program to Resource Manager Service

```
type AP_R is
  interface
    action
      in    Request(x : Xid);
      in    Debit(x : Xid; i : Integer);
      out   Results(i : Integer; x : Xid; b : Boolean);
end interface AP_R;
```

B.1.3 Application Program to Transaction Manager Service

```
type aptm_return_code is enum aptm_ok, aptm_error end enum;

type AP_TM is
  interface
    action
      in    open_call();
      out   open_retn(rc : aptm_return_code);
      in    close_call();
      out   close_retn(rc : aptm_return_code);
      in    begin_call();
      out   begin_retn(x : Xid; rc : aptm_return_code);
      in    commit_call(x : Xid);
      out   commit_retn(x : Xid; rc : aptm_return_code);
      in    rollback_call(x : Xid);
      out   rollback_retn(x : Xid; rc : aptm_return_code);
end interface AP_TM;
```

B.1.4 Transaction Manager to Resource Manager Service

```
type tmr_return_code is enum tmr_ok, tmr_error end enum;

type TM_R is
  interface
    action
      in   register_call();
      out  register_retn(rm : Integer; b : Boolean);
      in   reg_retn(x : Xid);
      out  reg_call(rm : Integer; x : Xid);
      in   open_call();
      out  open_retn(rm : Integer; rc : tmr_return_code);
      in   close_call();
      out  close_retn(rc : tmr_return_code);
      in   start_call(x : Xid);
      out  start_retn(rm : Integer; x : Xid; rc : tmr_return_code);
      in   end_call(x : Xid);
      out  end_retn(x : Xid; rc : tmr_return_code);
      in   prepare_call(x : Xid);
      out  prepare_retn(rm : Integer; x : Xid; rc : tmr_return_code);
      in   commit_call(x : Xid);
      out  commit_retn(rm : Integer; x : Xid; rc : tmr_return_code);
      in   rollback_call(x : Xid);
      out  rollback_retn(rm : Integer; x : Xid; rc : tmr_return_code);
end interface TM_R;
```

B.1.5 Resource Manager to Lock Manager Service

```
type Lock_Name is Integer;

type Lock_Class is enum
  LOCK_INSTANT, LOCK_SHORT, LOCK_MEDIUM,
  LOCK_LONG, LOCK_VERY_LONG
end enum;
```

```
type Lock_Reply is enum
   LOCK_OK, LOCK_TIMEOUT, LOCK_DEADLOCK, LOCK_NOT_LOCKED
end enum;

type Lock_Mode is enum
   LOCK_FREE, LOCK_S, LOCK_X, LOCK_U, LOCK_IS,
   LOCK_IX, LOCK_SIX, LOCK_WAIT
end enum;

type R_Lock is
   interface
      action
         in    lock_call( name : Lock_Name; xid : Xid; mode : Lock_Mode;
                          class : Lock_Class; timeout : Integer );
         out   lock_retn( name : Lock_Name; xid : Xid; mode : Lock_Mode;
                          class : Lock_Class; timeout : Integer;
                          reply : Lock_Reply );
         in    unlock_call( name : Lock_Name; xid : Xid );
         out   unlock_retn( name : Lock_Name; xid : Xid; reply : Lock_reply );
      provides
         unlock_class : function( class : Lock_class; all_le : Boolean; rid : Integer )
                          return Lock_reply;
      requires
         lockhash : function( name : Lock_name ) return Integer;
end interface R_Lock;
```

B.2 Components

B.2.1 Application Program

Application Program Interface

```
type Application_Program(NumRsrcs : Integer) is
  interface
    action
      in Ready();
    service
      TMs : dual AP_TM;
      Rsrcs(1 .. NumRsrcs) : dual AP_R;
end interface Application_Program;
```

Application Program Module Generator

```
module Simple_Program(NumRsrcs, NumTrans : Integer)
  return Application_Program(NumRsrcs)
  is
    action Debit(src, val : Integer);
    action Executed();
    trans_count : var Integer := 0;
  parallel
    when Ready do
      TMs.open_call();
      await TMs.open_retn;
      Debit(1,1);
      await Executed;
    end when;
```

```
  ||
    when Debit do
      TMs.begin_call();
    end when;
  ||
    when (?src in Integer, ?val in Integer, ?x in Xid, ?rc in aptm_return_code )
          Debit(?src, ?val) -> TMs.begin_retn(?x,?rc)
    do
      if (?rc = aptm_ok) then Rsrcs(?src).Debit(?x, ?val); end if;
    end when;
  ||
    when (?src in Integer, ?dst in Integer, ?val in Integer, ?x in Xid, ?rc in aptm_return_code)
          ( Debit(?src, ?val) -> TMs.begin_retn(?x,?rc) -> Rsrcs(?src).results(?src,?x) )
    do
      TMs.commit_call(?x);
    end when;
  ||
    when (?x in Xid, ?rc in aptm_return_code) TMs.commit_retn(?x,?rc) do
      Executed();
    end when;
end module Simple_Program;
```

B.2.2 Transaction Manager Interface and Behavior

```
type Transaction_Manager_tc(NumRsrcs, NumTrans : Integer) is
  interface
    service
      AP : AP_TM;
      LM : dual TM_R;
      Rsrcs(1. .NumRsrcs) : dual TM_R;
```

```
behavior
   x : var Xid := 0;
   last_xid : var Xid := 0;
   function New_Xid_t() return Xid is
   begin
      last_xid := $last_xid + 1;
      return $last_xid;
   end function New_Xid_t;

   NumStaticRsrcs : var Integer := NumRsrcs;
   Dynamic_Rsrc : array[Integer] of ref(Boolean)
         is (1..NumRsrcs, ref_to(Boolean,False));
   function NumDynamicRsrcs() return Integer is
      num : var Integer := 0;
   begin
      for i : Integer in 1..NumRsrcs do
         if ( $(Dynamic_Rsrc[i]) ) then num := $num + 1; end if;
      end do;
      return $num;
   end function NumDynamicRsrcs;

   Reg : discrete_array(Integer, discrete_array(Integer,ref(Boolean)))
                  is (1..NumTrans, default is (0..NumRsrcs,
                                             default is ref_to(Boolean, False)));
begin
   AP.open_call =>
      LM.open_call();
      (!i in 1..NumRsrcs by ||) Rsrcs(!i).open_call();;

   (?j in Integer) Rsrcs(?j).open_retn(?j,tmr_ok) =>
      Rsrcs(?j).register_call();;

   ((?j in Integer) Rsrcs(?j).register_retn(?j))^(~ NumRsrcs) =>
      AP.open_retn(aptm_ok);;
```

```
(?m in Integer, ?n in Integer) ( ((?j in Integer,?rc in tmr_return_code)
     Rsrcs(?j).open_retn(?j,?rc) where ?rc/=tmr_ok)^(~ ?n)
     ~ ((?j in Integer) Rsrcs(?j).open_retn(?j,tmr_ok))^(~ ?m)
     ~ ((?j in Integer) Rsrcs(?j).register_retn(?j))^(~ ?m))
     where ?n + ?m = NumRsrcs
=>
   AP.open_retn(aptm_ok);;

(?j in Integer, ?b in Boolean) Rsrcs(?j).register_retn(?j,?b) =>
   Dynamic_Rsrc[?j] := ?b;;

AP.begin_call =>
   x := New_Xid_t();
   for i : Integer in 1 .. NumRsrcs do
      if not ( $(Dynamic_Rsrc[i]) ) then
         Reg[$x,i] := true;
         Rsrcs(i).start_call($x);
      else
         Reg[$x,i] := false;
      end if;
   end do;
   NumStaticRsrcs := NumRsrcs - NumDynamicRsrcs();
   if ( $NumStaticRsrcs = 0 ) then AP.begin_retn($x,aptm_ok); end if;;

(?x in Xid)((?j in Integer) Rsrcs(?j).start_retn(?j,?x,tmr_ok))^(~ $NumStaticRsrcs)
where $NumStaticRsrcs > 0
=>
   AP.begin_retn(?x,aptm_ok);;

(?j in Integer, ?x in Xid) Rsrcs(?j).Reg_call(?j,?x) =>
   Reg[?x,?j] := true;
   Rsrcs(?j).Reg_retn(?x);;
```

```
(?x in Xid) AP.Commit_call(?x) =>
   for i : Integer in 1 .. NumRsrcs do
      Rsrcs(i).prepare_call(?x);
   end do;

(?x in Xid)((?j in Integer) Rsrcs(?j).prepare_retn(?j,?x,tmr_ok))^(~ NumRsrcs) =>
   for i:Integer in 1 .. NumRsrcs do
      Rsrcs(i).commit_call(?x);
   end do;

(?x in Xid)((?j in Integer) Rsrcs(?j).prepare_retn(?j,?x,tmr_error) ~
((?j in Integer, ?c in tmr_return_code)
   Rsrcs(?j).prepare_retn(?j,?x,?c))^(~ NumRsrcs-1))
=>
   for i:Integer in 1 .. NumRsrcs do
      Rsrcs(i).rollback_call(?x);
   end do;

(?x in Xid)((?j in Integer) Rsrcs(?j).commit_retn(?j,?x,tmr_ok))^(~ NumRsrcs) =>
   AP.commit_retn(?x,aptm_ok);;

(?x in Xid)((?j in Integer) Rsrcs(?j).rollback_retn(?j,?x,tmr_ok))^(~ NumRsrcs) =>
   AP.commit_retn(?x,aptm_error);;
end interface Transaction_Manager_tc;
```

B.2.3 Resource Manager

Resource Manager Interface

```
type Resource is
  interface
    action
      in   Init( id : Integer; val : Integer );
      out  Read( x : Xid; id : Integer; val : Integer );
      out  Write( x : Xid; id : Integer );
    service
      AP  : AP_R;
      TM  : TM_R;
      LK  : dual R_Lock;
end interface Resource;
```

Resource Manager Module Generator

```
module Account( MaxNumTrans : Integer ) return Resource is
    tmregister   : var Boolean;
    inited       : var Boolean;
    rmid, value  : var Integer;
    Locked       : array[Xid] of ref(Boolean)
                        is (1..MaxNumTrans, default is ref_to(Boolean,false));
  parallel
    when (?id in Integer, ?val in Integer) Init(?id,?val) do
      if ( ?id < 3 ) then tmregister := false;
      else tmregister := true;
      end if;
      rmid := ?id;
      value := ?val;
      inited := True;
    end when;
```

```
||
 when TM.open_call where $inited do
   TM.open_retn($rmid, tmr_ok);
 end when;
||
 when (?x in Xid) TM.start_call(?x) where $inited do
   TM.start_retn($rmid, ?x, tmr_ok);
 end when;
||
 when TM.register_call where $inited do
   TM.register_retn($rmid, $tmregister);
 end when;
||
 when (?x in Xid, ?val in Integer) AP.Debit(?x,?val) where $inited do
   if ($tmregister) then
         TM.Reg_call($rmid, ?x);
         await TM.Reg_retn(?x);
   end if;
   LK.Lock_call($rmid, ?x, LOCK_X, LOCK_INSTANT, 0);
   await LK.Lock_retn($rmid, ?x);
   Locked[?x] := true;
   Read(?x, $rmid, $value);
   Write(?x, $rmid);
   value := $value-?val;
   Read(?x, $rmid, $value);
   AP.results($rmid,?x, $value<0);
 end when;
||
 when (?x in Xid) TM.prepare_call(?x) do
   if ( $value < 0 ) then TM.prepare_retn($rmid, ?x, tmr_error);
   else TM.prepare_retn($rmid, ?x, tmr_ok);
   end if;
 end when;
```

```
  ||
    when (?x in Xid) TM.Commit_call(?x) do
      if ( $(Locked[?x]) ) then
          LK.Unlock_call($rmid, ?x);
          await LK.Unlock_retn($rmid, ?x);
          Locked[?x] := false;
      end if;
      TM.commit_retn($rmid, ?x, tmr_ok);
    end when;
  ||
    when (?x in Xid) TM.rollback_call(?x) do
      if ( $(Locked[?x]) ) then
          LK.Unlock_call($rmid, ?x);
          await LK.Unlock_retn($rmid, ?x);
          Locked[?x] := true;
      end if;
      TM.rollback_retn($rmid, ?x, tmr_error);
    end when;
end module Account;
```

B.2.4 Lock Manager

Lock Manager Interface

```
type Lock_Manager(NumObjs : Integer) is
  interface
    action
      in    Init(id : Integer; val : Integer);
      out   Read(x : Xid; id : Integer; val : Integer);
      out   Write(x : Xid; id : Integer);
```

```
    service
      AP : AP_R;
      TM : TM_R;
      LK : dual R_Lock;
      Rsrcs(1 .. NumObjs) : R_Lock;
end interface Lock_Manager;
```

Lock Manager Module Generator

```
module Locker_gen(NumRsrcs : Integer) return Lock_Manager(NumRsrcs) is
    rmid : var Integer;
    inited : var Boolean;
    tmregister : Boolean is true;
    action Animation_Show(s : String; mode : Integer; color : String);
  parallel
    when (?id in Integer, ?val in Integer) Init(?id,?val) do
      rmid := ?id;
      inited := True;
    end when;
  ||
    when TM.open_call do TM.open_retn($rmid,tmr_ok); end when;
  ||
    when TM.register_call do TM.register_retn($rmid,tmregister); end when;
  ||
    when (?x in Xid, ?val in Integer) AP.Debit(?x,?val) do
      assert( false );
    end when;
  ||
    when (?x in Xid) TM.prepare_call(?x) do assert( false ); end when;
  ||
    when (?x in Xid) TM.Commit_call(?x) do assert( false ); end when;
  ||
    when (?x in Xid) TM.start_call(?x) do assert( false ); end when;
  ||
    when (?x in Xid) TM.rollback_call(?x) do assert( false ); end when;
```

```
 ||
   for i : Integer in 1 .. NumRsrcs generate
     when (?x in Xid, ?m in Lock_Mode, ?c in Lock_Class, ?t in Integer)
          Rsrcs(i).Lock_call(i, ?x, ?m, ?c, ?t)
     do
          Animation_Show("Lock Granted",1,"blue");
          Rsrcs(i).Lock_retn(i, ?x, ?m, ?c, ?t, LOCK_OK);
          await Rsrcs(i).Unlock_call(i, ?x);
          Animation_Show("Lock Released",1,"blue");
          Rsrcs(i).Unlock_retn(i, ?x, LOCK_NOT_LOCKED);
     end when;
   end generate;
end module Locker_gen;
```

B.2.5 Resources

```
type Resources(NumRsrcs : Integer) is
  interface
    service
      Rsrcs(1 .. NumRsrcs) : Resource;
end interface Resources;

module Some_Rsrcs(NumRsrcs, MaxNumTrans : Integer) return Resources(NumRsrcs) is
    Rs : array[Integer] of Resource
          is (1..NumRsrcs, default is Account(MaxNumTrans));
  connect
    for i : Integer in 1..NumRsrcs generate
      Rsrcs(i).AP to Rs[i].AP;
      Rsrcs(i).TM to Rs[i].TM;
      Rs[i].LK to Rsrcs(i).LK;
    end generate;
```

```
  parallel
    for i : Integer in 1..NumRsrcs do
      Rs[i].Init(i,10);
    end do;
end module Some_Rsrcs;
```

B.3 Architecture

```
type Bank is interface end interface Bank;

module Bank_System(NumRsrcs, NumTrans : Integer) return Bank is
  MaxNumTrans : Integer is 2*NumTrans;
  AP          : Application_Program(NumRsrcs)
                    is Simple_Program(NumRsrcs, NumTrans);
  TM          : Transaction_Manager_tc(NumRsrcs, MaxNumTrans);
  LM          : Lock_Manager(NumRsrcs) is Locker_gen(NumRsrcs);
  Rs          : array[Integer] of Resource
                    is (1..NumRsrcs, default is Account(MaxNumTrans));
  connect
    AP.TMs  to TM.AP;
    LM.TM   to TM.LM;
    for i : Integer in 1..NumRsrcs generate
      TM.Rsrcs(i)     to Rs[i].TM;
      AP.Rsrcs(i)     to Rs[i].AP;
      Rs[i].LK        to LM.Rsrcs(i);
    end generate;
  initial
    LM.Init(0,0);
    for i : Integer in 1..NumRsrcs do Rs[i].Init(i,10); end do;
  parallel
    AP.Ready();
end module Bank_System;
```

Appendix C

Conformance Testing Maps

C.1 RAPIDE Maps

C.2 Atomicity

```
type Atomicity_Constraints is
  interface
    private
      action
        commit(xid : Integer);
        promise(xid : Integer);
        rollback(xid : Integer);
  constraint
    -- Atomicity Constraint
    never (?i in Integer) commit(?i) and rollback_call(?i);

    -- Coordination Constraint
    never (?i in Integer) promise(?i) || commit_call(?i);
  end interface Atomicity_Constraints;
```

```
map map_gen_for_atomicity() from S : Bank_System to Atomicity_Constraints is
    Xid2Integer : function (x : Xid) return Integer is ...
  rule
    (?i in Integer, ?x in Xid) S.Rsrcs.Rsrcs(?i).TM.prepare_retn(?i,?x,xa_ok)
    => promise( Xid2Integer(?x) );;

    (?i in Integer, ?x in Xid) S.Rsrcs.Rsrcs(?i).TM.commit_call(?i,?x)
    => commit( Xid2Integer(?x) );;

    (?i in Integer, ?x in Xid) S.Rsrcs.Rsrcs(?i).TM.rollback_call(?i,?x)
    => rollback( Xid2Integer(?x) );;
  end map map_gen_for_atomicity;
```

C.3 Isolation

```
type Isolation_Constraints is
  interface
    private
      action
        Read(xid, oid : Integer);
        Write(xid, oid : Integer);
  constraint
    never (?x1, ?x2, ?o in Integer)
      (Read(?x2,?o) -> Write(?x1,?o) -> Write(?x2,?o)) where ?x1 /= ?x2;

    never (?x1, ?x2, ?o in Integer)
      (Write(?x2,?o) -> Write(?x1,?o) -> Write(?x2,?o)) where ?x1 /= ?x2;

    never (?x1, ?x2, ?o in Integer)
      (Write(?x2,?o) -> Read(?x1,?o) -> Write(?x2,?o)) where ?x1 /= ?x2;
```

```
    never (?x1, ?x2, ?o in Integer)
      (Read(?x1,?o) -> Write(?x2,?o) -> Read(?x1,?o)) where ?x1 /= ?x2;
  end interface Isolation_Constraints;

map map_gen_for_isolation() from S : Bank_System to Isolation_Constraints is
    Xid2Integer : function (x : Xid) return Integer is ...
  rule
    (?id in Integer, ?x in Xid, ?v in Integer) S.Rsrcs.Rsrcs(?id).Read(?x,?id,?v)
    => Read( Xid2Integer(?x), ?id );;

    (?id in Integer, ?x in Xid) S.Rsrcs.Rsrcs(?id).Write(?x,?id)
    => Write( Xid2Integer(?x), ?id );;
  end map map_gen_for_isolation;
```

C.4 X/Open

C.5 Bank System Execution

```
module Bank_Test() return Root is
     NumRsrcs     : Integer is String_To_Integer(Arguments[1]);
     MaxNumTrans  : Integer is String_To_Integer(Arguments[2]);
     S            : Bank is Bank_System(NumRsrcs, MaxNumTrans);

--      m is map map_gen_for_atomicity(S);
--      n is map map_gen_for_isolation(S);
end module Bank_Test;
```

C.6 Mapped Execution

C.7 Violation

Bibliography

[1] SRI Computer Science Laboratory: Software Architecture Projects Internet Homepage. URL: http://www.csl.sri.com/ moriconi/mmprojects.html.

[2] G. Abowd, R. Allen, and D. Garlan. Using style to give meaning to software architecture. In *Proceedings of SIGSOFT'93, Software Engineering Notes*, volume 18, pages 9–20. ACM Symposium on Foundations of Software Engineering, December 1993.

[3] Rakesh Agrawal and David J. Dewitt. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Transactions on Database Systems*, 10(4):529–564, December 1985.

[4] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71–80. IEEE Computer Society Press, May 1994.

[5] L. M. Augustin, B. A. Gennart, Y. Huh, D. C. Luckham, and A. G. Stanculescu. Verification of VHDL designs using VAL. In *Proceedings of the 25th Design Automation Conference (DAC)*, pages 48–53, Anaheim, CA, June 1988. IEEE Computer Society Press.

[6] Larry M. Augustin, David C. Luckham, Benoit A. Gennart, Youm Huh, and Alec G. Stanculescu. *Hardware Design and Simulation in VAL/VHDL*. Kluwer Academic Publishers, October 1990. 322 pages.

[7] Gilbert Babin, François Lustman, and Peretz Shoval. Specification and design of transactions in information systems: A formal approach. *IEEE Transactions on Software Engineering*, 17(8):814–829, August 1991.

[8] Andy Barnhard. Component–based Solutions with OLE 2.0. *Software Development*, 2(9):47–51, September 1994.

[9] Frank Belz and David C. Luckham. A new approach to prototyping Ada–based hardware/software systems. In *Proceedings of the ACM Tri-Ada Conference*, Baltimore, December 1990. ACM Press.

[10] Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development.* Prentice/Hall International, 1982.

[11] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In van Eijk et al, editor, *The Formal description Technique LOTOS*, pages 23–73. North-Holland, 1989.

[12] Grady Booch. *Object–Oriented Analysis and Design with Applications.* Benjamin/Cummins, Redwood City, CA, second edition, 1994.

[13] Kraig Brockschmidt. *Inside OLE 2.* Microsoft Press, 1994.

[14] Doug Bryan. Rapide–0.2 Language and Tool-set Overview. Technical Note CSL–TN–92–387, Computer Systems Lab, Stanford University, February 1992.

[15] Doug Bryan. Using Rapide to Model and Specify Inter-object Beahvior. In *OOPSLA '94 workshop on Precise behavioral specifications in OO information modeling*, Oct. 24, 1994.

[16] Zhou ChaoChen, C. A. R. Hoare, and Anders P. Ravn. A duration calculus for real-time requirements in embedded software systems. ProCoS ESPRIT BRA 3104, June 1990.

[17] Panayiotis K. Chrysanthis and Krithi Ramamritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 194–203, Atlantic City, NJ, May 1990.

[18] C. J. Date. *An Introduction to Database Systems*, volume 2 of *The Systems Programming Series*. Addison-Wesley, Reading, Mass., 1985.

[19] C. J. Date. *An Introduction to Database Systems*, volume 1 of *The Systems Programming Series*. Addison-Wesley, Reading, Mass., fourth edition, 1987.

[20] J. Dawes. *The VDM-SL Reference Guide*. Pitman, 1991.

[21] David Dill, August 1995. Discussion with Prof. David Dill of Stanford Univeristy's Computer Science Department.

[22] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[23] K. P. Eswaran, Jim N. Gray, R. A. Lorie, and I. L. Traiger. The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[24] Armen Gabrielian and Matthew K. Franklin. Multilevel specification of real-time systems. *Communications of the ACM*, 34(5):50–60, May 1991.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Professional Computing Series. Addison–Wesley, Reading, MA., 1995.

[26] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, New York, 1993. World Scientific Publishing Company.

[27] B.A. Gennart. *Automated Analysis of Discrete Event Simulations Using Event Pattern Mappings.* PhD thesis, Stanford University, April 1991. Also Stanford University Computer Systems Laboratory Technical Report No. CSL–TR–91–464.

[28] Benoit A. Gennart and David C. Luckham. Validating discrete event simulations using event pattern mappings. In *Proceedings of the 29th Design Automation Conference (DAC)*, pages 414–419, Anaheim, CA, June 1992. IEEE Computer Society Press.

[29] Joseph A. Goguen. Lil — a library interconnect language. In *Report on Program Libraries Workshop*, pages 15–51, Menlo Park, CA, October 1983. SRI International.

[30] Mark Graham and Erik Mettala. The domain–specific software architecture program. In *Proceedings of DARPA Software Technology Conference, 1992*, pages 204–210, April 1992. Also published in *CrossTalk, The Journal of Defense Software Engineering*, pages 19–21, 32, October 1992.

[31] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Mateo, CA, 1993.

[32] Jim N. Gray. *Notes on Database Operating Systems.* Lecture Notes in Computer Science. Springer-Verlag, 1978.

[33] The Object Management Group. *The Common Object Request Broker: Architecture and Specification.* The Object Management Group, distributed by QED–Qiley, Wellesley, MA, revision 1.1 edition, December 1991. OMG Document 91.12.1 by OMG and X/Open.

[34] Jr. H. W. Lockhart. *OSF DCE: A Guide to Developing Distributed Applications.* McGraw–Hill, 1995.

[35] Paul Haggar and Jim Purtilo. Overview of QAD, an interface description language. Technical report, University of Maryland, College Park, MD, January 1993.

[36] D. P. Helmbold and D. C. Luckham. TSL: Task sequencing language. In *Ada in Use: Proceedings of the Ada International Conference*, pages 255–274. Cambridge University Press, May 1985.

[37] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[38] IBM, Research Triangle Park, NC. *System Network Architecture (SNA) Format and Protocol Reference Manual: Architecture Logic for Logical Unit Type 6.2*, 1991. IBM Publication SC30-3269.

[39] IEEE, Inc., 345 East 47th Street, New York, NY, 10017. *IEEE Standard VHDL Language Reference Manual*, March 1987. IEEE Standard 1076–1987.

[40] International Organization for Standardization. *Information Processing Systems – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1)*, December 1987. International Standard 8824.

[41] International Organization for Standardization. *American National Standard for Information Systems - Programming Language C*, 1992. ISO/IEC 9899:1990 (which is technically identical to ANS X3.159-1989, Programming Language C).

[42] International Standards Organization. *Information Processing Systems – Open Systems Interconnection – Basic Reference Model*, 1984. International Standard 7498.

[43] International Standards Organization. *Information Processing Systems – Open Systems Interconnection – Commit, Concurrency Control, and Recovery (OSI-CCR)*, 1989. ISO/IEC 9804.3:1989 (service) and 9805.3:1989 (protocol).

[44] International Standards Organization. *Database Language SQL2, an ISO Standard*, 1992. ISO/IEC 9075.

[45] International Standards Organization. *Information Processing Systems – Open Systems Interconnection – Distributed Transaction Processing (OSI-DTP)*, 1992. ISO/IEC DIS 10026-1:1991 (model), 10026-2:1991 (service), and 10026-3:1991 (protocol).

[46] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[47] Valérie Issarny and Christophe Bidan. Aster: A corba–based software interconnection system supporting distributed system customization. Early draft of a paper to be submitted for publication.

[48] Michael Jackson. *ControlH Users Manual.* Honeywell Systems and Research Center, version 0.19 edition, March 1993.

[49] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *Proc. 21-st ACM Symp. on Principles of Programming Languages, Portland*, 1994.

[50] John J. Kenney. Pattern matching architectures. An internal PAVG document., July 1995.

[51] John J. Kenney and Walter Mann. Anna package specification: Case studies. Technical Report CSL–TR–91–496, Computer Systems Lab, Stanford University, October 1991.

[52] John J. Kenney and Haigeng Wang. RAPIDE 1.0 Examples. Technical report, Computer Systems Lab, Stanford University, 1996. In preparation.

[53] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall Software Series. Prentice Hall, Englewood Cliffs, NJ, 1988.

[54] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, pages 3–42, 1956.

[55] Henry F. Korth and Abraham Silberschatz. *Database System Concepts*. Advanced Computer Science Series. McGraw-Hill, New York, N.Y., 1978.

[56] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[57] B. Lampson and H. Sturgis. Crash recovery in a distributed data storage system. unpublished memorandum, 1976.

[58] D. C. Luckham, D. P. Helmbold, S. Meldal, D. L. Bryan, and M. A. Haberler. Task sequencing language for specifying distributed Ada systems: TSL–1. In Habermann and Montanari, editors, *System Development and Ada, proceedings of the CRAI workshop on Software Factories and Ada. Lecture Notes in Computer Science. Number 275*, pages 249–305. Springer-Verlag, May 1986.

[59] D. C. Luckham and F. W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–23, March 1985.

[60] David Luckham, Francois Guimbretiere, Hai-Geng Wang, and Yung-Hsiang Lu. Applying event-based modelling to the ads high level architecture development process. Unpublished Technical Report.

[61] David C. Luckham. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs*. Texts and Monographs in Computer Science. Springer-Verlag, October 1990.

[62] David C. Luckham, David P. Helmbold, Sigurd Meldal, Douglas L. Bryan, and Michael A. Haberler. Task sequencing language for specifying distributed Ada systems: TSL-1. In *Proceedings of PARLE: Conference on Parallel Architectures and Languages Europe. Lecture Notes in Computer Science. Number 259, Volume II: Parallel Languages*, pages 444–463, Eindhoven, The Netherlands, 15–19 June 1987. Springer-Verlag.

[63] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995. (Also Stanford University Technical Report No. CSL–TR–94–608.).

[64] David C. Luckham and James Vera. Event based concepts and language for system architecture. In *Proceedings of the Workshop on Studies of Software Design*, May 1993.

[65] David C. Luckham and James Vera. μRapide: An Executable Architecture Definition Language. An early version of "An Event-Based Architecture Definition Language," IEEE TSE V21 N9., April 1993.

[66] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[67] David C. Luckham, James Vera, Doug Bryan, Larry Augustin, and Frank Belz. Partial orderings of event sets and their application to prototyping concurrent, timed systems. *Journal of Systems and Software*, 21(3):253–265, June 1993.

[68] David C. Luckham, James Vera, and Sigurd Meldal. Three concepts of system architecture. *submitted to the Communications of the ACM*, July 1995.

[69] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *ANNA, A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[70] D.C. Luckham, S. Meldal, D.P. Helmbold, D.L. Bryan, and W. Mann. An introduction to Task Sequencing Language, TSL 1.5. Technical Report 38, Department of Informatics, University of Bergen, Bergen, Norway, August 1989. Preliminary version.

[71] Neel Madhav. *Correctness and Error Detection in a Model of Distributed Program Executions*. PhD thesis, Stanford University, Stanford, CA 94305-4055,

September 1993. Also Stanford University Computer Systems Laboratory Technical Report No. CSL–TR–93–578.

[72] Robin Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science 92. Springer–Verlag, 1980.

[73] Mark Moriconi and Xiaolei Qian. Correctness and composition of software architectures. In *Proceedings of SIGSOFT'94, Software Engineering Notes*, pages 164–174, New Orleans, LA., December 1994. ACM Symposium on Foundations of Software Engineering.

[74] Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. A formal approach to correct refinement of software architectures. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[75] R. Neches, R. Fikes, T. Finin, T. R. Gruber, R. Patil, T. Senator, and W. R. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):37–56, 1993.

[76] Open Systems Foundation. *Distributed Computing Environment (DCE).* http://www.OSF.org.

[77] C. H. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[78] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[79] D.E. Perry and A.L Wolf. Foundations for the study of software architecture. In *Proceedings of SIGSOFT'92, Software Engineering Notes*, volume 17, no. 4, pages 40–52. ACM Symposium on Foundations of Software Engineering, October 1992.

[80] *IEEE Standard for Information Technology: POSIX Systems Services Interface: Standardization of OS Calls*, 1990.

[81] *IEEE Standard for Information Technology: POSIX Transaction Processing.* A draft standard.

[82] V.R. Pratt. Modeling concurrency with partial orders. *Int. J. of Parallel Programming*, 15(1):33–71, February 1986.

[83] Rapide Design Team. *The Rapide-1 Architectures Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[84] Rapide Design Team. *The Rapide-1 Executable Language Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[85] Rapide Design Team. *The Rapide-1 Pattern Language Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[86] Rapide Design Team. *The Rapide-1 Predefined Types Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[87] Rapide Design Team. *The Rapide-1 Specification Language Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[88] Rapide Design Team. *The Rapide-1 Types Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[89] Rapide Design Team. *The Rapide-1 Language Overview Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, December 1995.

[90] Alexandre Santoro, Woosang Park, and David Luckham. SPARC-V9 architecture specification with Rapide. Technical Report CSL–TR–95–677, Computer Systems Laboratory, Stanford University, September 1995.

[91] Douglas C. Schmidt. Using design patterns to develop reusable object–oriented communication software. *Communications of the ACM*, 38(10), October 1995.

[92] Will Tracz. Parameterized programming in LILEAnna. In *Proceedings of ACM Symposium on Applied Computing SAC'93*, February 1993.

[93] Jeffrey D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 1 of *Principles of Computer Science Series.* Computer Science Press, Rockville, Md., 1988.

[94] US Department of Defense, US Government Printing Office. *The Ada Programming Language Reference Manual*, February 1983. ANSI/MIL-STD-1815A-1983.

[95] Steve Vestal. *Software Programmer's Manual for the Honeywell Aerospace Compiled Kernel (The MetaH Language Reference Manual).* Honeywell Technology Center, Minneapolis, MN, 1994.

[96] Gio Wiederhold. *Database Design.* McGraw-Hill, New York, N.Y., second edition, 1983.

[97] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *COBOL Language*, December 1991. (ISBN: 1-872638-09-X C192 or XO/CAE/91/200).

[98] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *Distributed Transaction Processing: The XA Specification*, June 1991. CAE Specification.

[99] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *Distributed Transaction Processing: The Peer–to–Peer Specification*, December 1992. Snapshot.

[100] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *Distributed Transaction Processing: The TX (Transaction Demarcation) Specification*, November 1992. Preliminary Specification.

[101] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *Distributed Transaction Processing: Reference Model, Version 2*, November 1993. Guide.

[102] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *Transaction Processing – Six Volume Set*, November 1995.

Glossary

Action. A construct that models asynchronous communication between RAPIDE modules. An action is characterized by a name and a set of parameters. See page 39.

Architecture. A distinctive style of designing software. See page 7.

Atomic. After a transaction executes, either all or none of its operations take effect. See page 3.

Availability. After generation an event *immediately* becomes available to other behaviors, modules, their processes or connections. Events become unavailable to a process after they have participated in the triggering of that process. See page 67.

Commit. The declaration or process of making a transaction's updates public (visible to other transactions) and durable. See page 6.

Concurrency. Simultaneous execution of causally independent programs, processes, transactions, and so on. See page 40.

Conformance Testing. A technique for detecting differences between an application's execution and a standard's constraints. See page 15.

Consistency Constraint. A predicate on data that serves as a precondition, post–condition, and transformation condition on any transaction. See page 30.

Consistency. Correct. Data should satisfy certain properties called *consistency constraints*. Transactions should preserve these properties. See page 30.

Consistent Cut. Given a set S and a partial order $<$ on S, a *consistent cut* of $<$ is a partial order $<'$ on S' such that $S' \subseteq S$, $<' \subseteq <$, and $e' \in S' \wedge e \in S \wedge e < e' \rightarrow e \in S'$. See page 72.

Data Object A generic term meaning an object with a simple interface that is limited to reading and writing values. See page 2.

Database State The collection of all data values in the database. See page 24.

Database. A collection of data. See page 1.

Distributed Transaction Processing (DTP). A term used by X/Open and others to describe architectures supporting distributed transactions. These architectures define the interaction among transactional applications (APs), transaction managers (TMs), and resource managers (RMs). See page 7.

Distributed. An adjective for designs, architectures, and organizations. It implies no single point of control, processing, or storage. Contrast with *centralized.* See page 6.

Durability. State that survives failures. Durable memory is contrasted with *volatile* memory, which is reset at system restart. See page 6.

Event. An occurrence of an action. Events are characterized by the name of an action and a list of parameter values. See page 39.

Formal Methods. Mathematically based techniques for describing system properties. See page 9.

Generation. Events are generated by a behavior or process through performing of action, function and patterns calls. The order of event generation is consistent with the causal and temporal orders. See page 67.

Integrity constraint. A predicate on data that serves as a precondition, post–condition, and transformation condition on any transaction. See page 2.

Interleaving. A little bit of one, then a little of the other, then more of the first, etc. See page 40.

Isolation. Transactions are isolated only when they do not overlap other transaction, where overlap is with respect to time or database state. See page 27.

Log. A history of all changes made to the database. See page 3.

Observation. Events are observed by a behavior, module, or process one at a time in some total order that is consistent with the causal and temporal orders. All processes share a consistent observation order. See page 67.

Open System. A extensible system with standard interfaces. See page 14.

Persistent. State or memory that survives failures. Persistent memory contrast with *volatile memory*, that is reset at system restart. See page 6.

Primitive Action. The simplest operation of a transaction, a read or write action. See page 25.

Process. A single thread of control. See page 66.

Program State. The values given to the set of variables contained in the program. See page 39.

Reference Architecture. A clear, precise, executable (behavioral), and testable specification. See page 11.

Resource. A generic term meaning some valuable object with a complex interface, either a piece of data or a piece of hardware. See page 2.

Runtime Consistency Checking. A technique for verifying whether an execution of a program satisfies a specification at run time. See page 15.

Serial Execution. An execution is serial if each transaction runs to completion before the next one starts. See page 27.

Specification. A contract, a valuable piece of documentation, and a means of communication among a client, a specifier, and an implementor. See page 10.

Standard. Models to which others must conform inorder to communicate. See page 7.

Transaction. An *ACID* unit of work (atomic, consistent, isolated, and durable). See page 1.

Validation. A process or methodology for determining the validity of a system with respect to the system's integrity constraints. See page 10.

Verification. Mathematical proof techniques are used to demonstrate design correctness with respect to a set of constraints. See page 10.

Virtual Database State. A database state that may not actually exist at any time. See page 26.

X/Open Company Limited. A consortium of vendors who are defining portability standards for the UNIX environment. See page 8.

X/Open Distributed Transaction Processing (DTP). A distributed transaction processing architecture for a distributed two–phase commit protocol. The architecture defines application programming interfaces and interactions among transactional applications (APs), transaction managers (TMs), resource managers (RMs). See page 8.

C. See page 13.

Index