Mechanical verification of concurrency control and recovery protocols

MECHANICAL VERIFICATION OF CONCURRENCY CONTROL AND RECOVERY PROTOCOLS

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Chkliaev, Dmitri A.

Mechanical verification of concurrency control and recovery protocols / by Dmitri A. Chkliaev. - Eindhoven : Eindhoven University of Technology, 2001.
Proefschrift. - ISBN 90-386-0921-3
NUGI 852
Subject headings : software ; specifications / software verification /
distributed database organisation / fault tolerance
CR Subject Classification (1998) : D.2.4, F.3.1, C.2.4, H.2.7

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2001-11

Printed by University Press Facilities, Eindhoven
Cover design by Jan-Willem Luiten

MECHANICAL VERIFICATION OF CONCURRENCY CONTROL AND RECOVERY PROTOCOLS

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR AAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF.DR. R.A. VAN SANTEN, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VOOR PROMOTIES IN HET OPENBAAR TE VERDEDIGEN OP 27 SEPTEMBER 2001 OM 16.00 UUR

DOOR

DMITRI ALEKSANDROVICH CHKLIAEV

GEBOREN TE NOVOSIBIRSK, RUSLAND

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.Dipl.Ing. D.K. Hammer
en
prof.dr. J.C.M. Baeten

Copromotor:
dr. J.J.M. Hooman

Acknowledgments

Having reached the end of my project, I would like to thank everybody who made it possible for me to write this thesis. First of all, I wish to express my deep gratitude to direct supervisors of this Ph.D. project Jozef Hooman and Peter van der Stok, who organized the project and invited me to Eindhoven to carry it out. Without their daily help and advise I would not have gotten very far. Secondly, I thank Dieter Hammer for his excellent supervision of the Technical Applications (TT) group and for serving as my first supervisor.

I very much enjoyed the pleasant atmosphere in the TT group and nice social contacts with its members. Former TT members Richard Kelleners, Marja de Vroome and Elsa Gelis Escala have been especially helpful and entertaining. Our former secretary Marja de Vroome found an apartment for me where I still live. My former roommates at TUE Paul van Gorp and Maarten Bodlaender helped me to learn UNIX during the first year of my project. I am also grateful to Tineke van den Bosch and Michel Reniers for helping to prepare the Dutch summary of the thesis.

My special thanks go to Alexei Sintotski, who has been a perfect roommate and friend for 3 years and assisted me with many problems related to software and hardware. We had a lot of interesting discussions in room HG 6.85 and turned it into a center of intense activity in the area of formal specification and verification.

I am very grateful to Jos Baeten for offering me an excellent new job, a postdoc position in the group of Formal Methods. I am also grateful to him and my new supervisor Erik de Vink for giving me enough time to work on the final version of the thesis.

I thank members of the the reading committee Dieter Hammer (first supervisor), Jos Baeten (second supervisor), Jozef Hooman (co-supervisor), Peter van der Stok and Peter Apers, for their timely approval of this thesis and many useful remarks. I also thank all other members of the doctoral committee for their agreement to serve in the committee.

Finally, I thank my parents Alexander and Larisa, other family members and all my friends for their motivating support during my stay in Holland.

I learned a lot during this project, but as with any school I am glad that it is finally over. I am happy to obtain a Ph.D. degree at a relatively young age, and I look forward to face new challenges in my life!

Contents

Chapter 1

Introduction

This thesis concerns the formal verification of concurrency control and recovery protocols supported by tools. The aim of this chapter is to give a broad overview of the main topics and results of the thesis. It is organized as follows. Section 1.1 informally describes concurrency control and recovery protocols (CCRPs) and their correctness properties. Section 1.2 introduces the fields of formal methods and mechanical verification. Section 1.3 motivates the choice of tool support for verification of CCRPs. Section 1.4 presents previous work on verification of CCRPs. Section 1.5 gives an overview of the main results of the thesis. Section 1.6 discusses the most significant contributions of the thesis to the field of verification of CCRPs.

1.1 Concurrency control and recovery in data processing systems

We study concurrency control and recovery protocols aimed at *distributed data processing systems*. According to [LMWF94], such systems include, "in addition to traditional database systems, any kind of distributed system that manages long-lived and valuable data: banking, airline reservations, inventory control, and payroll management systems". Data processing systems (DPSs) are often very complex, because they should satisfy a large number of requirements related to their performance, reliability, availability and user interface.

The complete set of requirements is unique for each DPS. However, each DPS should satisfactorily deal with potential errors arising from two sources: concurrency and failures. To prevent such errors, a distributed DPS must support at least the following mechanisms: concurrency control, centralized recovery at a single site and distributed recovery.

One of the main notions for DPSs is a *transaction*, which is a logically indivisible sequence of actions performed on a database. Transactions that interleave their access to the database can interfere. The interleaving of transactions is represented in a schedule, which is a sequence of actions such as reads and writes of data items, where each action belongs to some transaction. *Concurrency control* protocols must prevent interference by ensuring *serializability*. They should only accept a schedule if it is equivalent to some serial schedule, i.e., a schedule that has no interleaving between actions of different transactions.

Centralized recovery protocols (at a single site) deal with two types of failures: transaction failures and memory failures. *Transaction failure* is defined as a situation, when a transaction is unable to finish its execution because of any reason (i.e. deadlock, concurrency control conflict). In this case, a transaction is *aborted*, and the recovery protocol must erase its possible partial effects. It must also ensure that the results of *committed* transactions, i.e. those that complete successfully, are never lost.

These two tasks are complicated by memory failures. *Memory failure* is defined as a loss or corruption of portions of memory because of any reason (for instance, a power failure). A memory failure often also leads to transaction failures, forcing some transactions to abort.

When a transaction accesses data at multiple sites, called the *participants* of this transaction, they all must agree on whether to commit or abort this transaction. *Distributed recovery* protocols provide the interaction between participants needed to reach a decision. Such protocols are also called *distributed (atomic) commitment* protocols.

The complete set of requirements for DPSs with respect to concurrency control and recovery is usually summarized as ACID (Atomicity, Consistency, Isolation and Durability) [GA93]:

- *Atomicity* means that the effect of a transaction is reflected in a database either completely or not at all, depending on whether the transaction is committed or aborted.
- *Consistency* requires some invariants (often called "integrity constraints") to be preserved by each individual transaction in each specified state of a database.
- *Isolation* means that different transactions do not interfere with each other. It is usually replaced by the more precise notion of serializability.
- *Durability* means that the effect of committed transactions must survive subsequent memory failures. This property is closely related to atomicity, and they are usually analyzed together.

Like the majority of modern distributed systems, data processing systems tend to be large, intricate and prone to errors. One of the ways to ensure their correct functioning is the use of a high-quality development process. According to Hoare, such process usually includes "rigorous management of procedures for design inspection and review; quality assurance based on a wide range of targeted tests; continuous evolution by removal of errors from products already in widespread use; and defensive programming, among other forms of deliberate over-engineering" [Hoa96].

However, in this thesis we focus on another possible approach, namely the use of *formal methods* in the process of software development. According to [Rus93], formal methods are "the use of mathematical techniques in the design and analysis of computer hardware and software; in particular, formal methods allow properties of a computer system to be predicted from a mathematical model of the system by a process akin to calculation". An overview of successful applications of formal methods to software analysis, debugging and verification can be found in [Rus93, CW96]. In the next two sections we explain the choice of a particular formal method used in this thesis, namely mechanical verification with the interactive proof checker of PVS.

1.2 Formal methods

Rushby [Rus93] distinguishes four levels of increasing rigor in formal methods :

- **Level 0:** *No use of formal methods.* This corresponds to current standard practice in which verification is a manual process of review and inspection applied to documents written in natural language, pseudocode, or a programming language, possibly augmented with diagrams. Validation is based on testing.
- **Level 1:** *Use of concepts and notation from discrete mathematics.* Some of the natural language used in requirement statements and specifications is replaced with concepts derived from logic and discrete mathematics, such as transitive relations or one-to-one functions. Proofs, if any, are performed in the informal style typical of a mathematical textbook.
- **Level 2:** *Use of formalized specification languages with some mechanized support tools.* Standards and conventions used by developers of the system are systematized as an informal "spec-

ification language". Tool support may be provided in the form of syntax checkers or typecheckers, but not proof checkers. Proofs are usually performed informally, as in Level 1.

- **Level 3:** *Use of fully formal specification languages with comprehensive support environments, including mechanized theorem proving or proof checking.* A specification language has a very direct interpretation in logic, and with corresponding proof methods. Proof methods are completely formalized (i.e., reduced to symbol manipulation) and usually mechanized in the form of a *proof checker* (i.e., a computer program that checks the steps of a proof proposed by the human user), or a *theorem prover* (i.e., a computer program that attempts to discover proofs without human guidance) or, most commonly, of something in between. Henceforth we don't distinguish proof checking and theorem proving, because the majority of modern verification systems support both interactive and automated proof search.

Given a correct mathematical model, level 3 provides the greatest confidence in the correctness of a system, but it is also most time-consuming and costly. Therefore it is only justified for *safety-critical* applications. An application is safety-critical if its incorrect functioning leads to big losses. The most obvious examples are automatic pilots and banking systems. It is often too dangerous (plane) or costly (bank) to test safety-critical applications in the real environment, and this also increases the importance of verification during the development process.

Unfortunately, even for safety-critical systems, the *complete* verification at level 3 is usually too costly, and it may never become widely used in industry. However, it is known that some parts of a safety-critical system are more important and more prone to errors than others. Therefore it is reasonable to focus verification efforts on these "crucial" parts, and to use more "lightweight" formal methods of levels 2, 1 or even 0 for less important parts. In [Hoa96], Hoare suggests that researchers working in the area of verification should "concentrate on the most crucial areas of a large software system, for example, synchronization and mutual exclusion protocols, dynamic resource allocation, and reconfiguration strategies for recovery from partial system failure. It is known that these are areas where obscure time-dependent errors, deadlocks and livelocks (thrashing) can lurk untestable for many years, and then trigger a failure costing many millions".

We completely agree with Hoare, and the protocols for concurrency control and recovery (CCRPs), studied in this thesis, are an example of the "crucial areas" he mentions. Concurrency control protocols dynamically allocate data items to transactions willing to access them, using such information as locks of data items and timestamps of transactions. Values of these data items, as well as locks and timestamps, may be lost in a system failure, and recovery from the failure requires immediate reconfiguration of the system. Such a reconfiguration often involves interaction between several sites and therefore the actions of these sites must be synchronized in some form. As a result, it is extremely difficult to ensure a number of properties for a combination of several distributed fault-tolerant protocols. Also note that some form of concurrency control and recovery is used in each data processing system.

Consequently, the most rigorous study of these protocols, i.e. mechanical verification at level 3, is not a luxury, but a necessity. It is therefore used throughout this thesis.

1.3 Mechanical verification

A special form of mechanical verification, different from proof checking, is *model checking* [CW96] . It is a technique that builds a finite state model of the system and then checks a desired property by an, in principle, exhaustive state space search. The search always terminates since the model is finite, and it is completely automatic and often fast, producing an answer in a few minutes or even seconds for

many models. The main weakness of the method is the *state explosion* problem: the state spaces for realistic systems are often very large, making exhaustive search very resource-consuming. A number of methods are suggested to eliminate "unnecessary" states, such as *partial reduction* and *symmetry reduction* [Bos01, God96, Ip96], but they do not work for all types of systems. In particular, it is not known how to apply model-checking to CCRPs, since they have an infinite (or at least extremely large) number of states. Indeed, the number of transactions executing in a modern system may be very large, and also the number of possible values for data items is usually extremely large. It might be possible to greatly reduce the number of database states using their *abstract interpretation* [Dam96], but this technique would require additional assumptions about the structure of the database. [LMWF94], also dealing with concurrency control and recovery protocols, does not suggest any solution to this problem. They state: "the properties we are concerned with in this book are well beyond the scope of current automatic technology. However, some of the proofs may be verifiable by current proof *checking* systems".

Interactive proof checking seems therefore to be the only feasible approach to verify concurrency control and recovery protocols. It can deal directly with infinite state spaces by using techniques like structural induction to prove properties that are universally quantified over infinite domains. However, it requires interaction with a human, and therefore may be very time-consuming. Another obvious obstacle is a lack of qualified humans to do the proofs.

A large number of proof checking systems is currently available; in [Dat] more than 70 such systems are listed. Some of them, such as Larch, are based on a first-order logic. The advantages of such systems are the relatively easy automation of proofs and the fact that first-order specifications are more easily understood by people with limited knowledge of mathematics. However, a first-order logic does not seem to be expressive enough to capture many interesting features of the protocols studied in this thesis in a convenient way.

For our project, we have chosen to use the verification system PVS [PVS] , based on a higher-order logic. The main reasons for this are as follows: 1) PVS is relatively easy to learn, certainly in comparison with other well-known proof checkers such as Isabelle and Coq [GMvdP98] ; 2) the specification language is very expressive, whereas the specifications are quite readable; 3) the automatic proof search facilities are very powerful; 4) many useful libraries have been provided, both by developers and by individual users.

PVS is developed at SRI, Stanford (in close collaboration with NASA) and has been already used in a large number of case studies, both in academia and in industry. The specification language of PVS is a strongly-typed higher-order logic. Specifications can be structured into a hierarchy of parameterized theories. There is a number of built-in theories and a mechanism for constructing abstract datatypes. The PVS system contains an interactive proof checker with, for instance, induction rules, automatic rewriting, and decision procedures for arithmetic. It allows users to construct proofs interactively, to discharge simple verification conditions automatically, and to check proofs mechanically.

1.4 History of the verification of CCRPs

1.4.1 Classical serializability theory

The notion of serializability has been introduced in [EGLT76] under the name "consistency". It is used as the main correctness condition in [BHG87]. Chapters 1 to 4 of that book present a large body of work on formal modelling of transactions, described as "classical theory of database concurrency control" in [LMWF94]. The most important concurrency control algorithms are motivated, described and verified in a semi-formal style. The verification is usually based on graph theory.

According to [LMWF94], this theory is primarily designed for single-level transaction systems rather than nested transaction systems, and for read and write operations rather than arbitrary database operations. It is assumed that each transaction consists of a fixed collection of operations, i.e. requests do not depend on earlier results. Memory failures are not considered, and it is assumed that the results of aborted transactions are removed by an appropriate recovery mechanism. Serializability, the main correctness notion of the theory, has been criticized in [LMWF94] for the fact that it is defined in terms of operations directly performed on a database, rather than an interface between a database and its users. Despite its shortcomings, the "classical theory" allows simple and rigorous correctness proofs for the most widely used concurrency control protocols, and also serves as a good source of inspiration for more advanced theoretical studies about that subject.

More practical issues related to concurrency control can be found in [GA93]. It also gives a large number of examples. [Pap86] presents a more theoretical analysis of different serializability notions (including more exotic ones), such as the complexity of testing these notions.

[BHG87] also presents protocols for centralized recovery and atomic commitment. Unfortunately, the modelling of them is much less formal than their modelling of concurrency control protocols, and rigorous correctness proofs are not given. For centralized recovery, even the correctness properties are not defined. For pedagogical reasons the protocols for concurrency control, centralized recovery and distributed recovery are studied in isolation, and the problem of combining them in a single system is not sufficiently addressed (see section 1.5.1 for more details).

1.4.2 Extensions of the classical theory

In this subsection, we discuss two important publications, aiming at an extension of the classical theory to allow rigorous verification of more realistic concurrency control protocols:

An issue of *Distributed Computing* from 1992 focuses on the specification and verification of serializability for a relatively simple transaction processing system with transaction failures suggested by Lamport. Broy uses algebraic and functional specification techniques [Bro92], Kurki-Suonio an action-based model similar to I/O automata [KS92], and Lam and Shankar state transition systems [LS92]. The issue ends with a critique by Lamport [Lam92]. His conclusions are rather negative and may be summarized as follows: 1) the problem of serializability turned out to be very difficult; 2) it is even not clear which informal definition of serializability is the most reasonable one; 3) each of the presented methods has major weaknesses; 4) the specification of Kurki-Suonio is not as general as the informal notion of serializability, whereas the specification of Broy is hard to understand, so that Lamport is not even sure what exactly it specifies; 5) the formal proofs are either not given or do not seem to be sufficiently rigorous.

In our opinion, [Bro92], [KS92] and [LS92] may at best be considered as a first step towards the fully rigorous verification of serializability. Each of them uses large specialized models, which seem to be more complex than the system that they are describing: for Broy, it is his version of algebraic specifications, for Kurki-Suonio, the experimental language DisCo, for Lam and Shankar, their theory of modules and interfaces. The excessive complexity of the specifications is also indicated by the fact that the authors are unable to present their formalisms in full detail (although their papers are quite large) and have to refer to their previous papers. Moreover, there are no rigorous proofs, and it is hard to see what they would look like.

[LMWF94] contains a much more ambitious extension: it adds to the classical theory a very general treatment of abstract datatypes, nested transactions and recovery from aborted transactions.

It presents the verification of atomicity (which in their interpretation includes serializability) for a number of large and complex concurrency control protocols, in which transactions failures are possible. Data items are represented by arbitrary structures (such as counters, queues and stacks), and transactions are nested, i.e., allowed to invoke subtransactions. The modelling is very detailed: some activities, described by a single event in the classical theory, are represented by five separate events. However, memory failures and distributed commitment are not considered. All protocols are treated in the framework of I/O automata . Both specifications and implementations are modelled as a parallel composition of a few I/O automata, each representing, for instance, transactions and subtransactions of the system, environment of the system, data objects, the scheduler, as well as other system components. The top-level specification, called a *serial system*, represents the atomicity requirement. The scheduler of a serial system is defined in such a way that it accepts only serial schedules. The implementation, which represents a particular protocol, is said to be correct if its executions "look the same" to the system's environment as executions of a serial system. This is formally defined in terms of trace inclusion, and the verification is usually based on simulation relations. All (very complex) verifications are performed manually, and the possibility of automation is only discussed. Considering the complexity of the specifications and reasoning, such automation appears to be rather challenging, and we don't know of any work in this direction.

In general, we do not question the theoretical importance of the modelling of transaction processing systems presented in [LMWF94]. However, we find it too complicated to serve as a good starting point for mechanical verification. Moreover, in this thesis we decided to extend the classical theory in a different way, by adding the recovery from memory failures and distributed recovery. This topic is just as important as the extensions considered in [LMWF94], because these types of recovery are present in any distributed database.

1.4.3 Other publications

In this subsection, a few other interesting publications on recovery protocols and on database consistency are discussed.

Centralized recovery. In [Kuo96], a data manager responsible for centralized recovery has been verified in an I/O automata framework. Its modelling is very much oriented at the ARIES system and it would be difficult to reuse the results for any other architecture. Just as in [LMWF94], distributed commitment is not considered and all verifications are performed manually.

Distributed recovery. Simplified versions of the Two-Phase Commit (2PC) protocol (without timing, site failures and recovery) has been formally treated in [JZ92], using algebraic techniques, and also in [KST00]. In the last paper, a simple safety property has been verified automatically, using a games-based model-checking technique. In [DF93], a 2PC based on crash-counts has been verified in an I/O automata framework. Finally, in [Sch96], a formal study of Two-Phase Commit and Three-Phase Commit protocols, based on protocol skeletons, has been given. Each of the protocols studied in these papers is not timed and uses a very simple communication mechanism; verification concerns only safety properties (except for [Sch96]). We are not aware of any formal verification of more realistic distributed commitment protocols, especially with tool support.

Database consistency. [Spe99] is the only application of proof checkers to databases we know of. Spelt's thesis contains interesting recent work on the verification of database *consistency*, presenting the verification of some integrity constraints with the support of the Isabelle theorem prover. It

achieves highly automated proofs for a particular object-oriented implementation of a database. We don't consider the verification of database consistency here, because we want to abstract from the details of database implementation.

1.5 Overview of the thesis

1.5.1 Protocols studied

Classical serializability theory. In chapter 2, we give a concise description of the protocols for concurrency control, centralized recovery at a single site and distributed recovery. Chapter 3 presents the formalization of the "classical theory" of database concurrency control in PVS and the verification of two main protocols of that theory: Two-Phase Locking (2PL) and Timestamp Ordering (TSO) . Two main forms of serializability have been formalized: conflict and view serializability. We prove that conflict serializability implies view serializability. This relation is well-known but has never been checked mechanically. A method to verify conflict serializability has been formulated in PVS and proved to be sound and complete with the proof checker of PVS. This method, based on the notion of *conflict-preserving timestamps*, is a modification of a traditional method for proving conflict serializability, based on conflict graphs. Although we proved these two methods to be equivalent, our method seems to be much more appropriate for mechanical verification. We use the method to verify 2PL and TSO protocols.

To make our method applicable to more complex protocols, we observe that many protocols can be modelled as variations of a few basic concurrency control protocols. Although these variations can be obtained in different ways, they can often be considered as *extensions* of a basic protocol. An extension is a protocol, which includes more control information (such as timestamps and versions) and corresponding new actions. To allow easy verification of extensions, we present a systematic way to extend concurrency control protocols with new actions and control information. We show that if such an extension satisfies a few simple correctness conditions, the new protocol is serializable by construction. To illustrate the usefulness of our method, we verify several layered extensions of the 2PL protocol.

Integration of concurrency control and recovery. In most of the the existing literature, the protocols for concurrency control, centralized recovery and distributed recovery are studied in isolation, at the same time making strong assumptions about each other. The problem of combining them in a formal way is largely ignored. For instance, in the "classic" serializability theory it is assumed that the results of aborted transactions are successfully removed by an appropriate recovery mechanism. The situation with centralized recovery protocols is even more complicated. In [BHG87], an abstract model of a database system is presented, which includes the transaction manager, the scheduler, the recovery manager and the cache manager. The recovery manager is solely responsible for recovery, and it is assumed that the scheduler invokes operations of the recovery manager in an order that produces a serializable execution. [Kuo96] shares the same separation of the scheduler and the recovery manager. However, it is also admitted in [BHG87], that in any realistic software architecture these modules are more tightly integrated. It is therefore obvious that "the scheduler" is also prone to failures, and without appropriate measures would be unable to guarantee serializability. Also, the centralized undo/redo protocol in [BHG87] does not support distributed commitment, because it allows to arbitrarily abort uncommitted transactions as a result of a system failure, thus endangering the consistency of a decision to abort a transaction among multiple sites. It would be impossible to introduce distributed commitment into this protocol without significant changes.

In chapter 4, we study the protocols for concurrency control and recovery in a more integrated fashion, with the aim of easier and more convincing verification than possible by combining isolated results about these protocols (that share common data). This also makes verification easier to be reused for more realistic architectures. We present a framework of an experimental transaction processing system (protocol), integrating a few very typical protocols for concurrency control and recovery: strict two-phase locking, undo/redo recovery and two-phase commit. We also give the formal definitions of atomicity, durability and serializability for a fault-tolerant environment.

In our model, both transaction and memory failures are considered, and concurrency control and recovery are closely integrated. We focus on the interaction between stable and volatile memory and on log management; the details of message exchange between distributed sites are, however, not considered. Although our model includes such interesting aspects of these protocols as distribution of data items over an arbitrary number of sites and different types of memory, it is not very complicated. As a result, we were able to completely verify the correctness properties (atomicity, durability and serializability) with the interactive theorem prover of PVS.

Atomic commitment. Chapter 5 presents the verification of non-blocking atomic commitment protocols for distributed recovery. An atomic commitment protocol (ACP) is said to be *non-blocking* if it allows a participant to terminate a transaction despite failures of other participants. Such protocols are desirable since they avoid a waste of recourses at correct participants. Unfortunately the most popular ACP, the Two-Phase Commit (2PC) protocol, may lead to *blocking* executions where a transaction remains unterminated at some correct participants for an arbitrary period of time. Such executions are not acceptable for *real-time database systems*, which impose strict timeliness requirements on transactions (for general information on real-time databases, see [BLS97, HSRT91, BvdS98, Bod99]). Non-blocking ACP's are usually combined with *termination protocols* for participants that crashed and become operational again after being repaired. Their aim is to help a participant to terminate a transaction despite the possible loss of information in a crash.

As a particular case study, we chose the non-blocking protocol of Babaoglu and Toueg (ACP-BT) [BT93a]. They suggested a few non-blocking protocols that share the basic structure of 2PC and achieve the non-blocking property by exploiting the properties of a *uniform reliable broadcast* mechanism which is used to disseminate certain messages. In [BT93b], Babaoglu and Toueg also suggested a termination protocol to be used in combination with their commitment protocol. We found an error in their protocol; there is a combination of crashes and message delays leading to inconsistent decisions. Instead of just correcting the error, we developed our own termination protocol for the ACP of Babaoglu and Toueg. Our protocol is simpler in some respects, it sometimes avoids an unnecessary decision to abort a transaction and has a new improved termination property.

All safety and liveness properties have been completely proved (with the interactive proof checker of PVS) for ACP-BT, combined with our own termination protocol for recovered participants. The protocol has been verified in an incremental fashion: first we verified the basic ACP-BT (without recovery), and then reused almost all of its proofs for the verification of the complete protocol. This verification was very complex and took several months to complete, but greatly improved our understanding of the protocol. PVS provided great help for managing and reusing the proofs. It also helped to formalize the correctness properties much better. For instance, the termination property for ACP's with recovery is defined in [BT93a] as: "If all participants that know about the transaction remain operational long enough, then they all decide". This is very vague, especially because it is not specified how long these participants should remain operational. Using PVS, we developed a new termination property for our protocol, which specifies timing more precisely and expresses a higher degree of fault-tolerance.

Finally, chapter 6 presents the conclusions of our work and a more detailed comparison with related work.

1.5.2 Methods used

Classical serializability theory. In chapter 3, the behaviour of a system is represented by a finite sequence of the form $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_n \xrightarrow{a_n} s_{n+1}$. Here s_i $(0 \leq i \leq n+1)$ are states, and a_i $(0 \leq i \leq n)$ are atomic (indivisible) actions. Besides values of data items, states maintain a control part to determine which actions on data items are allowed and which are not allowed in a particular state of a database. E.g., the control part for lock-based protocols determines which data items are locked and their mode of locking (shared or exclusive). The control part for timestamp-based protocols contains information about timestamps of transactions and data items. For the set of actions, it is only assumed that it includes actions of the form (W, T, x) and (R, T, x). (W, T, x) represents a write action by transaction T on variable x, whereas (R, T, x) represents a read action by transaction T on variable x. Additionally, there are usually other actions, necessary for the concurrency control. In this model, failures are assumed not to occur and the only actions changing the values of data items are writes. Therefore each read always obtains the value of a data item produced by the last write.

A sequence of the form $a_0 a_1 \ldots a_n$, corresponding to behaviour $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_n \xrightarrow{a_n} s_{n+1}$ is called a *schedule*. The main correctness notions of view and conflict serializability are defined by "global" predicates on schedules which explicitly mention only read and write actions. In this way, we abstract from the complete sets of actions and states used by a particular protocol. Serializability is a safety property, and therefore it is sufficient to represent infinite behaviours by all finite approximations.

The predicate defining conflict serializability is based on the well-known notion of conflicting actions, whereas for view serializability we use the *reads-from* relations from [Vid91]. Our approach is quite different from the model presented in [LMWF94], which represents serializability by a separate system with the scheduler that accepts only serial schedules.

A particular concurrency control protocol can be "constructed" by defining the set of actions A, the set of states S, the initial state $is \in S$, as well as the precondition of each action *Pre* (i.e., a predicate on $S \times A$) and the effect of each action *Effect* (i.e., a predicate on $S \times A \times S$). A behaviour $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_n \xrightarrow{a_n} s_{n+1}$ consisting of states from S and actions from A is a *run* of this protocol, if $s_0 = is$, and for all i $(0 \leq i \leq n)$ we have $Pre(s_i, a_i)$ and $Effect(s_i, a_i, s_{i+1})$. Correctness properties are proved by semantics-based reasoning for each schedule corresponding to some run, using the information about the state variables.

Integration of concurrency control and recovery. Performing the verification presented in chapter 3, we discovered that it is not very convenient to work with *finite* schedules in PVS. Indeed, when formulating a PVS lemma about the element of schedule S with index i, we always have to make sure that i does not exceed the length of S. This also generates a lot of annoying type-correctness conditions. To avoid this problem, we represent in chapter 4 the behaviour of a system by an *infinite* sequence of the form $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_n \xrightarrow{a_n} \ldots$. Atomicity, durability and serializability are proved for all finite approximations of these infinite behaviours. Here, it is not unreasonable to consider each finite behaviour as an approximation of some infinite behaviour. The justification is that the protocols for concurrency control and recovery are aimed at continuous processing of transactions during very long periods of time, and they allow new transactions to arrive at any moment.

In this chapter, there is an explicit distribution of data items over an arbitrary number of sites. Each action has a parameter, indicating at which site it is executed. The protocol performed by each

site is also defined by the precondition and the effect of each action and the initial state. In addition, the interaction between sites according to a commit protocol is defined by two "global" axioms on behaviours. Atomicity and durability are represented by the combination of two properties: one expressing the consistency of a decision to commit or abort a transaction, and another expressing the fact that all reads obtain values corresponding to the chosen recovery mechanism ("update in place").

To be able to reuse some definitions and proofs from chapter 3, we again define all correctness properties in terms of schedules, abstracting from state variables. It is known that it is very difficult to specify complicated properties using only sequences of actions and not state variables [Lam89]. To circumvent the problem, "dummy" parameters are given to two actions: read and restart from a crash. E.g., the read action has a parameter indicating the identifier of a transaction that wrote a value being read. As in [BHG87], serializability is defined using *committed projection* of schedules, which removes from schedules actions performed by aborted transactions.

Atomic commitment. In chapter 5, only one transaction is considered at a time, but as in chapter 4, we allow an arbitrary number of sites. The "local" behaviour of each site, executing a commit protocol for this transaction, is represented by an infinite sequence of the form $s_0 \stackrel{a_0}{\rightarrow} s_1 \stackrel{a_1}{\rightarrow} ... \; s_n \stackrel{a_n}{\rightarrow} ....$ Infinite behaviour represents a site that never completes its protocol due to failures of other sites, i.e. never decides "commit" or "abort". Such infinite behaviours cannot be eliminated, because we consider not only safety, but also liveness properties. It is difficult to reason in PVS about both finite and infinite behaviours [DGM97]. This is why we extend behaviours of sites that do decide to infinite behaviours using a fictitious time-out action. "Global" behaviour of a system is defined by a function which assigns to each site its local behaviour. Correctness properties are defined for global behaviours, using not only actions of these behaviours, but also a state variable indicating whether in this state a transaction is committed, aborted or neither.

To specify local behaviours, we extended our *Pre*/*Effect* state machines with timing. Each action has a parameter $time$, indicating the time moment when it happened. Each state has a parameter $timeout$, indicating the time moment before which some action should be taken. We require $time(a_{i+1}) \geq time(a_i)$, $time(a_i) \leq timeout(s_i)$ and non-Zeno behavior (only finitely many actions in finite time). Both sending and broadcasting of messages are formalized by a number of assertions on "global" behaviours.

To prove that the protocol satisfies a correctness property, we combine operational reasoning about state machines and reasoning in terms of these assertions.

1.6 Main contributions

Mechanical verification. Most of the papers published in the field of computer-assisted verification present verification of a single protocol using the tools and techniques favoured by a particular researcher. Studies of related protocols that use similar techniques are very rare. Perhaps the only exceptions are the verification of a few distributed protocols in the context of process algebra (see, e.g., [Kor94, vW95, GMvdP98]) and a few studies based on I/O automata (e.g., [Rom99, Gri00]). Moreover, since different people usually use different tools (e.g., HOL, Coq, Larch, Boyer-Moore prover, PVS), reuse of proofs for verification of a different (related) protocol is almost unknown. We study computer-assisted verification of *closely related* protocols for concurrency control and recovery. For each protocol, we use similar techniques and the same tool (PVS). In total, 2 large protocols, 4 smaller protocols and more than 10 correctness notions have been studied. Proofs for some protocols are reused to verify other protocols. These protocols and notions are organized into 3 sets of theo-

ries, and proofs in each set consist of some 10-thousand interactions with the proof checker. This makes our thesis the first large study on mechanized verification of concurrency control and recovery protocols in the context of databases.

Although our proofs are quite large, this thesis still demonstrates the feasibility of mechanical verification of concurrency control and recovery protocols. For two large protocols in chapters 4 and 5, all safety and liveness properties have been completely verified. The amount of time that the author spent doing these proofs is reasonable (a few months for each protocol), and the proofs are comprehensible. The correctness notions have been very naturally formalized in the higher-order logic of PVS, and relations between them have been extensively studied.

Protocol development. This thesis shows that formal specification and verification do not only provide confidence in the correctness of existing protocols, but also help to more easily develop new correct protocols. In chapter 3, a method to extend concurrency control protocols with new actions and control information has been suggested, which creates new protocols that are serializable by construction. In chapter 4, a few protocols for concurrency control and recovery, usually studied in isolation, have been integrated into a single systems. This integration does not only improve the efficiency of the system, but also allows easier verification than combining proofs for each of a number of isolated protocols that share common data structures. Finally, in chapter 5 our efforts to verify an atomic commitment protocol helped us to find an error and to replace it with a correct and more efficient protocol.

Specification techniques. We use a combination of the constructive and axiomatic approach for the definition of protocols, which seems to be very promising. In chapters 4 and 5, the protocol performed by each of the distributed sites is specified by a state machine, which is a formalism that is close to a possible implementation of the protocol in a programming language. Interaction between sites is specified by a few assertions, because we want to abstract from the implementation of this interaction. Such a combination is very suitable for incremental verification of complex protocols, because the protocol that implements the properties specified by the assertions may be verified later.

Publications. This thesis is based on four papers: [CHvdS99] is the short version of chapter 3, [CHvdS00c] is the short version of chapter 4, [CHvdS00b] describes the formal modelling of the protocol presented in chapter 5 and [CHvdS00a] focuses on the verification of the protocol presented in chapter 5.

Chapter 2

Basic transaction processing techniques

In this chapter, we give a more detailed explanation of requirements for data processing systems and informally describe the main techniques used in such systems. It is organized as follows. In section 2.1, we explain our model of the system and its correctness properties: atomicity, durability and serializability. In the rest of this chapter we describe the most important protocols and techniques that ensure these three properties. In section 2.2, we present three concurrency control protocols: Two Phase Locking, strict Two Phase Locking and Timestamp Ordering. In section 2.3, memory management mechanisms are discussed. Section 2.4 explains the interaction between distributed sites according to atomic commitment protocols.

2.1 Model of the system and correctness notions

A database consists of a set of named *data items*, distributed over a set of *sites*. Sites are interconnected through a communication network and can communicate with each other by exchanging messages. Each site has both volatile and stable memory. *Volatile* memory is fast, but limited in size due to its relatively high cost, and only part of the database can be kept in volatile memory. *Stable* memory is slow, but cheap and abundant. At any time, each data item has a *stable value* in stable memory, and it may also have a *volatile value* in volatile memory.

We consider protocols in which transactions perform atomic actions on data items. The most important actions are read and write, which are the only actions of transactions that directly concern the values of the data items. Let $Read(T, x, v)$ represent a read action of transaction T on data item x obtaining value v, and $Write(T, x, v)$ a write action of transaction T assigning value v to data item x. Actions $(Commit(T), st)$ and $(Abort(T), st)$ indicate the successful and unsuccessful termination of transaction T at site st, respectively. Note that the abort of a transaction may be initiated by the transaction itself or may be forced by the system, for instance after a memory failure. Besides this, there are actions $(Crash, st)$ and $(Recover, st)$, indicating a memory failure at site st and restart after a memory failure at site st, respectively, as well as other actions for memory management. A finite or infinite sequence of actions forms a schedule . Thus actions in a schedule are totally ordered and not partially ordered as in [BHG87].

For performance reasons, read and write actions are performed on volatile values of data items. However, a volatile value may disappear at any time (for instance, as a result of a memory failure), and if it does not exist, a read action obtains a stable value of this data item. Values are usually labelled by identifiers of transactions that produced these values. For instance, if transaction $T1$ writes to x, we denote the value that it assigns as x_{T1}, thus obtaining action $Write(T1, x, x_{T1})$. It is assumed

that by the time execution starts according to a schedule, each data item x has *initial* stable value x_{T0} and no volatile value (in the more formal model initial values are assumed to be produced by *initial transaction* $T0$).

2.1.1 Atomicity and durability

We are interested in three of the ACID properties: atomicity, durability and serializability. In our model, atomicity and durability can be represented by the combination of two properties:

1) *Decision consistency.* The decisions to commit or abort a transaction are *consistent* in a schedule S: any transaction that commits at some site in S, does not abort at any site in S.

2) *"Read policy"*. The informal definition of atomicity given in the introduction uses the expression "reflected in a database", which should not be understood literally. Indeed, in our model stable and volatile values of each data item may be different at any time, and a read action may obtain either of them. Therefore it is more appropriate to define presence in a database only in terms of observable behaviour , i.e. the results of a user's interaction with the database by means of transactions. In this thesis we restrict ourselves to the "classical" model, in which the only actions of transactions that access the value of a data item are read and write. Therefore, the formal definition of atomicity should relate the values obtained by read actions to the values produced by write actions.

Two different "read policies" are used in practice. For instance, consider the schedule $Write(T1, x, x_{T1})Read(T2, x, v)$, where $T1 \neq T2$. In the *"deferred update"* approach, each read should obtain the "last committed value" of a data item, i.e. the last value written to a data item that is produced by a committed transaction. Since transaction $T1$ has not committed yet, the read by $T2$ should ignore the value of x produced by it, therefore obtaining the initial value x_{T0}. In the *"update in place"* approach, each read should obtain the "last non-aborted value" of a data item, i.e. the last value written to a data item that is produced by a transaction that has not aborted yet. Therefore, the read of x by $T2$ should obtain x_{T1}. If $T1 = T2$, then in both approaches $T2$ should obtain x_{T1}. Note that these definitions of atomicity also imply *durability*, i.e. the requirement that values produced by committed transactions cannot be lost before they are overwritten by other committed values. Henceforth we assume "update in place" policy, because it is much more common in practical applications.

2.1.2 Serializability

For a schedule S, let $C(S)$ denote its *committed projection,* i.e. a schedule obtained from S by removing all actions except for reads and writes of committed transactions. Note that if a transaction T reads a value written by a transaction that subsequently aborts, then T should also be aborted. Since we assume "update in place" for S, it is easy to see that in $C(S)$ each read obtains exactly the last value written to a data item. S is said to be *serializable*, if $C(S)$ is (in some sense) equivalent to some serial schedule. Serial schedules are those which have no interleaving between actions of different transactions. For instance, schedule $Write(T2, y, y_{T2})Write(T1, x, x_{T1})Read(T1, y, y_{T2})$ is serial, because an action by $T2$ precedes both actions by $T1$. Schedule $Write(T1, x, x_{T1})Write(T2, y, y_{T2})$ $Read(T1, y, y_{T2})$ is not serial, because two actions by $T1$ are interleaved by an action by $T2$.

There are different ways to define equivalence of schedules. The most intuitively appealing one leads to the notions of *view serializability* . Informally, two schedules are view equivalent iff for each transaction $T1$ that reads the value of x written by $T2$ in one schedule, $T1$ also reads the value of x

written by $T2$ in the other schedule. A schedule is said to be view serializable, if it is view equivalent to some serial schedule. For instance, schedule $Write(T1, y, y_{T1})Write(T2, y, y_{T2})Read(T1, y, y_{T2})$ is view serializable, because it is view equivalent to serial schedule $Write(T2, y, y_{T2})Read(T1, y, y_{T2})$. Although these two schedules consist of different actions, in both of them $T1$ reads the value of y produced by $T2$, whereas no transaction reads the value of y produced by $T1$. However, schedule $Read(T1, y, y_{T0})Write(T2, y, y_{T2})Read(T1, y, y_{T2})$ is not view serializable, because it is impossible to construct a serial schedule in which $T1$ reads both the initial value of y and the value produced by $T2$.

Another form of schedule equivalence is *conflict equivalence*, leading to conflict serializability. We say that actions $a1$ and $a2$ are *conflicting* in $C(S)$, if they are performed either 1) by the same transaction, or 2) on the same data item and at least one of them is write. It is easy to see that if two consecutive actions in a schedule are not conflicting, then their order does not influence the result of executing this schedule. Note that the order of actions by the same transaction may also not influence the result of a schedule, but we do not allow swapping them, because actions of a transaction are assumed to be totally ordered. A schedule is said to be conflict serializable, if it can be transformed into a serial schedule by a finite sequence of swaps of consecutive non-conflicting actions. For instance, schedule $Write(T1, x, x_{T1})Read(T2, x, x_{T1})Read(T2, y, y_{T0})Read(T1, y, y_{T0})$ is conflict serializable, because it can be transformed into a serial schedule $Write(T1, x, x_{T1})Read(T1, y, y_{T0})Read(T2, x, x_{T1})$ $Read(T2, y, y_{T0})$ by first swapping non-conflicting actions $Read(T2, y, y_{T0})$ and $Read(T1, y, y_{T0})$, and then non-conflicting actions $Read(T2, x, x_{T1})$ and $Read(T1, y, y_{T0})$. However, schedule $Write(T1, y, y_{T1})Write(T2, y, y_{T2})Read(T1, y, y_{T2})$, which is already shown to be view serializable, is not conflict serializable, because it is not serial and all consecutive actions in it are conflicting.

There is no efficient algorithm to determine whether a schedule is view serializable. Indeed, it is shown in [Pap79] that testing view serializability is NP-complete. However, for conflict serializability there are algorithms with a complexity that is proportional to the square of the length of schedule being tested. Thus testing conflict serializability is reasonably efficient, and this is why this notion is more often used in practice than view serializability.

The most popular algorithm for testing conflict serializability is based on *conflict graphs*. For schedule S, nodes in this graph are transactions that performed some actions in S, and a directed arrow from $T1$ to $T2$ ($T1 \neq T2$) is included in the graph if S includes conflicting actions $a1$ and $a2$ by $T1$ and $T2$ respectively, such that $a1$ precedes $a2$. S is conflict serializable if and only if its conflict graph is acyclic. Thus the conflict graph of schedule $Write(T1, y, y_{T1})Write(T2, y, y_{T2})Read(T1, y, y_{T2})$ consists of arrows $T1 \longrightarrow T2$ and $T2 \longrightarrow T1$. They form a cycle, and we again conclude that this schedule is not conflict serializable.

2.1.3 Schedulers

The part of a database system responsible for concurrency control is called a *scheduler*. It ensures serializability by restricting the order in which reads, writes, commits and aborts of different transactions are performed. After receiving a particular action from some transaction, the scheduler can take one of three decisions:

- Execute the action (if the action is a read, returning the value being read to the corresponding transaction).
- Reject it, forcing the corresponding transaction to abort.
- Delay it by placing it in an internal queue (and later remove it from the queue and either execute or reject it).

The simplest scheduler may operate as follows. If after schedule S is executed it receives action $a1$, it constructs the conflict graph for schedule $S1 = S\ a1$. If this graph is acyclic, $a1$ is executed. Otherwise, $a1$ is rejected or delayed. Such schedulers do exist [BHG87], but they are not often used in practice, because testing whether a large graph is acyclic still requires a lot of recourses. In section 2.2 we present schedulers that operate according to much more efficient protocols, that are based on *locks* and *timestamps* instead of conflict graphs. They ensure serializability without performing much computation at all.

2.2 Locks and timestamps

2.2.1 Basic Two Phase Locking protocol

The basic 2PL protocol, as described in most textbooks on databases (e.g., [SKS97, BHG87]), requires that access to data items is done in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item. There are various modes in which a data item may be locked. The basic 2PL protocol has only two modes:

- **Shared**. If a transaction T has obtained a shared-mode lock on item x, then T can read, but cannot write, x.
- **Exclusive**. If a transaction T has obtained an exclusive-mode lock on item x, then T can both read and write x.

Let A and B represent arbitrary lock modes. Suppose that transaction $T2$ requests a lock of mode B on item x on which transaction $T1$ ($T1 \neq T2$) currently holds a lock of mode A. If $T2$ can be granted a lock on x immediately, in spite of the presence of the mode A lock, then we say that mode B is *compatible* with mode A. In the 2PL protocol, shared mode is compatible with shared mode, but not with exclusive mode; exclusive mode is not compatible with both shared and exclusive modes.

To access a data item, transaction T must first *lock* that item in the corresponding mode. To request shared or exclusive lock on x, it submits action $Slock(T, x)$ or $Xlock(T, x)$ to the scheduler, respectively. If the data item is already locked in an incompatible mode, the request to lock this item is delayed. Later, when the incompatible lock is released, this lock request may be executed. For instance, if we have schedule $Xlock(T1, x)Write(T1, x, x_{T1})$ and transaction $T2$ wants to read x, the scheduler will delay its $Slock(T2, x)$ action. Transaction $T1$ may *release* its exclusive lock on x by issuing action $Unlock(T1, x)$ or $Downgrade(T1, x)$. In the last case, the mode of its lock is *downgraded* from exclusive to shared. After the exclusive lock by $T1$ is released, the scheduler may perform the delayed request by $T2$ to lock x. Thus schedule $Xlock(T1, x)Write(T1, x, x_{T1})Unlock(T1, x)$ $Slock(T2, x)Read(T2, x, x_{T1})$ will be accepted by the 2PL protocol.

Delayed requests are usually performed according to their order in the queue of the scheduler. However, there also exist other techniques for ordering lock requests. For instance, if the system assigns each transaction a *priority*, we may always execute a lock request made by a transaction with the highest priority among the delayed ones.

The 2PL protocol requires that each transaction issues lock and unlock requests in two phases:

- **Growing phase.** A transaction may obtain locks, but may not release any lock.
- **Shrinking phase.** A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. When the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests. Thus a transaction is in the shrinking phase if an only if it performed at least one unlock or downgrade action, and is in the growing phase otherwise. For instance, after schedule $Xlock(T1, x)Write(T1, x, x_{T1})$ $Unlock(T1, x)Slock(T2, x)Read(T2, x, x_{T1})$, transaction $T1$ is already in the shrinking phase (because it performed $Unlock(T1, x)$ action), whereas transaction $T2$ is still in the growing phase. Thus the scheduler based on the 2PL protocol will accept action $Xlock(T2, y)$ in this state, and will reject action $Xlock(T1, y)$.

An important problem of the 2PL schedulers is that they are subject to *deadlocks*. For instance, if after schedule $Xlock(T1, x)Write(T1, x, x_{T1})Xlock(T2, y)Write(T2, y, y_{T2})$ the scheduler receives first $Slock(T1, y)$ and then $Slock(T2, x)$, it will have to delay both of these lock requests. After that, neither $T1$ nor $T2$ can complete without violating the two-phase rule. For instance, action $Slock(T1, y)$ may be executed only after $T2$ releases its exclusive lock on y. But $T2$ will enter the shrinking phase after releasing its lock on y, and will never be allowed to perform action $Slock(T2, x)$. Thus $T1$ will forever wait for permission to read y, whereas $T2$ will forever wait for permission to read x.

When the scheduler detects a deadlock, it aborts one of the transactions involved in this deadlock. Usually the transaction with the lowest priority is aborted. However, detecting deadlocks is not easy. One simple strategy is based on *timeouts*. If the scheduler finds that a transaction is waiting too long for a lock, it guesses that it is involved in a deadlock and therefore aborts it. However, this may be a mistake, because a transaction may just be waiting for a lock kept by another transaction that is taking a long time to finish. Another strategy is to maintain a directed *waits-for graph* with an edge from transaction $T1$ to transaction $T2$ if $T1$ is waiting for $T2$ to release some lock. If there is a cycle in this graph, then transactions involved in this cycle form a deadlock, and one of them must be aborted. There is also a number of techniques used together with the 2PL protocol that allow to prevent deadlocks.

2.2.2 Recoverability and strict locking

When a transaction aborts, we should erase not only its effects on data, but also on other transactions. Therefore if a transaction $T2$ reads a value of x written by $T1$, and $T1$ subsequently aborts, then $T2$ should also be aborted. Since atomicity does not allow us to abort a transaction that has already committed, $T2$ cannot commit until all transactions that wrote values read by $T2$ are guaranteed not to abort, that is, have themselves committed. Executions in which a transaction T cannot commit until all transactions that wrote values read by T have committed are called *recoverable*.

Aborting a transaction $T1$ may cause a whole chain of transactions to abort, because we may have to abort a transaction $T2$ that read a value written by $T1$, then a transaction $T3$ that read a value written by $T2$, and so on. This phenomenon is called *cascading abort*, and it is very undesirable in practice. We say that a database *avoids cascading aborts* (or is *cascadeless*), if it does not allow a transaction to read a value until the transaction that wrote that value commits or aborts. Cascadeless executions are obviously recoverable.

From a practical point of view, even avoiding cascading aborts is not enough for easy recovery from aborts. Indeed, if $T2$ writes to x and later aborts, it is convenient to undo its effect on x by restoring the *before image* of x, i.e. the value that x had just before been over-written by $T2$. However, if this before image is produced by a transaction that also aborts by the time $T2$ aborts, then it is useless for recovery. It is therefore convenient to prevent a transaction from over-writing a value produced by an uncommitted transaction. Executions that satisfy both this condition and cascadelessness are

called *strict*. Strict executions therefore forbid both dirty reads, i.e. reads of values produced by uncommitted transactions, and dirty over-writes, i.e. writes to data items whose current values are produced by uncommitted transactions.

The basic 2PL protocol allows both dirty reads and dirty over-writes. As the schedule $Xlock(T1, x)$ $Write(T1, x, x_{T1})Unlock(T1, x)Slock(T2, x)Read(T2, x, x_{T1})Commit(T2)Abort(T1)$ shows, its executions are even not recoverable. However, there is a version of 2PL called *strict 2PL* that produces only strict executions. In strict 2PL a transaction is not allowed to release any locks before it commits or aborts. When it commits or aborts, it releases all its locks in one atomic action. It is easy to see that this approach indeed eliminates all dirty reads and dirty over-writes. For example, in both schedules $Xlock(T1, x)Write(T1, x, x_{T1})Commit(T1)Slock(T2, x)Read(T2, x, x_{T1})$ and $Xlock(T1, x)Write(T1, x, x_{T1})Abort(T1)Slock(T2, x)Read(T2, x, x_{T0})$, transaction $T2$ reads committed values of x, x_{T1} and x_{T0}, respectively. Although strict 2PL allows much less flexibility than basic 2PL and keeps locks for much longer periods, it is very convenient for recovery. This is why in practice 2PL implementations usually take the form of strict 2PL.

2.2.3 Timestamp Ordering Protocol

The Timestamp Ordering protocol (TSO), cited from [SKS97], is defined using the *timestamps* of transactions and data items. With each transaction $T1$ in the system, we associate a unique fixed timestamp $TS(T1)$. This timestamp is assigned when the transaction $T1$ *enters the system* by performing a $Start(T1)$ action. If a transaction $T1$ has been assigned timestamp $TS(T1)$, and a new transaction $T2$ enters the system afterwards, then $TS(T1) < TS(T2)$. We mention two simple methods for implementing the scheme:

- Use the value of the system clock as the timestamp; that is, a transaction's timestamp equals the value of the clock when the transaction enters the system.
- Use a counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp equals the value of the counter when the transaction enters the system. This approach will be used in our modelling of the protocol in PVS in chapter 3.

In chapter 3, we will show that these timestamps also determine the serializability order in the correctness proof. Thus, if $TS(T1) < TS(T2)$, then the protocol ensures that the produced schedule is equivalent to some serial schedule in which $T1$ appears before $T2$.

Two timestamp values, read and write timestamps, are also associated with each data item x:

- $Wts(x)$, which denotes the largest timestamp of any transaction that successfully wrote x.
- $Rts(x)$, which denotes the largest timestamp of any transaction that successfully read x.

These timestamps are updated whenever a new read or write action on x is performed. The scheduler based on the TSO protocol ensures that any conflicting read and write actions are executed in timestamp order, and it operates as follows:

- Suppose that transactions T wants to read x.
  - If $TS(T) < Wts(x)$, then this implies that T is attempting to read a value of x which was already overwritten. Hence, the read action is rejected and T is aborted.
  - If $TS(T) >= Wts(x)$, then the read action is executed, and $Rts(x)$ is set to the maximum of $Rts(x)$ and $TS(T)$.
- Suppose that transactions T wants to write x.

- If $TS(T) < Rts(x)$, then this implies that the value of x that T is producing was previously needed and it was assumed that it would never be produced. Hence, the write action is rejected and T is aborted.
- If $TS(T) \geq Rts(x)$ and $TS(T) < Wts(x)$, then this implies that T is attempting to write an obsolete value of x. Hence, the write action is ignored.
- Otherwise, i.e. if $TS(T) \geq Rts(x)$ and $TS(T) \geq Wts(x)$, the write action is executed, and $Wts(x)$ is set to $TS(T)$.

If an action violates the timestamp order, this will never change in the future (because timestamps of data items may only increase). This is why the corresponding transaction is aborted and not delayed. A transaction T, which is aborted as a result of an attempt to read or write a data item, is assigned a new timestamp and restarted.

It is easy to see that the following schedule will be accepted by the TSO protocol: $Start(T1)$ $Start(T2)Write(T2, x, x_{T2})Start(T3)Write(T3, x, x_{T3})Write(T1, y, y_{T1})Read(T2, y, y_{T1})$. Suppose that timestamps of transactions are assigned using a counter with initial value 1 and that initial write and read timestamps of x and y are equal to 0. In this case after executing this schedule we have: $TS(T1) = 1$, $TS(T2) = 2$, $TS(T3) = 3$, $Wts(x) = 3$, $Rts(x) = 0$, $Wts(y) = 1$ and $Rts(y) = 2$. Note that this schedule will not be accepted by the 2PL protocol, Indeed, in 2PL, $T2$ must release its exclusive lock on x before action $Write(T3, x, x_{T3})$. After action $Write(T1, y, y_{T1})$ it must obtain a shared lock on y to read y, and this would contradict two-phase rule of 2PL.

The following schedule will be accepted by the 2PL protocol: $Start(T2)Start(T3)Xlock(T3, x)$ $Write(T3, x, x_{T3})Unlock(T3, x)Slock(T2, x)Read(T2, x, x_{T3})$. However, this schedule will not be accepted by the TSO protocol. Indeed, if timestamps are assigned as in the previous example, at the time before $Read(T2, x, x_{T3})$ action is executed we have: $TS(T2) = 1$ and $Wts(x) = 2$. Thus this last action will be rejected.

Executions of this basic TSO protocol are not recoverable, because it does not restrict commits of dependent transactions. The TSO scheduler avoids deadlocks, because it never forces a transaction to wait. However, it may waste recourses by aborting too many transactions. In chapter 3, we ignore aborted transactions under the assumption that their effects are removed by an appropriate recovery mechanism.

2.3 Memory management

As we already mentioned, transactions perform reads and writes on versions of data items located in volatile memory. Therefore a write does not immediately change the value of a data item in stable memory. The exchange of data between volatile and stable memory is realized by operations $Fetch$ and $Flush$. $Fetch(x)$ copies x from stable into volatile memory, whereas $Flush(x)$ moves x from volatile into stable memory. A flush operation is usually performed when there is no space in volatile memory for new data items. If a transaction attempts to read a data item existing only in stable memory, that item must be fetched to volatile memory and read after that.

There are two types of memory failures: *system failures*, when the entire contents of volatile memory are lost, and *media failures*, when portions of stable memory are lost. System failures are repaired using redundant copies of data in stable memory. To provide resilience to media failures, we must keep the contents of stable memory on at least two different storage devices that have a sufficiently low probability of failing at the same time. We consider only recovery from system failures in this thesis, because media failures are relatively rare, and recovery from them is more difficult and sometimes even impossible.

Having both volatile and stable memory complicates un-doing effects of aborted transactions, and also introduces the problem of re-doing effects of committed transactions. We say that a recovery mechanism *requires undo*, if it allows to flush data items written by a transaction that has not committed yet. If a system failure occurs at this point, on recovery the stable database will contain effects of the uncommitted transactions, which usually must be *undone* to ensure atomicity of future reads. For instance, let's consider schedule $Write(T1, x, x_{T1})Flush(x)(Crash, st)(Recover, st)$ $(Abort(T1), st)$ (with x and y located at site st), in which $T1$ aborts as a result of a system failure at site st. After the crash, the stable value of x is equal to x_{T1} (because of the flush action on x), whereas the volatile value of x does not exist. If the effects of $T1$ are not undone, these values will remain the same. Therefore a read action on x by any transaction will obtain x_{T1}, i.e. a value produced by an aborted transaction $T1$.

We say that a recovery mechanism *requires redo*, if it allows a transaction to commit before all the values it wrote have been flushed from volatile to stable database. Should a system failure occur at this point, on recovery the stable database will be missing some of the effects of the committed transactions, which must be *redone*. For instance, let's consider schedule $Write(T1, x, x_{T1})$ $(Commit(T1), st)(Crash, st)(Recover, st)$ (with x located at site st). After the crash, the stable value of x is equal to the initial value $x0$ (because the value produced by $T1$ has not been flushed), whereas the volatile value of x does not exist (because it was lost in a crash). If the effects of $T1$ are not redone, these values will remain the same. Therefore a read action on x by any transaction will obtain x_{T0}, i.e. a value over-written by a committed transaction $T1$.

In this thesis, we concentrate on the protocols that require both undo and redo. Such protocols allow to flush values from volatile memory whenever it is convenient for performance reasons, and therefore may be very efficient. However, they also tend to be rather complex.

To make undo and redo after a system failure possible, we must store some information about the values of data items written by transactions in stable memory in addition to the stable database itself. Such information is usually stored as a *log*. In the basic undo/redo algorithms (e.g., the algorithm in in [BHG87]) the log is represented as a sequence of entries of the form $[T, x, v]$, identifying the value v that transaction T wrote into data item x, and such entry is added each time the corresponding write occurs. The log also contains the sets of committed and aborted transactions, as well as other information. When volatile memory is lost in a system failure, the protocol examines the log, finds the last committed values of all data items that have ever been updated, and restores them as new volatile values of these data items.

Copies of data items in the log may consume a lot of space. If most updates modify only small portions of data items, it would be more efficient to add to the log only the portion of each data item that was actually modified. This may be implemented by a *partial logging algorithm* for a particular physical or logical structure of data items, which may consist of pages, records, stacks, queues etc. For instance, if each data item is a page consisting of a number of lines, and a read action inserts a new line l as the fifth line of page p, a traditional algorithm would add the complete new value of p to the log (because from its viewpoint, all of p is being written). A partial "logical" logging algorithm would only add a log record that says "insert line l as the fifth line of p".

Another method to reduce the size of the log is to remove from it periodically all entries that are no longer needed for recovery. Note that if a transaction aborts, we no longer care about values that it wrote. If a data item has several committed values in the log, we will only need the *last* of these values for recovery, and can discard all others. The procedure that removes useless log entries is called *garbage collection*.

Another important technique used in combination with undo/redo is *checkpointing*, which transforms the *stable database* into the state in which it contains the last committed value of each data

item. After a system failure, we may only have to undo or redo the data items modified after the last checkpointing. Therefore checkpointing significantly reduces the amount of work needed to recover from a failure. One simple checkpointing method [BHG87] is to stop accepting new transactions periodically, wait for all active transactions to commit or abort, flush all volatile values, and them mark the end of the log to indicate that the checkpointing took place.

2.4 Atomic commitment

As mentioned in section 1.1, a distributed transaction accesses data at multiple participants. After all actions of a transaction are executed, its participants must reach a consistent decision on whether to commit or abort this transaction. A participant will be willing to abort transaction T and discard the values it wrote, if execution of T violates serializability, atomicity or some other requirement (e.g., T is involved in a deadlock). Otherwise, it will be willing to commit T and make its updates permanent. It is clear that a participant may *decide* to commit T only if it is willing to commit T and all other participants of T are willing to commit T.

In *atomic commitment* protocols (ACP), a participant *votes*, i.e. distributes a message containing its *vote*, YES or NO, to indicate whether it is willing to commit or abort a transaction. In the simplest ACP's, called *One-Phase Commit* protocols, a participant sends its YES or NO vote directly to all other participants of T. After that, it decides to commit T if all votes it receives are YES, and decides to abort T otherwise. The disadvantage of such ACP's is the large number of messages that have to be sent: for n participants, at least $n(n-1)$ messages.

In *Two-Phase Commit* (2PC) protocols, one of the participants also acts as a *coordinator* to orchestrate the decision process about a transaction among participants. Often the site where a transaction originated serves as a coordinator. A Two-Phase ACP usually works as follows. In Phase 1, after receiving an order from the coordinator to *vote*, each participant sends to the coordinator its vote: YES or NO. The coordinator collects all votes and makes a decision. If a YES vote was received from all participants, then the decision is **commit**; otherwise it is **abort**.

In Phase 2, the coordinator disseminates the decision to all participants. If a participant receives the decision from the coordinator, it decides **commit** or **abort** according to this decision; if no decision arrives, it should find some other way to reach a decision. After deciding **commit** a participant makes the transaction's updates permanent, after deciding **abort** it discards all the updates. It is easy to see that in 2PC protocols fewer messages are sent than in One-Phase Commit protocols, and this is why 2PC protocols are used in practice.

The interaction between participants may be complicated by possible failures of some of these participants or the communication network connecting them. Here we do not consider communication failures. For site failures, we assume that a site may be either *operational* or *crashed* at any given time. When operational, it follows exactly the actions specified by the programs it is executing. Failures, such as system failures, may cause operational sites to *crash*, after which they take no actions at all. A site that has never crashed since the start of the protocol is said to be *correct*. Crashed sites may *recover* to become operational again. During recovery, they may use the contents of stable storage that is kept in the *log*.

2.4.1 Correctness properties

Any reasonable ACP should satisfy at least the following properties:

AC1: All participants that decide reach the same decision.
AC2: If any participant decides **commit**, then all participants must have voted YES.
AC3: If all participants vote YES and no failures occur, then all participants decide **commit**.
AC4: Each participant decides at most once (that is, a decision is irreversible).

An ACP is said to be non-blocking if it satisfies the following property in addition to AC1-AC4:

AC5: Every *correct* participant eventually decides.

Unfortunately, most ACP's cannot prevent blocking executions. For example, consider an execution where the coordinator crashes during the distribution of the decision message. Suppose that some participant p receives the decision and then crashes, while all other participants stay correct, but do not receive the decision. If p or the coordinator do not recover, then other participants cannot decide even after contacting each other: any such decision may contradict the decision made by p.

Note that this non-blocking property is stated in terms of participants that never crash. If a participant crashes and then recovers, it should reach a decision using a *termination protocol* rather than a basic ACP. When some recovered participant p cannot achieve a decision independently (usually because of the risk of violating AC1), it must contact other participants, ask them about their state, and then decide on the basis of their replies.

In some scenarios, p will need the replies of *all other* participants in order to decide. In these scenarios, it is necessary that all other participants remain operational long enough to reply to p's request for help. A termination protocol is non-blocking, if this condition is also *sufficient* for p to recover. This means, if p recovers and stays operational for some sufficiently long period of time and all other participants also stay operational during this period of time, then p can always decide on the basis of their replies. It is easy to see that this *termination property* can also be defined in the following equivalent form:

AC6: If all participants remain operational long enough, then they all decide.

2.4.2 Broadcast and timeouts

To disseminate the decision message to all participants, the coordinator often uses some *broadcast service*. This service may be described as *reliable* or *unreliable*. If the broadcast service is unreliable and the broadcaster crashes during the broadcast to a group of processes, then some processes (correct or not) from this group may receive the message, whereas some other correct process may never receive it. If the broadcast service is reliable, this scenario is impossible. We will show in chapter 5 that the use of reliable broadcast may make the 2PC protocol non-blocking.

The main idea of such non-blocking protocols is to make decisions based on *timeouts*. Suppose correct participant p knows the upper time bound for receiving the decision message from the coordinator by any correct participant. If this time bound has passed, p may conclude that the coordinator has crashed and the decision message is lost. Since the broadcast service is reliable, p may also conclude that no other participant received the decision message or will ever receive it. Thus no other participant will decide **commit**, and therefore p may decide **abort** without endangering the decision consistency.

Reliable broadcast can be implemented by having each site *relay* the broadcasted message to some of the other recipients of this message before accepting it. Although this generates additional messages, broadcast still allows to achieve the non-blocking property at lower cost than other proto-

cols such as *Three-Phase Commit*, which adds to the 2PC an additional round of message exchange between the coordinator and the participants.

Chapter 3

Classical Serializability Theory

In this chapter, we formalize the "classical theory" of database concurrency control in PVS and verify a few important protocols. As we already mentioned, transaction and memory failures are assumed not to occur, and each read obtains the last written value of a data item. The main correctness properties in this model are view and conflict serializability, which must be verified for each finite schedule accepted by a particular protocol. We first present their mathematical definitions in higher-order logic, and later show how to implement these definitions in PVS. Since most PVS proofs are very large and technical, we show only mathematical proofs extracted from them, in which the less important details are left out.

The specification language of PVS is certainly not very complicated. Therefore, we expect the reader who read the first six sections of this chapter to be able to understand the PVS specifications given in section 3.7. These specifications include complete definitions of protocols and correctness notions, as well as the most important lemmas and theorems.

This chapter is organized as follows. In section 3.1, view and conflict serializability are defined, their formalization in PVS is explained, and their relation to each other is proved. In section 3.2, we present a method to verify conflict serializability, based on conflict-preserving timestamps, and show its soundness and completeness. We also show the equivalence of our method to a traditional method based on conflict graphs. In section 3.3, we provide a general specification pattern for protocols and apply it to the specification of the Two Phase Locking protocol and the Timestamp Ordering protocol. In section 3.4, conflict serializability is verified for these protocols using our method. In section 3.5, we formalize extensions of protocols and the restrictions on these extensions, needed to ensure their correctness. In section 3.6, we apply our method to specify and verify several layered extensions of the 2PL protocol. Section 3.7 contains the PVS specifications.

3.1 View and Conflict serializability

The intuitive meaning of view and conflict serializability was already explained in section 2.1.2. In this section, we present the rigorous formalization of these notions, first in mathematical notation and then in PVS notation. Three changes are made to the model used in section 2. Firstly, since each read obtains the last written value of a data item, it is no longer necessary to provide a "value" parameter to read and write actions to study dependencies between transactions. Secondly, $Crash$ and $Recover$ actions and actions for memory management are not considered. Thirdly, all aborted transactions are assumed to be removed by an appropriate recovery mechanism, so we don't have to consider commit and abort actions of transactions.

In PVS, we assume that each action belongs to some transaction. Consequently, each action is represented by a record with three fields, called act, tr, and dat, expressing that a particular action is performed by a transaction on a data item. (W, T, x) and (R, T, x) represent write and read actions by transaction T on data item x, respectively. We may also have other actions, for instance, for locking and unlocking data items. If a is an action represented by (A, T, x), we have $act(a) = A$, $tr(a) = T$ and $dat(a) = x$.

To define view serializability, we first define view equivalence between schedules, following [SKS97].

Definition 3.1
The schedules $S1$ and $S2$ are *view equivalent* if the following three conditions are met for each transaction $T1$ and $T2$ and a data item x:

1. Transaction $T2$ reads the initial value of x in schedule $S1$ if and only if it also reads the initial value of x in schedule $S2$.
2. Transaction $T2$ reads a value of x written by $T1$ in schedule $S1$ if and only if it also reads a value of x written by $T1$ in schedule $S2$.
3. Transaction $T1$ performs the last write action on x in schedule $S1$ if and only if it also performs the last write action on x in schedule $S2$.

Conditions 1 and 2 ensure that each transaction reads the same values in both schedules. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.

The definition of view equivalence can be presented in a more formal way using the notion of a *reads-from* relation [Vid91]. To define it, we add to our set of transactions two special transactions $T0$ and TF. The transaction $T0$ is a fictitious write-only *initial* transaction which writes the initial values of all the data items and doesn't have any other actions, and TF is a fictitious read-only *final* transaction that reads the values of all the data items and doesn't have any other actions. $T0$ and TF are fictitious in the sense that their actions do not appear in actual schedules, and this is why they are treated separately in the following definition of a reads-from relation. Henceforth, the number of elements in S is represented by $length(S)$, and $S(i)$ $(0 \leq i < length(S))$ represents the action in a schedule S with index i.

Definition 3.2
The *reads-from* relation $Reads\_from$ of a schedule S is defined as follows:

$$
\begin{aligned}
&Reads\_from(S) = \{\, (T1, x, T2) \mid \\
&\quad ((T1 = T0 \,\&\, T2 = TF) \Longrightarrow (\forall k, T3 : k < length(S) \Longrightarrow S(k) \neq (W, T3, x))) \,\&\\
&\quad ((T1 = T0 \,\&\, T2 \neq TF) \Longrightarrow \\
&\qquad (\exists m : m < length(S) \,\&\, S(m) = (R, T2, x) \,\&\, (\forall k, T3 : k < m \Longrightarrow S(k) \neq (W, T3, x)))) \,\&\\
&\quad ((T1 \neq T0 \,\&\, T2 = TF) \Longrightarrow \\
&\qquad (\exists l : l < length(S) \,\&\, S(l) = (W, T1, x) \,\&\\
&\qquad\quad (\forall k, T3 : (l < k \,\&\, k < length(S)) \Longrightarrow S(k) \neq (W, T3, x)))) \,\&\\
&\quad ((T1 \neq T0 \,\&\, T2 \neq TF) \Longrightarrow \\
&\qquad (\exists l, m : l < m \,\&\, m < length(S) \,\&\, S(l) = (W, T1, x) \,\&\, S(m) = (R, T2, x) \,\&\\
&\qquad\quad (\forall k, T3 : (l < k \,\&\, k < m) \Longrightarrow S(k) \neq (W, T3, x)))) \,\}
\end{aligned}
$$

Definition 3.3
Two schedules S and S' are *view equivalent,* represented by $view\_equiv(S, S')$ iff their reads-from relations are equal.

It is easy to see that definition 3.3 is just a more formal version of definition 3.1.

Definition 3.4
The schedule S is serial, represented by $serial(S)$, if it has no interleaving between actions of different transactions. In mathematical form:
$serial(S) \stackrel{def}{\Longleftrightarrow} \forall T1, T2, i, j, k :$
$(i < j \ \& \ j < k \ \& \ k < length(S) \ \&$
$T1 = tr(S(i)) \ \& \ T2 = tr(S(j)) \ \& \ T1 = tr(S(k))) \Longrightarrow T1 = T2$

Definition 3.5
The schedule S is view serializable, represented by $View\_serializable(S)$, iff it is view equivalent to some serial schedule:
$View\_serializable(S) \stackrel{def}{\Longleftrightarrow} \exists S0 : serial(S0) \ \& \ view\_equiv(S, S0)$

Next we define conflict equivalence and conflict serializability. Suppose S includes two consecutive actions $a1 = (A1, T1, x)$ and $a2 = (A2, T2, y)$, where $A1$ and $A2$ belong to $\{R, W\}$. As already explained in section 2, the order of $a1$ and $a2$ does not influence the result of computation if either $x \neq y$ or ($x = y$ and $A1 = A2 = R$). If $x = y$ and ($A1 = W$ or $A2 = W$), then the order of $a1$ and $a2$ may change the result of computation. The order should also be maintained if $T1 = T2$, since actions of a transaction are totally ordered.

Definition 3.6
The actions $(A1, T1, x)$ and $(A2, T2, y)$ are *conflicting* iff
$T1 = T2 \ \vee \ (x = y \ \& \ (A1 = W \ \vee \ A2 = W))$.

Definition 3.7
The schedules $S1$ and S are *elementary equivalent* iff $S1 = S3\, a1\, a2\, S4$, $S2 = S3\, a2\, a1\, S4$ for some $S3$ and $S4$, and the actions $a1$ and $a2$ are not conflicting.

Definition 3.8
The schedules $S1$ and $S2$ are *conflict equivalent*, denoted by *conf_equiv(S1, S2)*, iff there is a finite sequence of schedules S_0, S_1, ... S_k, $k \geq 0$, such that $S1 = S_0$, $S2 = S_k$ and for all $i < k$ the schedules S_i and S_{i+1} are elementary equivalent.

Definition 3.9
The schedule S is conflict serializable, denoted by *Conf_serializable(S)*, iff it is conflict equivalent to some serial schedule:
Conf_serializable(S) $\stackrel{def}{\Longleftrightarrow} \exists S0 : serial(S0) \ \&$ *conf_equiv*$(S, S0)$

Since swaps of non-conflicting actions do not change the result of a computation, we can expect that they do not change the reads-from relation as well. Indeed, we proved a PVS lemma, expressing that any conflict equivalent schedules $S1$ and $S2$ are also view equivalent.

$\forall S1, S2 : conf\_equiv(S1, S2) \Longrightarrow view\_equiv(S1, S2)$ ConfViewLem

Lemma ConfViewLem is proved by a tedious analysis of all subcases in definition 3.2, which is not given here. It easily implies the main result of this section, namely that any conflict serializable schedule is also view serializable. This result is represented by PVS theorem ConfViewStronger.

$\forall S : Conf\_serializable(S) \Longrightarrow View\_serializable(S)$ ConfViewStronger

3.1.1 PVS implementation

In PVS, the set of atomic actions is defined by non-empty *type* `ActionNames`, containing at least read and write actions (denoted as `R` and `W`). These two actions are defined as *constants* of the type `ActionNames`. In the PVS notation (henceforth written in `typewriter` style):

```
ActionNames : TYPE+

R, W : ActionNames
```

The set of data items is defined by uninterpreted type `DataItems`; the set of transactions is defined by type `Transactions`, representing the names of transactions. Moreover, in PVS *theory* `Actions` we define the type `Actions` consisting of *records* (denoted by `[# ... #]`) with three fields, called `act`, `tr`, and `dat`, the meaning of which we already explained. A theory starts with a name, followed by a number of parameters (types or constants). The specification itself is between the keywords `BEGIN` and `END`. A theory can be imported by another theory, which provides actual types and values for the parameters.

```
Actions [ActionNames, Transactions, DataItems: TYPE+] : THEORY
BEGIN

Actions : TYPE = [# act : ActionNames,
                    tr  : Transactions,
                    dat : DataItems #]

END Actions
```

In theory `ConfOrd`, which imports theory `Actions`, a schedule S is defined as a *variable* of the type `Schedules` (variable declarations are marked by the keyword `VAR`). This type includes all finite sequences of actions. Formally, S is a function from the set of natural numbers less than $length(S)$ to the type `Actions`. Just as in our mathematical notation, the element of S with index i is represented by $S(i)$, and the type-checking mechanism of PVS always requires $i < length(S)$.

```
ConfOrd [ Transactions, DataItems, ActionNames : TYPE+,
   R, W: ActionNames ] : THEORY
BEGIN

IMPORTING Actions [ActionNames, Transactions, DataItems]

Schedules : TYPE = finite_sequence[Actions]

S : VAR Schedules
```

Serializability notions are defined using the well-known logical symbols such as & (conjunction, also written as `AND`), `OR` (disjunction), `=>` (implication, also written as `IMPLIES`), `NOT` (negation), `FORALL` (universal quantification), `EXISTS` (existential quantification), `TRUE` (truth) and `FALSE` (falsehood). For instance, the PVS definition of a serial schedule looks almost the same as its mathematical definition 3.4:

```
serial(S) : bool = FORALL T1, T2, i, j, k :
   (i < j & j < k & k < length(S) &
    T1 = tr(S(i)) & T1 = tr(S(k)) & T2 = tr(S(j))) => T1 = T2
```

A theory usually contains a number of definitions followed by a number of lemmas or theorems about these definitions. Lemmas and theorems have names and are marked by the keywords `LEMMA` and `THEOREM`, respectively. Besides different keywords, there is no other difference between lemmas and theorems. PVS takes the closure of each lemma or theorem with respect to its free variables. E.g., theorem ConfViewStronger from this section is represented in theory `ConfView` as follows:

```
ConfViewStronger : THEOREM
   Conf_serializable(S) => View_serializable(S)
```

PVS also has an `IF-THEN-ELSE` construction to analyze subcases, and a `WITH` construction to override a value of a record or a function. At the end of this chapter, PVS theories `ConfOrd` and `ConfView` are given, which include PVS definitions corresponding to definitions from 3.1 to 3.9. By comparing these definitions, the reader can get a good insight into the use of PVS notation.

3.2 Our method of verification

We present a general method for mechanical verification of conflict serializability. Our approach is a modification of a traditional method for proving conflict serializability based on *conflict graphs*. We do not use graphs, but do need a notion of conflicting transactions which is defined as a *conflict relation*.

Definition 3.10
A conflict relation $Conflict(S)$ of a schedule S is defined as follows: an ordered pair $(T1, T2)$ belongs to $Conflict(S)$ iff $T1 \neq T2$ and

- S includes actions $a1$ and $a2$ by $T1$ and $T2$, respectively,
- $a1$ precedes $a2$ in S,
- $a1$ and $a2$ are conflicting.

It is well-known (although never mechanically verified) that a schedule S is conflict serializable iff the relation $Conflict(S)$, considered as a graph in which nodes are transactions, is acyclic. Our method does not use graph theory, but assigns *timestamps* to transactions. A timestamp is technically a very simple notion: a function from the set of transactions to natural numbers.

Definition 3.11
A timestamp TS is a *conflict-preserving timestamp (CPT)* with respect to schedule S, denoted by $CPT(S, TS)$, iff for each transactions $T1$ and $T2$ the following condition holds:
$CPT(S, TS) \stackrel{def}{\Longleftrightarrow} Conflict(S)(T1, T2) \Longrightarrow TS(T1) < TS(T2)$

For example, if $S = (W, T1, x)(R, T2, x)(W, T3, y)(W, T4, y)$, then the following timestamps $TS1$ and $TS2$ are conflict-preserving with respect to S:

- $TS1(T1) = 3$, $TS1(T2) = 4$, $TS1(T3) = 1$, $TS1(T4) = 2$ and $TS1(T) = 0$ for all other transactions T,

- $TS2(T1) = 7$, $TS2(T2) = 8$, $TS2(T3) = 9$, $TS2(T4) = 10$ and $TS2(T) = 11$ for all other transactions T.

However, the following timestamp $TS3$ is not conflict-preserving with respect to S: $TS3(T1) = 4$, $TS3(T2) = 3$, $TS3(T3) = 1$, $TS3(T4) = 2$ and $TS3(T) = 0$ for all other transactions T.

Definition 3.12
A schedule S is *ordered*, denoted by $Ordered(S)$, iff there is a timestamp TS which is conflict-preserving with respect to S:
$Ordered(S) \stackrel{def}{\Longleftrightarrow} \exists TS : CPT(S, TS)$

We proved that any ordered schedule is conflict serializable, and any conflict serializable schedule is ordered.

$\forall S : Ordered(S) \Longleftrightarrow Conf\_serializable(S)$ OrdConfEquiv

Proof of theorem OrdConfEquiv. The proof has been constructed by means of the interactive proof checker of PVS and is technically fairly complicated. We give an idea of the two directions of the proof.

$\forall S : Conf\_serializable(S) \Longrightarrow Ordered(S)$ ConfOrd

According to the definition of conflict serializability, there is a serial schedule $S0$, which is conflict equivalent to S. We define a timestamp $TS0(S0)$, which assigns timestamps of transactions according to the the order of appearance of these transactions in $S0$. More formally, if a transaction performed some actions in $S0$, then $TS0(S0)$ assigns to such transaction the index of one of its actions, and assigns 0 to all other transactions:

$IndexSet(S0, T) = \{i \mid i < length(S0) \,\&\, T = tr(S0(i))\}$

$TS0(S0)(T) = \; if \; IndexSet(S0, T) \neq \emptyset \; then \; choose(IndexSet(S0, T)) \; else \; 0$

It is easy to prove $CPT(S0, TS0(S0))$. Since S and $S0$ are conflict equivalent, it is also not difficult to prove that their conflict relations are equal. This implies that $TS0(S0)$ is also a CPT for S, i.e. S is indeed ordered. This completes the proof of ConfOrd.

$\forall S : Ordered(S) \Longrightarrow Conf\_serializable(S)$ OrdConf

According to the definition of $Ordered$, there exists a CPT TS for S. Note that TS need not be injective, i.e. two or more transactions, which performed some actions in S, may have equal timestamps. However, to prove conflict serializability of S, we must establish a *total* order on all transactions participating in S. Such total order is formally defined by the notion of *injective* timestamps. Let $active(S)$ represent the set of all transactions which performed some actions in S:

$active(S) = \{\, T \mid \exists\, i : i < length(S) \,\&\, T = tr(S(i)) \,\}$

Timestamp TS is said to be *injective* for S, represented by $inj(S, TS)$, if it assigns different timestamps to all transactions in $active(S)$:

$$inj(S, TS) \stackrel{def}{\Longleftrightarrow} \forall T1, T2 : (T1 \in active(S)\ \&\ T2 \in active(S)\ \&\ T1 \neq T2) \Longrightarrow TS(T1) \neq TS(T2)$$

Definition 3.13
A schedule S is *injectively ordered*, denoted by $OrderedI(S)$, iff there is a timestamp TS which is conflict-preserving and injective with respect to S.
$OrderedI(S) \stackrel{def}{\Longleftrightarrow} \exists TS : CPT(S, TS)\ \&\ inj(S, TS)$

We split the proof of OrdConf into two parts. In theorem OrdOrdI, we prove that any ordered schedule is also injectively ordered, because it is possible to transform any CPT to an injective CPT. In theorem OrdIConf, we prove that any injectively ordered schedule is conflict serializable.

$$\forall S : Ordered(S) \Longrightarrow OrderedI(S) \qquad \text{OrdOrdI}$$

$$\forall S : OrderedI(S) \Longrightarrow Conf\_serializable(S) \qquad \text{OrdIConf}$$

3.2.1 Proof of theorem OrdOrdI

First we prove that if a schedule S has a CPT TS, then the *transitive closure* of $Conflict(S)$ is acyclic.

Definition 3.14
A relation $ConflictTC(S)$ of a schedule S, called the *transitive closure* of $Conflict(S)$, is defined as follows: a pair $(T1, T2)$ belongs to $ConflictTC(S)$ iff there is a finite sequence of transactions $T_0, T_1, \ldots T_k$, $k \geq 0$, such that $T1 = T_0,\ T2 = T_k$ and for all $i < k$ the pair (T_i, T_{i+1}) belongs to $Conflict(S)$.

Definition 3.15
A schedule S is said to have an *acyclic conflict relation*, represented by $Cacyclic(S)$, iff the transitive closure of its conflict relation is irreflexive: $\forall T : (T, T) \notin ConflictTC(S)$.

If $(T1, T2) \in ConflictTC(S)$ and $CPT(S, TS)$, it is easy to prove by induction on the length of sequence connecting $T1$ and $T2$ in definition 3.14, that $TS(T1) < TS(T2)$. This implies the following theorem:

$$\forall S : Ordered(S) \Longrightarrow Cacyclic(S) \qquad \text{OrdAcyc}$$

By this theorem it remains to prove the most difficult part, namely that any schedule with an acyclic conflict relation has an injective CPT.

$$\forall S : Cacyclic(S) \Longrightarrow OrderedI(S) \qquad \text{AcycOrdI}$$

Suppose $Cacyclic(S)$. Then $ConflictTC(S)$ is a transitive and irreflexive relation on the set of transactions $active(S)$. In PVS theory `AssignInj` (see section 3.7.3), we prove the following general fact: for any non-empty final set fs of elements of arbitrary type Tp and any transitive and irreflexive relation ord on elements of Tp, there exists a *topological sorting* of fs with respect to ord, i.e. a function $ts(fs, ord)$ from fs to natural numbers, satisfying the following properties (*) and (**):

$$(x \in fs \;\&\; y \in fs \;\&\; (x, y) \in ord) \Longrightarrow ts(fs, ord)(x) < ts(fs, ord)(y) \qquad (*)$$

$$(x \in fs \;\&\; y \in fs \;\&\; x \neq y) \Longrightarrow ts(fs, ord)(x) \neq ts(fs, ord)(y) \qquad (**)$$

Thus $ts(fs, ord)$ is an injective function on elements of fs, which preserves the order introduced by ord. In PVS theory `AcycOrd` (see section 3.7.3), which imports theory `AssignInj` substituting the type `Transactions` for Tp, we use this topological sorting to obtain an injective CPT. Timestamp $TSI(S)$ is obtained as follows: if $T \in active(S)$, then $TSI(S)(T) = ts(active(S), ConflictTC(S))(T)$, and $TSI(S)(T) = 0$ otherwise. Since $(T1, T2) \in Conflict(S)$ implies $(T1, T2) \in ConflictTC(S)$, by using (*) and (**) we can easily prove that $TSI(S)$ is indeed an injective CPT for S.

Existence of a topological sorting. To complete the proof of theorem AcycOrdI, it remains to prove the existence of a topological sorting $ts(fs, ord)$ satisfying (*) and (**). Let's define the *set of maximal elements* of fs according to ord as follows:

$$maxset(fs, ord) = \{\, t1 \mid t1 \in fs \;\&\; \forall t2 : t2 \in fs \Longrightarrow (t1, t2) \notin ord \,\}$$

We proved that if ord is transitive and irreflexive and fs is non-empty, then $maxset(fs, ord)$ is also non-empty. The proof is by induction on the cardinality of fs. If $card(fs) = 1$, then the only element of fs is obviously maximal in fs. Suppose now $card(fs) = k + 1$. fs can be represented as $fs1 \cup \{t0\}$ for some arbitrary $t0 \in fs$. We have $card(fs1) = k$, and by the induction hypothesis, $maxset(fs1, ord) \neq \emptyset$. Suppose that there exists $t1 \in maxset(fs1, ord)$ such that $(t1, t0) \in ord$. Then $t0 \in maxset(fs, ord)$. Indeed, suppose that $(t0, t2) \in ord$ for some $t2 \in fs$. Irreflexivity of ord implies $t2 \in fs1$, and transitivity of ord implies $(t1, t2) \in ord$. This contradicts the maximality of $t1$ in $fs1$. Thus for any $t1 \in maxset(fs1, ord)$ we have $(t1, t0) \notin ord$. Let $t3$ represent an arbitrary element of $maxset(fs1, ord)$. It is easy to see now that $t3$ also belongs to $maxset(fs, ord)$, and this completes the induction step.

Let $max(fs, ord)$ represent an arbitrary element of $maxset(fs, ord)$. Function $ts(fs, ord)$ is recursively defined below, where $rest(fs, ord)$ stands for $fs \setminus \{max(fs, ord)\} = \emptyset$.

$$ts(fs, ord)(x) = \begin{cases} 1 & rest(fs, ord) = \emptyset, \\ \begin{cases} card(fs) & x = max(fs, ord), \\ ts(rest(fs, ord), ord)(x) & \text{otherwise.} \end{cases} & \text{otherwise.} \end{cases} \qquad (3.1)$$

Thus $ts(fs, ord)$ assigns $card(fs)$ to a maximal element of fs, and then recursively call the definition of $ts(rest(fs, ord), ord)$. Cardinality of the corresponding set decreases by 1 with every call, and therefore this recursion always terminates. By a tedious induction on $card(fs)$, similar to induction that we used to prove $maxset(fs, ord) \neq \emptyset$, it is possible to prove that $ts(fs, ord)$ indeed satisfies (*) and (**).

3.2.2 Equivalence of serializability notions

We proved $Ordered \Longrightarrow Cacyclic \Longrightarrow OrderedI \Longrightarrow Conf\_serializabile \Longrightarrow Ordered$ in PVS theorems OrdAcyc, AcycOrdI, OrdIConf and ConfOrd, respectively. This means that all these notions are equivalent. In particular, our timestamp-based methods $Ordered$ and $OrderedI$

are equivalent to the traditional method for proving conflict serializability, represented by predicate $Cacyclic$. In this thesis, we use either $Ordered$ or $OrderedI$ to prove conflict serializability for a particular protocol. It is easier to use $Ordered$, because this method does not require a timestamp to be injective. However, if we want to not only prove serializability, but also obtain serial schedules corresponding to schedules of our protocol, an injective CPT is more convenient. Indeed, an injective CPT immediately determines the order of transactions in the corresponding serial schedules, and therefore we don't have to apply topological sorting. This is why we use an injective CPT to verify the 2PL protocol and the TSO protocol.

3.2.3 Proof of theorem OrdIConf

Suppose that timestamp TS is an injective CPT for S. Our goal is to construct a sequence of non-conflicting swaps, transforming S to a serial schedule. Some additional definitions are needed. Let $dset(S, TS)$, the "distance set", denote the set of pairs of indices such that corresponding actions in S are "wrongly" ordered by TS:

$$dset(S, TS) = \{\ (l, m)\ \mid\ l < m\ \&\ m < length(S)\ \&\ TS(tr(S(l))) > TS(tr(S(m)))\ \}$$

Let $d(S, TS)$, which we call "the distance between S and a serial schedule according to T", denote the number of elements in $dset(S, TS)$. It is easy to prove that if $d(S, TS) = 0$ and TS is injective for S, then S is serial. We can also prove that if S is not serial and TS is injective for S, then $d(S, TS) > 0$. Indeed, suppose that S is not serial. It means there exist actions $a1$, $a2$ and $a3$, transactions $T1$ and $T2$ such that $a1$ and $a3$ are performed by $T1$, $a2$ is performed by $T2$, and in S, $a1$ precedes $a2$ and $a2$ precedes $a3$. Injectivity of TS for S implies that either $TS(T1) > TS(T2)$ or $TS(T2) > TS(T1)$. In the first case, the pair of indexes of actions $a1$ and $a2$ belongs to $dset(S, TS)$. In the second case, the pair of indexes of actions $a2$ and $a3$ belongs to $dset(S, TS)$. The definition of $d(S, TS)$ now implies $d(S, TS) > 0$.

Let $swap(S, k)$, where $k + 1 < length(S)$, denote the schedule obtained from S by swapping its actions with indexes k and $k + 1$. Let the set of schedules $swapset(S, TS)$ to be defined as follows:

$$swapset(S, TS) = \{\ S2\ \mid\ \exists\, k : (k, k+1) \in dset(S, TS)\ \&\ S2 = swap(S, k)\ \}$$

We proved that if $d(S, TS) > 0$, i.e. S is not serial, then $swapset(S, TS)$ is nonempty. Let $S1$ denote an arbitrary schedule from this set. TS is a CPT for S, which implies that $S1$ is obtained from S by a non-conflicting swap. We also proved that $d(S1, TS) = d(S, TS) - 1$, i.e. $S1$ is "closer" to a serial schedule than S (technically, a rather difficult PVS lemma).

These results provide a basis for our proof. We recursively construct a sequence of schedules $FS(S, TS)$, in which S is the first schedule, and each consecutive schedule is arbitrarily chosen from the $swapset$ of the previous one. We stop this process as soon as we reach schedule $S0$ such that $d(S0, TS) = 0$. It is obvious that "distance" d decreases by 1 with each consecutive schedule, and therefore exactly $d(S0, TS)$ swaps will be needed. Since TS is injective for S, and swaps do not change the set $active$, we prove by induction on the length of $FS(S, TS)$ that TS is also injective for all other schedules in $FS(S, TS)$, including $S0$. As we already proved, this implies that $S0$ is serial. Again by induction on the length of $FS(S, TS)$, we prove that TS is also a CPT for all other schedules in $FS(S, TS)$. It implies that each schedule in $FS(S, TS)$ is obtained from the previous one by a *non-conflicting* swap. This proves that S can be transformed into a serial schedule by a sequence of non-conflicting swaps, i.e. S is indeed conflict serializable.

3.3 Specification of protocols

As mentioned in the introduction, concurrency control protocols maintain a control part to determine which actions on data items are allowed and which are not allowed in a particular state of a database. For instance, the control part for lock-based protocols determines which data items are locked and in which mode, whereas the control part for timestamp-based protocols contains information about timestamps of transactions and data items. In PVS, the control part for database protocols is defined by type `States`.

Each action causes certain changes in the control part. For example, for lock-based protocols, it may lock or unlock some data items. For timestamp-based protocols, this concerns the adjustment of read and write timestamps of some data items. Therefore, we define the initial value of the control part, i.e. the initial state, and how the control part is changed after every possible action. We also have to define which actions are allowed in a particular state, and which are not. E.g., a transaction cannot lock a data item in an exclusive mode if it is already locked by another transaction. Consequently, a concurrency control protocol is defined by the following steps:

1. Define type `ActionNames`, containing the atomic actions `R` and `W` and possibly some other atomic actions, responsible for the adjustment of control information.
2. Define type `States`, containing all control information essential for the definition of the protocol, and define the initial state `is`.
3. Define which actions are allowed in a particular state by means of the `Pre` predicate; a function with two arguments of types `States` and `Actions`, respectively, and result of type `bool`. For a state `s1` and an action `a1` we have `Pre(s1, a1) = TRUE` iff `a1` is allowed in `s1`.
4. Define how a particular state is changed after applying a particular (allowed) action by the `Effect` predicate. For states `s1` and `s2` and an action `a1`, we have `Effect(s1, a1, s2) = TRUE` iff `s2` is obtained from `s1` by applying `a1`.

As we already explained in section 1.5.2, the behaviour of a system is represented by a finite sequence of the form $s_0 \stackrel{a_0}{\rightarrow} s_1 \stackrel{a_1}{\rightarrow} ... \; s_n \stackrel{a_n}{\rightarrow} s_{n+1}$. Here s_i $(0 \leq i \leq n+1)$ are states, and a_i $(0 \leq i \leq n)$ are actions. Infinite behaviours are represented by all finite approximations. Sequence r is a correct behaviour or *run* iff s_0 is the initial state, actions are enabled, as expressed by the `Pre` predicate, and subsequent states are related by the `Effect` predicate.

We define runs in generic PVS theory `ConstructProt` with parameters `States`, `is`, `Actions`, `Pre` and `Effect`. This theory must be imported to define a particular protocol. Note that we abstract from the structure of the type `Actions`. A run r is formalized as a record with two fields: $states(r)$ is a finite sequence of states, and $actions(r)$ is a finite sequence of actions, where $states(r)$ has one more element then $actions(r)$. For the example run above, we have $states(r) = s_0 s_1 ... s_n s_{n+1}$ and $actions(r) = a_0 a_1 ... a_n$.

To represent a particular protocol, defined by `States`, `is`, `Actions`, `Pre` and `Effect`, we import theory `ConstructProt` with corresponding parameters. For a run r of this protocol, we say that $actions(r)$ is *a schedule, accepted by this protocol*, i.e. a protocol is a predicate on schedules:

$$S \in protocol \stackrel{def}{\Longleftrightarrow} \exists r : S = actions(r)$$

Given a particular protocol, theorem OrdConfEquiv from section 3.2 provides a sound and complete method for proving serializability. For a protocol expressed by a predicate *protocol* on schedules, our aim is to prove that each schedule S accepted by this protocol is ordered, i.e. has a conflict-preserving timestamp:

$\forall r : \exists TS : CPT(actions(r), TS)$ ProtOrd

After that, theorem OrdConfEquiv implies that the protocol indeed ensures conflict serializability:

$\forall S : protocol(S) \Longrightarrow Conf\_serializable(S)$ ProtConf

3.3.1 Two Phase Locking protocol

In PVS theory `Tpl` (see section 3.7.5), we specify the protocol that was informally described in section 2.2.1, following the four steps mentioned above.

1. `ActionNames`. In our model, locking is incorporated in read and write actions, and hence does not require a separate action. We only add an `unlock` action to unlock a data item which is locked in a shared or exclusive mode and a `downgrade` action which changes the mode of the lock from exclusive to shared.
   In auxiliary theory `AllActionNames`, the set of atomic actions `AllActionNames` is defined. It includes all actions of the 2PL protocol, as well as additional actions of two extensions of the 2PL protocol, which will be considered in section 3.6.
   ```
   AllActionNames : TYPE =
      {R, W, unlock, downgrade, Rrequest, Wrequest}
   ```
   In theory `Tpl` this set is restricted to the actions of the 2PL protocol using the predicate `ActionNames2PL`.
   ```
   at : VAR AllActionNames

   ActionNames2PL(at) : bool =
      R?(at) OR W?(at) OR unlock?(at) OR downgrade?(at)
   ```
   This predicate is used when we import the definition of the type `Actions` with corresponding parameters.
   ```
   IMPORTING Actions [(ActionsNames2PL), Transactions, DataItems]
   ```
2. We define `States2PL` by a record with three fields. The fields `xset` and `sset` map each transaction to a set of data items which this transaction locks in an exclusive and shared mode, respectively. The field `shrinking` consists of transactions which already entered the shrinking phase and therefore cannot issue any new locks. All transactions that are not in the shrinking phase are assumed to be in the growing phase, so we don't need to define a set of transactions that are still in the growing phase.
   ```
   States2PL : TYPE =
      [# xset : [Transactions -> setof[DataItems] ],
         sset : [Transactions -> setof[DataItems] ],
         shrinking : setof[Transactions] #]
   ```
   In the initial state, `is2PL`, all the data items are unlocked and no transaction is shrinking.
3. We only explain the parts of the `Pre2PL` predicate corresponding to read, unlock and downgrade actions. The part of `Pre2PL` corresponding to a read action requires that a transaction T may read a data item x only if it is allowed to make corresponding locks, i.e.:
   - No other transaction locks x in an exclusive mode.
   - If T is in the shrinking phase, then x must be already locked by T in a shared or exclusive mode (because a transaction in the shrinking phase is not allowed to issue any new locks).

In order to eliminate useless unlock and downgrade actions, the predicate `Pre2PL` allows an unlock action only if x is currently locked by T in a shared or exclusive mode, and a downgrade action only if x is currently locked by T in an exclusive mode. In PVS these parts of `Pre2PL` are denoted as follows:

```
Pre2PL(s1, a1) : bool =
LET x = dat(a1), T = tr(a1) IN
CASES act(a1) OF
   R: (FORALL T2 : xset(s1)(T2)(x) => T2 = T) &
      (shrinking(s1)(T) => (xset(s1)(T)(x) OR sset(s1)(T)(x))),
   W: ... ,
   unlock: xset(s1)(T)(x) OR sset(s1)(T)(x),
   downgrade: xset(s1)(T)(x)
ENDCASES
```

4. The `Effect2PL` predicate defines how the locks are updated by each action. Correctness of this assignment depends on the `Pre` predicate, and therefore these two predicates must be analyzed together. Here we only explain the effect of a read action. Suppose a state $s2$ is obtained from a state $s1$ by applying an action $a1 = (R, T, x)$. Two cases are possible:

   - If x belongs to $xset(s1)(T)$, i.e. is locked by T in an exclusive mode, then the `Pre` predicate will require that x is not locked in an exclusive mode by any other transaction (otherwise T is not allowed to read x). There is no need to issue any new locks, and the state is not changed.
   - If x does not belong to $xset(s1)(T)$, then T must lock x in a shared mode to perform a read action.

   This transformation is formally defined by the following predicate (note that the `WITH` construct is used to override a field or a value of a record or a function, respectively; so below x is added to the shared locks of T in $s1$):

```
IF xset(s1)(T)(x) THEN s2 = s1
ELSE s2 = s1 WITH [sset := sset(s1) WITH
                      [T := add(x, sset(s1)(T))] ENDIF
```

3.3.2 Timestamp Ordering Protocol

In PVS theory `Tso` (see section 3.7.6), we specify the protocol that is informally described in section 2.2.3. As already explained, in this model we avoid the need to formalize aborts of transactions by considering only schedules consisting of "successful" actions, i.e. actions which don't violate the timestamp order. We also eliminate a *Start* action of a transaction by incorporating it into the first read or write action of this transaction. Therefore we only need read and write actions.

1. `ActionNamesTSO : TYPE = { R, W }`
2. `StatesTSO` is defined by a record with four fields: a read timestamp `Rts` and a write timestamp `Wts` of data items, a timestamp `TTS` of transactions and a counter `counter`, all defined on natural numbers. In the initial state, `isTSO`, the value of all timestamps is 0. The initial value of the counter is 1, and it is incremented each time when a timestamp is assigned to a new transaction.

```
StatesTSO : TYPE = [# Rts : [DataItems -> nat],
                      Wts : [DataItems -> nat],
                      TTS : [Transactions -> nat],
```

```
counter : nat #]
```

3. Predicate `PreTSO` eliminates all actions that violate the timestamp order. Suppose in a state $s1$ a transaction T reads or writes a data item x. Three cases are possible.
   - If $TTS(s1)(T) = 0$, it means that T is entering the system by performing its first action. Therefore, the *new* timestamp of T will be greater than timestamps of all other transactions. Thus T is allowed to read or write x.
   - If $TTS(s1)(T) > 0$ and T reads x, then the timestamp of T must be greater or equal than the write timestamp of x:
     `TTS(s1)(T) >= Wts(s1)(x)`
   - If $TTS(s1)(T) > 0$ and T writes x, then the timestamp of T must be greater or equal than both the read timestamp of x and the write timestamp of x:
     `TTS(s1)(T) >= Rts(s1)(x) & TTS(s1)(T) >= Wts(s1)(x)`
4. Predicate `EffectTSO` expresses the proper adjustment of all timestamps and the counter. Suppose in a state $s1$ a transaction T reads or writes a data item x.
   Adjustment of the timestamps of transactions. Only the timestamp of T might be changed. It is only changed if $TTS(s1)(T) = 0$, i.e. T did not enter the system yet, and becomes equal to the current value of the counter.
   Adjustment of the timestamps of data items. Only the timestamp of x is changed. Two cases are possible.
   - T reads x. If $TTS(s1)(T) = 0$, the read timestamp of x becomes equal to the new timestamp of T, i.e. the current value of the counter. Note that in our PVS specifications the timestamps of transactions and data items are updated simultaneously, so we cannot explicitly use here the new timestamp of T. Otherwise, the read timestamp of x becomes equal to the maximum of the current read timestamp of x and the current timestamp of T. Thus `Rts(s2)(x)` is defined by
     ```
     IF TTS(s1)(T) = 0 THEN counter(s1)
     ELSE max(Rts(s1)(x), TTS(s1)(T)) ENDIF
     ```
   - T writes x. If $TTS(s1)(T) = 0$, the read timestamp of x becomes equal to the new timestamp of T, i.e. the current value of the counter. Otherwise, it becomes equal to the current timestamp of T.

   Adjustment of the counter. The counter is incremented only if $TTS(s1)(T) = 0$, i.e. T is a new transaction. Thus `counter(s2)` is defined by
   ```
   IF TTS(s1)(T) = 0 THEN counter(s1) + 1 ELSE counter(s1) ENDIF
   ```

3.4 Verification of the 2PL and TSO protocols

As shown in section 3.3, the verification of conflict serializability for a particular protocol may be reduced to proving theorem ProtOrd, which expresses the existence of a conflict preserving timestamp for each schedule accepted by the protocol. We successfully applied this method to the machine-checked verification of the TSO protocol and the 2PL protocol.

3.4.1 The 2PL protocol

Suppose a schedule S is accepted by the 2PL protocol. In theory `Tpl`, we define the desired CPT, called $TS2PL(S)$, as follows. If a transaction T has performed some unlock or downgrade actions

in S, i.e. entered the shrinking phase, we take as a timestamp of T the index of the *first* shrinking action. If T was active in S, but has not yet performed any unlock or downgrade actions, we take as a timestamp of T the index of its *last* action in S. Otherwise, we take 0 as a timestamp of T. The PVS definition, in which $IndexSet$ is the set that was defined in section 3.2 and $ShrinkSet$ defines the set of indices of unlock and downgrade actions for a particular transaction, is as follows:

$$ShrinkSet(S,T) = \{i \mid i < length(S0)\ \&\ T = tr(S0(i))\ \&\ (act(S(i)) = unlock \vee act(S(i)) = downgrade)\}$$

$$TS2PL(S)(T) = \textit{if } ShrinkSet(S,T) \neq \emptyset \textit{ then } minimum(ShrinkSet(S,T)) \textit{ else } \textit{if } IndexSet(S,T) \neq \emptyset \textit{ then } maximum(IndexSet(S,T)) \textit{ else } 0$$

Using the interactive theorem prover of PVS, we proved the following theorem CPT2PL, which obviously implies theorem ProtOrd.

$$\forall r : CPTinj(actions(r), TS2PL(actions(r))) \qquad \text{CPT2PL}$$

Proof of theorem CPT2PL. Using the definitions of minimum and maximum, we can easily prove that $TS2PL(S)$ is injective for each schedule S. Now let $S = actions(r)$ for an arbitrary run r. Suppose that a pair of transactions $(T1, T2)$ belongs to $Conflict(S)$. We need to prove $TS2PL(S)(T1) < TS2PL(S)(T2)$. The definition of $Conflict(S)$ implies that there exist indexes i and j and a data item x such that

$$\begin{array}{l} i < j\ \&\ j < length(actions(r))\ \& \\ T1 \neq T2\ \&\ T1 = tr(actions(r)(i))\ \&\ T2 = tr(actions(r)(j))\ \& \\ x = dat(actions(r)(i))\ \&\ x = dat(actions(r)(j))\ \& \\ ((act(actions(r)(i)) = W\ \&\ act(actions(r)(j)) = W) \vee \\ \ (act(actions(r)(i)) = R\ \&\ act(actions(r)(j)) = W) \vee \\ \ (act(actions(r)(i)) = W\ \&\ act(actions(r)(j)) = R)) \end{array}$$

Suppose $act(actions(r)(j)) = W$. It implies that $act(actions(r)(i)) = W$ or $act(actions(r)(i)) = R$. We can notice that in state $states(r)(j+1)$, transaction $T2$ holds an exclusive lock on x. However, in state $states(r)(i+1)$, $T2$ does not hold an exclusive lock on x. Indeed, since $T1$ reads or writes x in action $actions(r)(i)$, in state $states(r)(i+1)$ it holds an exclusive or shared lock on x, and this lock is not compatible with an exclusive lock by $T2$. It implies that at some moment between i and $j+1$, $T2$ must obtain a *new* exclusive lock on x. Predicate $Effect2PL$ implies that such lock may be obtained only by performing a write action on x. Therefore there exists some index m such that $i < m \leq j$, $actions(r)(m) = (W, T2, x)$ and x does not belong to $xset(states(r)(m))(T2)$. Since a transaction in the shrinking phase cannot issue any new locks, it implies that in state $states(r)(m)$, $T2$ is still in the growing phase. We will prove that $TS2PL(S)(T2) \geq m$. Suppose $TS2PL(S)(T2) < m$. It implies that $T2$ performed an unlock or downgrade action with an index $m' < m$ and entered the shrinking phase. The definition of $Effect2PL$ implies that a transaction can never return from the shrinking phase to the growing phase. Contradiction, because at the state with index $m > m'$, T is still in the growing phase.

We can notice that in state $states(r)(m+1)$, a transaction $T1$ does not hold any locks on x. Indeed, since $T2$ writes x in action $actions(r)(m)$, in state $states(r)(m+1)$ it holds an exclusive

lock on x, and this lock is not compatible with any lock by $T1$. However, in state $states(r)(i+1)$, $T1$ holds an exclusive or shared lock on x. Predicate $Effect2PL$ implies that at some moment between i and $m+1$, $T1$ must *unlock* x, i.e. there exists some index l such that $i < l \leq m$ and $actions(r)(l) = (unlock, T1, x)$. We also notice that $l < m$, because $actions(r)(m)$ is a write action. It is easy to see that $TS2PL(S)(T1) \leq l$. Indeed, we defined $TS2PL(S)(T1)$ as the index of the *first* unlock or downgrade action.

Therefore we proved that if $act(actions(r)(j)) = W$ and ($act(actions(r)(i)) = W$ or $act(seq(S)(i)) = R$), then there exist l and m such that $TS2PL(S)(T1) \leq l, l < m$ and $TS2PL(S)(T2) \geq m$. It implies that $TS2PL(S)(T1) < TS2PL(S)(T2)$. We use similar reasoning to prove that if $act(actions(r)(i)) = W$ and $act(actions(r)(j)) = R$, then again $TS2PL(S)(T1) < TS2PL(S)(T2)$. Since these two cases are the only possible ones, this completes the proof of theorem CPT2PL.

3.4.2 The TSO protocol

To verify the TSO protocol, we use a slightly different technique: timestamps are defined not for schedules, but for runs corresponding to these schedules. For a run r, timestamp $TSTSO$ is defined as the timestamp of transactions in the last state of r:

$$TSTSO(r) = TTS(last(states(r)))$$

Using the interactive theorem prover of PVS, we proved that $TSTSO(r)$ is an injective CPT for schedule $actions(r)$, which obviously implies theorem ProtOrd.

$$\forall r : CPTinj(actions(r), TSTSO(r)) \qquad \text{CPTTSO}$$

Proof of theorem CPTTSO. First we prove injectivity of $TSTSO(r)$ with respect to $actions(r)$. This fact is also used in the proof that $TSTSO(r)$ is conflict-preserving for $actions(r)$. The proof of injectivity is based on the following invariants (1), (2) and (3), which are all proved by induction on the length of $actions(r)$. In invariant (2), $lps(r)$ denotes the *longest proper subrun* of a run r ($length(actions(r)) > 0$), i.e. if $r = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_n \xrightarrow{a_n} s_{n+1}$, then $lps(r) = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_n$. The intuitive meaning of this invariant is as follows: once a transaction becomes active in a run, its timestamp $TSTSO$ is not changed by any consecutive actions. In invariant (3), $lst(r)$ denotes the last state of a run r.

$$\forall r, T : active(actions(r))(T) \Longleftrightarrow TSTSO(r)(T) > 0 \tag{1}$$

$$\forall r, T : (length(actions(r)) > 0 \;\&\; active(actions(lps(r)))(T)) \Longrightarrow TSTSO(r)(T) = TSTSO(lps(r))(T) \tag{2}$$

$$\forall r, T : TSTSO(r)(T) < counter(lst(r)) \tag{3}$$

Suppose $T1 \neq T2$ and both $T1$ and $T2$ belong to $active(actions(r))$. To show injectivity of $TSTSO(r)$, we need to prove $TSTSO(r)(T1) \neq TSTSO(r)(T2)$. The proof is also done by induction on $length(actions(r))$. If $actions(r)$ is empty, then $active(actions(r))$ is also empty. Suppose now that the formula is proved for any length not greater than k, and let $length(actions(r)) = k+1$. Then $actions(r)$ can be represented as $actions(lps(r))\ a$ for some action a, and we have

$length(actions(lps(r))) = k$. If both $T1$ and $T2$ also belong to $active(actions(lps(r)))$, then by the induction hypothesis, $TSTSO(lps(r))(T1) \neq TSTSO(lps(r))(T2)$. By applying (2) to both $T1$ and $T2$, we obtain $TSTSO(r)(T1) \neq TSTSO(r)(T2)$. Suppose now that one of $T1$ and $T2$, for instance $T2$, does not belong to $active(actions(lps(r)))$. It is obvious that action a is performed by $T2$, and that $T1 \in active(actions(lps(r)))$. Invariant (1) implies $TSTSO(lps(r))(T2) = 0$. The definition of $EffectTSO$ now implies $TSTSO(r)(T2) = counter(lst(lps(r)))$, regardless of whether a is a read or a write action. Invariant (3) implies $TSTSO(lps(r))(T1) < counter(lst(lps(r)))$. By applying invariant (2) to $T1$, we obtain $TSTSO(r)(T) < counter(lst(r))$, and therefore $TSTSO(r)(T1) < TSTSO(r)(T2)$.

Two additional invariants are needed to prove that $TSTSO(r)$ is conflict-preserving for $actions(r)$, which are also proved by induction on the length of $actions(r)$. The informal definition of a write timestamp in section 2.2.3 indicates that a write timestamp of a data item x is not smaller than timestamp of any transaction that wrote x. This is formally established by invariant (4). This invariant is defined using the set of transactions $Wset(r, x)$, which includes all transactions that wrote x in a run r.

$$Wset(r, x) = \{ T \mid \exists i : i < length(actions(r)) \ \& \ actions(r)(i) = (W, T, x) \}$$

$$\forall r, T, x : T \in Wset(r, x) \Longrightarrow TSTSO(r)(T) \leq Wts(lst(r))(x) \tag{4}$$

$$\forall r, x : Wts(lst(r))(x) < counter(lst(r)) \tag{5}$$

We prove now that $TSTSO(r)$ is conflict-preserving for $actions(r)$ by induction on $length(actions(r))$. If $actions(r)$ is empty, then any timestamp is conflict-preserving for $actions(r)$. Suppose now that the formula is proved for any length not greater than k, and let $length(actions(r)) = k + 1$. Then $actions(r)$ can be represented as $actions(lps(r))a$ for some action a, and we have $length(actions(lps(r))) = k$. Suppose that a pair of transactions $(T1, T2)$ belongs to $Conflict(actions(r))$. If the pair $(T1, T2)$ also belongs to $Conflict(actions(lps(r)))$, then by the induction hypothesis, $TSTSO(lps(r))(T1) < TSTSO(lps(r))(T2)$. By applying (2) to both $T1$ and $T2$, we obtain $TSTSO(r)(T1) < TSTSO(r)(T2)$. Suppose now $(T1, T2) \notin Conflict(actions(lps(r)))$. By definition of $Conflict$, it is obvious that a is performed either by $T1$ or $T2$. Without loss of generality, we can assume that a is performed by $T2$. We consider only the case when a is a read action. The case of a write action has a similar proof, however, this proof uses an additional invariant, similar to invariant (4), which expresses that a read timestamp of a data item x is not smaller than timestamp of any transaction that read x.

It is obvious that $T1 \in active(actions(lps(r)))$. Since only a write action may conflict with a read action by $T2$, it is easy to see that $T1 \in Wset(lps(r), x)$. By applying invariants (2) and (4) to $T1$, we obtain $TSTSO(r)(T1) \leq Wts(lst(lps(r)))(x)$. Two cases are possible.

- If $TTS(lps(r))(T2) > 0$, the definition of $PreTSO$ implies $TSTSO(lps(r))(T2) \geq Wts(lst(lps(r)))(x)$. Invariants (1) and (2) now imply $TSTSO(r)(T2) \geq Wts(lst(lps(r)))(x)$. Thus $TSTSO(r)(T1) \leq TSTSO(r)(T2)$. But *injectivity* of $TSTSO(r)$ implies $TSTSO(r)(T1) < TSTSO(r)(T2)$.
- If $TTS(lps(r))(T2) = 0$, the definition of $EffectTSO$ implies $TSTSO(r)(T2) = counter(lst(lps(r)))$. By applying invariant (5) to $T1$, we obtain $TSTSO(r)(T1) < counter(last(states(lpr(r))))$. Thus $TSTSO(r)(T1) < TSTSO(r)(T2)$.

3.5 Extensions of serializable protocols

Although in section 3.2 we have formulated a complete method to prove conflict serializability, it is not always easy to find a conflict-preserving timestamp function for any schedule (and to prove that it actually is one). Observing that many protocols can be seen as extensions of a basic protocol (such as 2PL or TSO), we investigate how we can obtain serializability of an extension from serializability of a basic protocol. First we define the notion of an extension more precisely.

We say that protocol `NewProt` is an *extension* of protocol `OldProt` iff

- `OldActionNames`, the set of atomic actions of `OldProt`, is a subset of `NewActionNames`, the set of atomic actions of `NewProt`.
- `ExtendedStates`, the control part of `NewProt`, is obtained from `OldStates`, the control part of `OldProt`, by adding a record `ext` of type `Extension`, representing the added control information:
  `ExtendedStates : TYPE = [# old : OldStates, ext : Extension #]`

Our goal is to prove that if `OldProt` ensures conflict serializability and extension `NewProt` satisfies certain conditions, then `NewProt` also ensures conflict serializability. Below we derive the required conditions during the construction of the proof.

Let $Conf\_ser\_Old$ and $Conf\_ser\_New$ be instances of the predicate $Conf\_serializable$ for schedules from `OldProt` and `NewProt`, respectively. Our aim is to prove the following theorem.

$$OldProt \in Conf\_ser\_Old \Longrightarrow NewProt \in Conf\_ser\_New \qquad \text{MainTheorem}$$

Proof. Suppose `OldProt` ensures conflict serializability and schedule $NewS$ is accepted by `NewProt`. The proof that $NewS$ is conflict serializable consists of two steps.

Step 1 We prove that $NewS$ is a *refinement* of some schedule $OldS$, accepted by `OldProt`, i.e. it is obtained from $OldS$ by adding some actions. To construct $OldS$, we simply remove from $NewS$ all added actions, i.e. all actions that do not occur in $OldActionNames$. The result is formally defined by function $Extract(NewS)$. Note that we don't remove any read or write actions, because R and W belong to $OldActionNames$. The following theorem expresses that $Extract(NewS)$ is accepted by `OldProt`.

$$NewProt(NewS) \Longrightarrow OldProt(Extract(NewS)) \qquad \text{ExtractOld}$$

As we show below, the proof of this theorem reveals the required correctness conditions. Since `OldProt` is conflict serializable, theorem ExtractOld implies $Conf\_ser\_Old(Extract(NewS))$.

Step 2 If $Extract(NewS)$ is conflict serializable, then also $NewS$:

$$Conf\_ser\_Old(Extract(NewS)) \Longrightarrow Conf\_ser\_New(NewS) \qquad \text{ConfNewOld}$$

The proof of this theorem uses *completeness* of our verification method for conflict serializability. Since $Extract(NewS)$ is conflict serializable, theorem ConfOrd implies that $Extract(NewS)$ has a conflict preserving timestamp (CPT). Let TS denote this timestamp. We prove that TS is also a CPT for $NewS$. Indeed, suppose that transactions $T1$ and $T2$ are conflicting in $NewS$. It implies that $NewS$ includes two conflicting actions $a1$ and $a2$ by $T1$ and $T2$ respectively. Both $a1$ and $a2$ can be only read or write actions, and therefore $act(a1)$ and $act(a2)$ belong to $OldActionNames$. The

definition of $Extract$ implies that $Extract(NewS)$ also includes $a1$ and $a2$, in the same order. The definition of conflict relation now implies that $T1$ and $T2$ are also conflicting in $Extract(NewS)$. TS is a CPT for $Extract(NewS)$, and it implies $TS(T1) < TS(T2)$. We proved that if transactions $T1$ and $T2$ are conflicting in $NewS$, then $TS(T1) < TS(T2)$. Therefore TS is indeed a CPT for $NewS$, i.e. $NewS$ is ordered. Theorem OrdConf now implies that $NewS$ is conflict serializable.

Theorem ConfNewOld implies $Conf\_ser\_New(NewS)$, and this completes the proof of theorem MainTheorem. **End Proof**

It remains to prove theorem ExtractOld and to derive the required correctness conditions.

Proof of theorem ExtractOld
Assume $NewProt(NewS)$. Then there exists a run $NewR = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \; s_n \xrightarrow{a_n} s_{n+1}$ of `NewProt` such that $NewS = actions(NewR)$, i.e. $NewS = a_0 a_1 ... a_n$. Let $a'_0 a'_1 ... a'_k$ be the sequence obtained from $a_0 a_1 ... a_n$ by removing all actions that are not in $OldActionNames$, i.e., $Extract(NewS) = a'_0 a'_1 ... a'_k$.

To prove $OldProt(Extract(NewS))$, we construct a run $s'_0 \xrightarrow{a'_0} s'_1 \xrightarrow{a'_1} ... \; s'_k \xrightarrow{a'_k} s'_{k+1}$ of `OldProt`. This run, denoted by $ExtractR(NewR)$, is obtained by removing from run $NewR$ any action that is not in $OldActionNames$ and its successor state. Moreover, we take only the field "old" of the remaining states. Since $Extract$ and $ExtractR$ both remove the same actions (those with action names not in $OldActionNames$), observe that $actions(ExtractR(NewR)) = Extract(actions(NewR)) = Extract(NewS) = a'_0 a'_1 ... a'_k$. Hence, it remains to prove that $OldR = ExtractR(NewR)$ is a run of `OldProt`.

It is more convenient to prove the following, stronger statement, consisting of two parts (where lst was introduced in section 3.4.2):

(i) $ExtractR(NewR)$ is a run of `OldProt` and

(ii) $lst(ExtractR(NewR)) = old(lst(NewR))$.

The proof proceeds by induction on the length of $actions(NewR)$.

Basic Step Let $length(actions(NewR)) = 0$. Then $NewR = NewInitState$ and, by definition of $ExtractR$, $ExtractR(NewR) = old(NewInitState)$. Hence, $ExtractR(NewR)$ is a run if $old(NewInitState)$ is equal to the initial state of `OldProt`. Then also **(ii)** is satisfied. This leads to the first condition.

Condition 1 $old(NewInitState) = OldInitState$

Induction Step Let $length(actions(NewR)) = k+1$. Then $NewR = lps(NewR) \xrightarrow{a} lst(NewR)$ for some action a, where lps was defined in section 3.4.2. Let's denote $NewR1 = lps(NewR)$. We distinguish two cases.

$act(a) \notin OldActionNames$ Then $ExtractR(NewR) = ExtractR(NewR1)$. For part **(i)**, recall that by the induction hypothesis $ExtractR(NewR1)$, and hence also $ExtractR(NewR)$, is a run of `OldProt`.
For **(ii)**, note that $lst(ExtractR(NewR)) = lst(ExtractR(NewR1))$ $= old(lst(NewR1))$, using the induction hypothesis. To obtain $old(lst(NewR1)) = old(lst(NewR))$, we introduce a condition expressing that if we apply a newly added action a to an extended state $es1$, then the old part of it should not change.

Condition 2
$act(a) \notin OldActionNames \Longrightarrow$
$(NewEffect(es1, a, es2) \Longrightarrow old(es1) = old(es2))$

$act(a) \in OldActionNames$ By definition of $ExtractR$, we have in this case
$ExtractR(NewR) = ExtractR(NewR1) \xrightarrow{a} old(lst(NewR))$.
By the induction hypothesis, part **(ii)**, we have
$lst(ExtractR(NewR1)) = old(lst(NewR1))$. (*)
To prove **(i)**, note that $ExtractR(NewR)$ is a run of `OldProt` if the following two conditions are satisfied.

- a is allowed in the last state of $ExtractR(NewR1)$, that is,
  $OldPre(lst(ExtractR(NewR1)), a)$.By (*), it remains to prove
  $OldPre(old(lst(NewR1)), a)$.Since a is allowed in the last state of $NewR1$, we have that $NewPre(lst(NewR1), a)$.Hence it is sufficient to require that any old action a which is allowed in an extended state es according to $NewPre$, is also allowed in the $old(es)$ according to $OldPre$.

  Condition 3 $act(a) \in OldActionNames \Longrightarrow$
  $(NewPre(es, a) \Longrightarrow OldPre(old(es), a))$

- $old(lst(NewR))$ is obtained from $lst(ExtractR(NewR1))$ by applying a to it, i.e. $OldEffect(lst(ExtractR(NewR1)), a, old(lst(NewR)))$. By (*), it remains to prove $OldEffect(old(lst(NewR1)), a, old(lst(NewR)))$.
  Since $lst(NewR)$ is obtained from $lst(NewR1)$ by applying a to it, we have $NewEffect(lst(NewR1), a, lst(NewR)) = TRUE$. Hence it is sufficient to require, for any old action oa, that $NewEffect$ must transform the old part of an extended state $es1$ in the same way $OldEffect$ does.

  Condition 4 $act(a) \in OldActionNames \Longrightarrow$
  $(NewEffect(es1, a, es2) \Longrightarrow OldEffect(old(es1), a, old(es2)))$

This proves **(i)**. To prove **(ii)**, observe that by the definition of $ExtractR$, in this case $lst(ExtractR(NewR)) = old(lst(NewR))$.

This completes the induction step and also the proof of ExtractOld. **End Proof**

To implement extensions in PVS, we define a general PVS theory `ExtendProt`. As parameters, it has all types and predicates that are needed to define `OldProt` and `NewProt`. Theorem MainTheorem, which establishes the main result, is proved in `ProtExtend`. The conditions 1 through 4 mentioned above are added to this theory by including them as four *assumptions*. If any theory imports `ExtendProt` then a proof of these assumptions is required.

Given a conflict serializable protocol `OldProt`, we can prove serializability of an extension `NewProt` by importing theory `ExtendProt`. This requires a proof of the four assumptions. Once they have been proved, we can use MainTheorem, and obtain conflict serializability of `NewProt`.

3.6 Two extensions of the 2PL protocol

We applied our method to the basic 2PL protocol, verified in section 3.4.1. This protocol has been extended in two steps, leading to a more realistic protocol which is serializable by construction.

3.6.1 Adding sequences of waiting transactions

In the first extension, specified in theory `TplExt1` (see section 3.7.8), we associate with each data item a sequence of transactions that are waiting for the permission to read or write this data item. If a transaction is not allowed to read or write a data item x immediately (because it is currently locked in an incompatible mode), the corresponding action is inserted into the sequence of x. After x becomes available, a postponed action from the sequence of x may be executed.

The operation of inserting an action into a sequence is modeled by a *read request action* (`Rrequest`) and a *write request action* (`Wrequest`). The full set of atomic actions is represented by PVS type `AllActionNames` (see section 3.3.1).

The extension of the state consists of function `Extension1` that maps each data item to a finite sequence, consisting of read and write requests performed by certain transactions; in the initial state all sequences are empty. The extended state is called `States1`. The PVS definition is as follows (where `ExtendStates` is the auxiliary theory from section 3.7.7 in which extensions of states are defined):

```
ReadWrite : TYPE = { at : AllActionNames | R?(at) OR W?(at) }

Requests : TYPE = [# tr : Transactions, act  : ReadWrite #]

Extension1 : TYPE = [ DataItems -> finite_sequence[Requests]]

IMPORTING ExtendStates[ States2PL, Extension1 ]

States1 : TYPE = ExtendedStates[States2PL, Extension1]

NewIS1 : States1 = (# old := is2PL, ext := LAMBDA x : empty_seq #)
```

New precondition `NewPre1` ensures that not only preconditions defined by `Pre2PL` are satisfied, but also some additional preconditions, expressed by predicate `AddPre1`. In the current extension, we don't require that actions of the waiting sequences are executed on a first-in, first-out basis. A transaction, waiting in a sequence of any data item, cannot perform any new actions (including read and write requests). Therefore, as defined by `AddPre1`, a transaction T is allowed to read or write a data item x in an extended state with extension $e1$ if one of the following conditions is true:

- T does not belong to the set of waiting transactions, denoted by $waiting(e1)$, and the sequence of x, represented by $e1(x)$, is empty,
- $e1(x)$ is nonempty, and includes the corresponding read or write request by T.

New effect predicate `NewEffect1` transforms the state in the same way `Effect2PL` does for old actions, it leaves the old part of the state unchanged for added actions, and includes an additional predicate `AddEffect1` to define how to insert and remove requests from the waiting sequences. If a transaction T reads or writes a data item x, and T does not belong to $waiting(e1)$, then predicate `AddEffect1` does not change any sequence. Otherwise, predicate `AddPre1` implies that $e1(x)$ is nonempty and includes the corresponding read or write request by T. Therefore, `AddEffect1` removes the corresponding request from $e1(x)$.

3.6.2 Adding priorities to waiting transactions

We define a second-level extension of the 2PL protocol by extending the first-level extension above such that the processing of transactions depends on their *priorities*. A priority function PR assigns to

each transaction T its priority $PR(T)$ from the set of natural numbers.

We also introduce the notion of *urgent* transactions, which is important for real-time protocols. Assume given a natural number U. Transaction T is called *urgent with respect to* U, if $PR(T) \geq U$. We define our protocol in a new theory, such that its set of parameters includes PR and U. Changing PR and U, we obtain different protocols. Therefore our theory actually defines a class of protocols.

In this extension, states are extended with a pair (PR, U). The set of actions is not changed. Instead, some additional restrictions are introduced on the order, in which existing actions are performed. The aim of these new restrictions is to ensure that "urgent" transactions obtain immediate access to data items, whereas that non-urgent transactions should be served on a first-in, first-out basis.

Suppose a data item x has a sequence q. Let $MaxPriority(q)$ denote the maximum of priorities of transactions that have requests in q. If $MaxPriority(q) > U$, it means that q includes requests from urgent transactions, and therefore we must execute one of the urgent transactions with the highest priority $MaxPriority(q)$. Otherwise, we may execute the first-inserted request of the waiting sequence. The complete specification of this extension is given in theory `TplExt2` (see section 3.7.8).

3.6.3 Correctness of the obtained extensions

After importing the theory `ExtendProt` with corresponding parameters for both protocols, it turned out to be very easy to prove that our four assumptions are satisfied for both protocols. Therefore our extensions indeed ensure conflict serializability.

Note that one may satisfy the conflict serializability condition by not accepting any schedule, i.e. aborting or indefinitely delaying all actions submitted to the scheduler. Therefore, we additionally show that for every valid schedule in the initial protocol there is a representative in the extended protocol. For the first extension of the 2PL protocol presented above, it is easy to see that for every schedule S in the 2PL protocol there is a representative S' in the extension, which consists of the same actions. Let S' be a schedule where a transaction never tries to read or write a data item if it is not immediately available; then all sequences of requests are always empty and S' is indeed accepted by the extension. The same holds for the second extension of the 2PL protocol.

3.7 PVS specifications

In this section, we include PVS theories that contain all important definitions and theorems. These theories are valid although not complete. We don't include less important definitions and most lemmas because of space considerations. For technical reasons (e.g., some of these theories will be later imported by PVS specifications from chapter 4), the order of the definitions is slightly different than in the main part of the chapter.

3.7.1 Method for verification of conflict serializability

```
Actions [ActionNames, Transactions, DataItems: TYPE+] : THEORY
BEGIN

Actions : TYPE = [# act : ActionNames,
                    tr  : Transactions,
                    dat : DataItems #]
```

```
END Actions



fin_seq [T: TYPE] : THEORY
BEGIN

IMPORTING finite_sequences[T]

finseq : VAR finite_sequence
t1 : VAR T

prefix(finseq | length(finseq) > 0) : finite_sequence =
 (# length := length(finseq) - 1,
    seq := (LAMBDA (n:below[length(finseq) - 1]): seq(finseq)(n)) #)

last(finseq | length(finseq) > 0) : T = seq(finseq)(length(finseq) - 1)

first(finseq | length(finseq) > 0) : T = seq(finseq)(0)

one(t1) : finite_sequence =
   (# length := 1, seq := (LAMBDA (k : below[1]): t1) #)

END fin_seq



ConfOrd [Transactions, DataItems, ActionNames : TYPE+,
   R, W: ActionNames] : THEORY
BEGIN

IMPORTING Actions [ActionNames, Transactions, DataItems]

Schedules : TYPE = finite_sequence[Actions]

T, T1, T2 : VAR Transactions
i, j, k, l, m, n : VAR nat
S, S0, S1, S2 : VAR Schedules

serial(S) : bool = FORALL T1, T2, i, j, k :
   (i < j & j < k & k < length(S) &
    T1 = tr(S(i)) & T2 = tr(S(j)) & T1 = tr(S(k))) => T1 = T2

act_conflict(a1, a2) : bool =
   tr(a1) = tr(a2) OR
      (dat(a1) = dat(a2) &
      ((act(a1) = W & act(a2) = W) OR (act(a1) = W & act(a2) = R) OR
       (act(a1) = R & act(a2) = W)))

elem_equiv(S1, S2) : bool =
   length(S1) = length(S2) &
   EXISTS k : k < length(S1) - 1 &
      seq(S2) = (seq(S1) WITH [(k) := S1(k + 1), (k + 1) := S1(k)]) &
```

```
      (NOT act_conflict(S1(k), S1(k + 1)))

fs : VAR finite_sequence[Schedules]

IMPORTING fin_seq[Schedules]

conf_equiv0(S1, S2, (k | k > 0)) : bool = EXISTS fs :
   length(fs) = k + 1 & S1 = first(fs) & S2 = last(fs) &
   FORALL i : i < length(fs) - 1 => elem_equiv(fs(i), fs(i + 1))

conf_equiv(S1, S2) : bool =
   (S1 = S2) OR EXISTS k : k > 0 & conf_equiv0(S1, S2, k)

Conf_serializable(S) : bool = EXISTS S0 : serial(S0) & conf_equiv(S, S0)

Conflict(S)(T1, T2) : bool = (T1 /= T2) &
   EXISTS i, j : i < j & j < length(S) &
      T1 = tr(S(i)) & T2 = tr(S(j)) & act_conflict(S(i), S(j))

Timestamp : TYPE = [Transactions -> nat]

TS : VAR Timestamp

CPT(S, TS) : bool =
   FORALL T1, T2 : Conflict(S)(T1, T2) => TS(T1) < TS(T2)

Ordered(S) : bool = EXISTS TS : CPT(S, TS)

active(S) : setof[Transactions] =
   { T | EXISTS i : i < length(S) & T = tr(S(i)) }

inj(S, TS) : bool = FORALL T1, T2 :
   (active(S)(T1) & active(S)(T2) & T1 /= T2) => TS(T1) /= TS(T2)

CPTinj(S, TS) : bool = CPT(S, TS) & inj(S, TS)

OrderedI(S) : bool = EXISTS TS : CPTinj(S, TS)

cpt_lem : LEMMA (conf_equiv(S1, S2) & CPTinj(S2, TS)) => CPTinj(S1, TS)

IndexSet(S, T) : setof[nat] = { i | i < length(S) & T = tr(S(i)) }

ind_lem : LEMMA active(S)(T) => nonempty?(IndexSet(S, T))

TS0(S) : Timestamp =
   LAMBDA T : IF active(S)(T) THEN choose(IndexSet(S, T)) ELSE 0 ENDIF

serial_lem : LEMMA serial(S) => CPTinj(S, TS0(S))

ConfOrd : THEOREM Conf_serializable(S) => Ordered(S)

END ConfOrd
```

```
OrdConf [Transactions, DataItems, ActionNames : TYPE+,
   R, W: ActionNames ] : THEORY
BEGIN

IMPORTING ConfOrd [Transactions, DataItems, ActionNames, R, W ]

PN : TYPE = [nat, nat]

IMPORTING finite_sets_card_def[PN]

S, S0, S1, S2 : VAR Schedules
i, j, k, l, m, n, L, M : VAR nat
TS : VAR Timestamp
pn, pn1, pn2, x1, x2, y : VAR PN

dset(S, TS) : setof[PN] =
   { pn | PROJ_1(pn) < PROJ_2(pn) & PROJ_2(pn) < length(S) &
          TS(tr(S(PROJ_1(pn)))) > TS(tr(S(PROJ_2(pn)))) }

dfinite : LEMMA is_finite[PN](dset(S, TS))

d(S, TS) : nat = card(dset(S, TS))

swap(S1, (k | k + 1 < length(S1))) : Schedules =
   (# length := length(S1),
      seq := (seq(S1) WITH [(k) := S1(k + 1), (k + 1) := S1(k)]) #)

swapset2(S1, TS) : setof[Schedules] =
   { S2 | EXISTS k : k + 1 < length(S1) &
             TS(tr(S1(k))) > TS(tr(S1(k + 1))) & S2 = swap(S1, k) }

swap_lem : LEMMA d(S1, TS) > 0 => nonempty?[Schedules](swapset2(S1, TS))

swap2(S1, (TS | d(S1, TS) > 0)) : Schedules = choose(swapset2(S1, TS))

dset2(S, TS, k) : finite_set[PN] = remove((k, k + 1), dset(S, TS))

card_lem: LEMMA (k + 1 < length(S1) & TS(tr(S1(k))) > TS(tr(S1(k + 1))) &
   S2 = swap(S1, k)) => card(dset2(S1, TS, k)) = card(dset(S2, TS))

card_lem2 : LEMMA
   (d(S1, TS) > 0 & S2 = swap2(S1, TS)) => d(S2, TS) = d(S1, TS) - 1

inj_lem : LEMMA (d(S, TS) = 0 & inj(S, TS)) => serial(S)

FS(S, TS) : RECURSIVE finite_sequence[Schedules] =
   IF d(S, TS) = 0 THEN one(S) ELSE o(one(S), FS(swap2(S, TS), TS)) ENDIF
      MEASURE d(S, TS)

equiv_lem : LEMMA (k = d(S, TS) & CPT(S, TS)) =>
   (FORALL i : i < k =>
```

```
       elem_equiv(seq(FS(S, TS))(i), seq(FS(S, TS))(i + 1)))

OrdIConf : THEOREM OrderedI(S) => Conf_serializable(S)

END OrdConf
```

3.7.2 View serializability

```
ConfView [Transactions, DataItems, ActionNames : TYPE+,
   R, W: ActionNames ] : THEORY
BEGIN

IMPORTING OrdConf [Transactions, DataItems, ActionNames, R, W ]

T, T1, T2   : VAR Transactions
i, j, k, l, m, n   : VAR nat

T0, TF : Transactions

Schedules2 : TYPE = { s0 : Schedules |
   FORALL i : i < length(s0) => (tr(s0(i)) /= T0 & tr(s0(i)) /= TF) }

S, S0, S1, S2 : VAR Schedules2
x, y : VAR DataItems
a1, a2, a3 : VAR Actions

Writes(a1, x) : bool = (act(a1) = W & dat(a1) = x)

Writeof(a1, x, T) : bool = (act(a1) = W & dat(a1) = x & tr(a1) = T)

Readof(a1, x, T) : bool = (act(a1) = R & dat(a1) = x & tr(a1) = T)

Reads_from(T1, x, T2, S) : bool =
 IF T1 = T0 THEN
     IF T2 = TF THEN
         (FORALL k : k < length(S) => (NOT Writes(S(k), x)))
     ELSE (EXISTS m : m < length(S) & Readof(S(m), x, T2) &
             (FORALL k : k < m => (NOT Writes(S(k), x))))
     ENDIF
 ELSE IF T2 = TF THEN
       (EXISTS l : l < length(S) & Writeof(S(l), x, T1) &
         (FORALL k : (l < k & k < length(S)) => (NOT Writes(S(k), x))))
      ELSE (EXISTS l, m : l < m & m < length(S) &
             Writeof(S(l), x, T1) & Readof(S(m), x, T2) &
             (FORALL k : (l < k & k < m) => (NOT Writes(S(k), x))))
      ENDIF
 ENDIF

view_equiv(S1, S2) : bool = FORALL T1, x, T2 :
   Reads_from(T1, x, T2, S1) = Reads_from(T1, x, T2, S2)

View_serializable(S) : bool = EXISTS S0 : serial(S0) & view_equiv(S, S0)
```

```
reads_lem4 : LEMMA (elem_equiv(S1, S2) & T1 /= T0 & T2 /= TF &
   Reads_from(T1, x, T2, S1)) => Reads_from(T1, x, T2, S2)

reads_induct : LEMMA (k = n + 1 & conf_equiv0(S1, S2, k)) =>
   Reads_from(T1, x, T2, S1) = Reads_from(T1, x, T2, S2)

ConfViewStronger : THEOREM Conf_serializable(S) => View_serializable(S)

END ConfView
```

3.7.3 Acyclic conflict relations

```
AssignInj [Tp : TYPE] : THEORY
BEGIN

Ord : TYPE =
   { ord0 : pred[[Tp, Tp]] | irreflexive?(ord0) & transitive?(ord0) }

ord : VAR Ord
i, j, k : VAR nat
t1, t2, x, y, a : VAR Tp

fs : VAR non_empty_finite_set

maxset(fs, ord) : setof[Tp] =
   { a | fs(a) & (FORALL x : fs(x) => (NOT ord(a, x))) }

max_nonempty : LEMMA nonempty?(maxset(fs, ord))

max(fs, ord) : Tp = choose(maxset(fs, ord))

ts(fs, ord) : RECURSIVE [(fs) -> nat] =
   IF empty?(remove(max(fs, ord), fs)) THEN (LAMBDA (x | fs(x)) : 1)
   ELSE (LAMBDA (x | fs(x)) :
            IF x = max(fs, ord) THEN card(fs)
            ELSE ts(remove(max(fs, ord), fs), ord)(x) ENDIF) ENDIF
      MEASURE card(fs)

ts_ord : LEMMA
   (fs(x) & fs(y) & ord(x, y)) => ts(fs, ord)(x) < ts(fs, ord)(y)

ts_inj : LEMMA
   (fs(x) & fs(y) & x /= y) => ts(fs, ord)(x) /= ts(fs, ord)(y)

END AssignInj



AcycOrd [Transactions, DataItems, ActionNames : TYPE+,
   R, W: ActionNames ] : THEORY
BEGIN

IMPORTING OrdConf2 [Transactions, DataItems, ActionNames, R, W ]
```

```
S, S0, S1, S2 : VAR Schedules
T, T1, T2   : VAR Transactions
i, j, k, l, m, n   : VAR nat
TS : VAR Timestamp
fst : VAR finite_sequence[Transactions]

ConflictC0(S, (k | k > 0))(T1, T2) : bool =
   EXISTS fst : length(fst) = k + 1 & T1 = first(fst) & T2 = last(fst) &
    (FORALL i : i < length(fst) - 1 => Conflict(S)(fst(i), fst(i + 1)))

ConflictC(S)(T1, T2) : bool =
   EXISTS k : k > 0 & ConflictC0(S, k)(T1, T2)

Cacyclic(S) : bool = irreflexive?(ConflictC(S))

IMPORTING AssignInj [Transactions]

trans_lem : LEMMA transitive?(ConflictC(S))

TSI(S | Cacyclic(S)) : Timestamp = LAMBDA T :
   IF active(S)(T) THEN ts(active(S), ConflictC(S))(T) ELSE 0 ENDIF

TSI_lem : LEMMA Cacyclic(S) => CPTinj(S, TSI(S))

AcycOrdI : LEMMA Cacyclic(S) => OrderedI(S)

cpt_induct : LEMMA
   (CPT(S, TS) & k = n + 1 & ConflictC0(S, k)(T1, T2)) => TS(T1) < TS(T2)

OrdAcyc : LEMMA Ordered(S) => Cacyclic(S)

END AcycOrd
```

3.7.4 Definition of correct behaviours

```
ConstructProt [ States, Actions : TYPE, is : States,
   Pre : [[States, Actions] -> bool],
   Effect : [[States, Actions, States] -> bool] ] : THEORY
BEGIN

PreRuns : TYPE = [# states  : finite_sequence[States],
                    actions : finite_sequence[Actions] #]

Runs : TYPE = { r0 : PreRuns |
   length(states(r0)) = length(actions(r0)) + 1 &
   states(r0)(0) = is &
   FORALL (i : nat) :
      i <  length(actions(r0)) =>
          (Pre(states(r0)(i), actions(r0)(i)) &
           Effect(states(r0)(i), actions(r0)(i), states(r0)(i + 1))) }

S : VAR finite_sequence[Actions]
```

```
protocol(S) : bool = EXISTS (r : Runs) : S = actions(r)

END ConstructProt
```

3.7.5 The 2PL protocol

```
AllActionNames : THEORY
BEGIN

AllActionNames : TYPE = {R, W, unlock, downgrade, Rrequest, Wrequest}

END AllActionNames



Tpl [Transactions, DataItems : TYPE+ ] : THEORY
BEGIN

IMPORTING AllActionNames

at : VAR AllActionNames

ActionNames2PL(at) : bool =
   R?(at) OR W?(at) OR unlock?(at) OR downgrade?(at)

IMPORTING ActionsDef [(ActionNames2PL), Transactions, DataItems]

States2PL : TYPE = [# xset: [Transactions -> setof[DataItems] ],
                      sset: [Transactions -> setof[DataItems] ],
                      shrinking: setof[Transactions] #]

T, T0, T1, T2   : VAR Transactions
x, y : VAR DataItems
s, s1, s2 : VAR States2PL
a1, a2, a3 : VAR Actions
i, j, k, l, m, n   : VAR nat

is2PL : States2PL = (# xset := LAMBDA T : emptyset,
                       sset := LAMBDA T : emptyset,
                       shrinking := emptyset #)

Pre2PL(s1, a1) : bool =
 LET x = dat(a1), T = tr(a1) IN
    CASES act(a1) OF

R: (FORALL T2 : xset(s1)(T2)(x) => T2 = T) &
   (shrinking(s1)(T) => (xset(s1)(T)(x) OR sset(s1)(T)(x))),

W: (FORALL T2 : (xset(s1)(T2)(x) OR sset(s1)(T2)(x)) => T2 = T) &
   (shrinking(s1)(T) => xset(s1)(T)(x)),

unlock: xset(s1)(T)(x) OR sset(s1)(T)(x),
```

```
downgrade: xset(s1)(T)(x)

    ENDCASES

Effect2PL(s1, a1, s2) : bool =
 LET x = dat(a1), T = tr(a1) IN
    CASES act(a1) OF

R: IF xset(s1)(T)(x) THEN s2 = s1
   ELSE s2 = s1 WITH [sset := sset(s1) WITH [T := add(x, sset(s1)(T))]]
   ENDIF,

W: IF xset(s1)(T)(x) THEN s1 = s2
   ELSE s2 = s1 WITH
      [xset := xset(s1) WITH [T := add(x, xset(s1)(T)) ],
       sset := sset(s1) WITH [T := remove(x, sset(s1)(T)) ] ]
   ENDIF,

unlock: s2 = s1 WITH
      [xset := xset(s1) WITH [T := remove(x, xset(s1)(T)) ],
       sset := sset(s1) WITH [T := remove(x, sset(s1)(T)) ],
       shrinking := add(T, shrinking(s1)) ],

downgrade: s2 = s1 WITH
   [xset := xset(s1) WITH [T := remove(x, xset(s1)(T)) ],
    sset := sset(s1) WITH [T := add(x, sset(s1)(T)) ],
    shrinking := add(T, shrinking(s1)) ]

     ENDCASES

IMPORTING ConstructProt [States2PL, Actions, is2PL, Pre2PL, Effect2PL]
IMPORTING ConfView [Transactions, DataItems, (ActionNames2PL?), R, W]
IMPORTING min, max

S : VAR Schedules

ShrinkSet(S, T) : setof[nat] =
   { i | i < length(S) & T = tr(S(i)) &
         (act(S(i)) = unlock OR act(S(i)) = downgrade) }

TS2PL(S) : Timestamp = LAMBDA T :
   IF nonempty?(ShrinkSet(S, T)) THEN minimum(ShrinkSet(S, T)) ELSE
      IF nonempty?(IndexSet(S, T)) THEN maximum(IndexSet(S, T))
      ELSE 0 ENDIF ENDIF

CPT2PL : THEOREM CPTinj(actions(r), TS2PL(actions(r)))

ProtOrd : THEOREM EXISTS TS: CPT(actions(r), TS)

ProtConf : THEOREM protocol(S) => Conf_serializable(S)

Sc : VAR Schedules2
```

```
ProtView : THEOREM protocol(Sc) => View_serializable(Sc)

END Tpl
```

3.7.6 The TSO protocol

```
tso [Transactions, DataItems : TYPE+ ] : THEORY
BEGIN

ActionNamesTSO : TYPE = { R, W }

IMPORTING Actions [ActionNamesTSO, Transactions, DataItems]

StatesTSO : TYPE = [# Rts : [DataItems -> nat],
                      Wts : [DataItems -> nat],
                      TTS : [Transactions -> nat],
                      counter : nat #]

T, T0, T1, T2   : VAR Transactions
x, y : VAR DataItems
s, s1, s2 : VAR StatesTSO
a1, a2, a3 : VAR Actions

isTSO : StatesTSO = (# Rts := LAMBDA x : 0, Wts := LAMBDA x : 0,
                       TTS := LAMBDA T : 0, counter := 1 #)

PreTSO(s1, a1) : bool =
 LET x = dat(a1), T = tr(a1) IN
    CASES act(a1) OF

R: TTS(s1)(T) > 0 => TTS(s1)(T) >= Wts(s1)(x),

W: TTS(s1)(T) > 0 =>
      (TTS(s1)(T) >= Rts(s1)(x) & TTS(s1)(T) >= Wts(s1)(x))

   ENDCASES

EffectTSO(s1, a1, s2) : bool =
 LET x = dat(a1), T = tr(a1) IN
    CASES act(a1) OF

R: s2 = s1 WITH
   [Rts := Rts(s1) WITH [x := IF TTS(s1)(T) = 0 THEN counter(s1)
                             ELSE max(Rts(s1)(x), TTS(s1)(T)) ENDIF],
    TTS := TTS(s1) WITH [T := IF TTS(s1)(T) = 0 THEN counter(s1)
                             ELSE TTS(s1)(T) ENDIF],
    counter := IF TTS(s1)(T) = 0 THEN counter(s1) + 1
               ELSE counter(s1) ENDIF],

W: s2 = s1 WITH
   [Wts := Wts(s1) WITH [x := IF TTS(s1)(T) = 0 THEN counter(s1)
                             ELSE TTS(s1)(T) ENDIF],
```

```
    TTS := TTS(s1) WITH [T := IF TTS(s1)(T) = 0 THEN counter(s1)
                              ELSE TTS(s1)(T) ENDIF],
    counter := IF TTS(s1)(T) = 0 THEN counter(s1) + 1
               ELSE counter(s1) ENDIF]

   ENDCASES

IMPORTING ConstructProt [StatesTSO, Actions, isTSO, PreTSO, EffectTSO]
IMPORTING OrdConf [Transactions, DataItems, ActionNamesTSO, R, W]
IMPORTING fin_seq [StatesTSO]

r, r1, r2 : VAR Runs

TSTSO(r) : Timestamp = TTS(last(states(r)))

CPTTSO : THEOREM CPTinj(actions(r), TSTSO(actions(r)))

ProtOrd : THEOREM EXISTS TS: CPT(actions(r), TS)

ProtConf : THEOREM protocol(S) => Conf_serializable(S)

END tso
```

3.7.7 Extensions of serializable protocols

```
ExtendStates  [ OldStates, Extension : TYPE ] : THEORY
BEGIN

ExtendedStates : TYPE = [# old  : OldStates, ext : Extension #]

END ExtendStates



VisiblePred [ActionNames : TYPE, R, W : ActionNames] : THEORY
BEGIN

Visible? : TYPE = { P : pred[ActionNames] | P(R) & P(W) }

END VisiblePred



ExtendProt
 [ Transactions, DataItems, NewActNames : TYPE+,
   R, W : NewActNames,
   (IMPORTING VisiblePred[NewActNames, R, W])
   OldActName? : Visible?,
   (IMPORTING ActionsDef[(OldActName?), Transactions, DataItems],
              ActionsDef[NewActNames, Transactions, DataItems] )
   OldStates, NewStatePart : TYPE,
   (IMPORTING ExtendStates[OldStates, NewStatePart])
   OldInitState : OldStates,
```

```
   OldPre : [[OldStates,
              Actions[(OldActName?),Transactions, DataItems]] -> bool],
   OldEffect : [[OldStates,
                 Actions[(OldActName?),Transactions, DataItems],
                 OldStates] -> bool],
   NewInitState : ExtendedStates,
   NewPre : [[ExtendedStates,
              Actions[NewActNames, Transactions, DataItems]] -> bool],
   NewEffect : [[ExtendedStates,
                 Actions[NewActNames, Transactions, DataItems],
                 ExtendedStates] -> bool] ] : THEORY
BEGIN

ASSUMING

es, es1, es2 : VAR ExtendedStates
oa, oa1, oa2 : VAR Actions[(OldActName?),Transactions,DataItems]
aa, aa1, aa2 : VAR { na | NOT OldActName?(act(na)) }

InitialStateAssume : ASSUMPTION old(NewInitState) = OldInitState

EffectNewAssume : ASSUMPTION
   NewEffect(es1, aa, es2) => old(es1) = old(es2)

PreAssume : ASSUMPTION NewPre(es, oa) => OldPre(old(es), oa)

EffectOldAssume : ASSUMPTION NewEffect(es1, oa, es2) =>
                             OldEffect(old(es1), oa, old(es2))
ENDASSUMING

OldActions : TYPE = Actions[(OldActName?), Transactions, DataItems]
NewActions : TYPE = Actions[NewActNames,Transactions,DataItems]
OldSchedules : TYPE = finite_sequence[OldActions]
NewSchedules : TYPE = finite_sequence[NewActions]

OldS : VAR OldSchedules
NewS : VAR NewSchedules
NewA, NewA1, NewA2 : VAR NewActions

IMPORTING fin_seq[NewActions]

Extract0(NewA) : OldSchedules =
   IF OldActName?(act(NewA)) THEN one(NewA) ELSE empty_seq ENDIF

Extract(NewS) : RECURSIVE OldSchedules =
   IF length(NewS) = 0 THEN empty_seq[NewActions]
   ELSE o[NewActions] (Extract(prefix(NewS)), Extract0(last(NewS)))
   ENDIF
      MEASURE length(NewS)

IMPORTING ConstructProt [ OldStates, OldActions,
                          OldInitState, OldPre, OldEffect ]
```

```
IMPORTING ConstructProt [ ExtendedStates, NewActions,
                          NewInitState, NewPre, NewEffect ]

OldProt(OldS) : bool = protocol
   [OldStates, OldActions OldInitState, OldPre, OldEffect] (OldS)

NewProt(NewS) : bool = protocol
   [ExtendedStates, NewActions, NewInitState, NewPre, NewEffect] (NewS)

IMPORTING  AcycOrd [Transactions, DataItems, (OldActName?), R, W]
IMPORTING  AcycOrd [Transactions, DataItems, NewActNames, R, W]

Conf_Ser_Old(OldS) : bool =
   Conf_serializable [Transactions,DataItems,(OldActName?),R,W] (OldS)

Conf_Ser_New(NewS) : bool =
   Conf_serializable [Transactions,DataItems,NewActNames,R,W] (NewS)

ExtractOld : THEOREM NewProt(NewS) => OldProt(Extract(NewS))

ConfNewOld : THEOREM Conf_Ser_Old(Extract(NewS)) => Conf_Ser_New(NewS)

MainTheorem : THEOREM
   subset?(OldProt, Conf_Ser_Old) => subset?(NewProt, Conf_Ser_New)

END ExtendProt
```

3.7.8 Two extensions of the 2PL protocol

```
TplExt1 [Transactions, DataItems : TYPE+ ] : THEORY
BEGIN

IMPORTING tpl [Transactions, DataItems]
IMPORTING Actions [AllActions, Transactions, DataItems]

ReadWrite : TYPE = { at : AllActions | R?(at) OR W?(at) }

Requests : TYPE = [# tr : Transactions, act  : ReadWrite #]

Extension1 : TYPE = [ DataItems -> finite_sequence[Requests]]

IMPORTING ExtendStates[ States2PL, Extension1 ]

States1 : TYPE = ExtendedStates[States2PL, Extension1]

at : VAR AllActions

x, y : VAR  DataItems
i, j, k, l, m, n   : VAR nat
T, T1, T2   : VAR Transactions
e1, e2 : VAR Extension1
es1, es2 : VAR States1
s1, s2, s3 : VAR States2PL
```

```
a1, a2    : VAR Actions[AllActions, Transactions, DataItems]
re : VAR Requests
q, q1 : VAR finite_sequence[Requests]

NewIS1 : States1 = (# old := is2PL, ext := LAMBDA x : empty_seq #)

waiting(e1) : setof[Transactions] =
   { T | EXISTS x, i : i < length(e1(x)) & T = tr(seq(e1(x))(i)) }

IndSet(q, re) : setof[nat] = { i | i < length(q) & seq(q)(i) = re }

request_exists(q, re) : bool = nonempty?[nat](IndSet(q, re))

remove_element(q, (i | length(q) > 0 & i < length(q))) :
   finite_sequence[Requests] =
      (# length := length(q) - 1,
         seq := (LAMBDA (k : below[length(q) - 1]) :
                   IF k < i THEN seq(q)(k) ELSE seq(q)(k + 1) ENDIF) #)

remove_request(q, (re | request_exists(q, re))) :
   finite_sequence[Requests] = remove_element(q, choose(IndSet(q, re)))

Remove_request(e1, e2, x, (re | request_exists(e1(x), re))) : bool =
   e2 = e1 WITH [ x := remove_request(e1(x), re) ]

IMPORTING fin_seq[Requests]

Insert_request(e1, e2, x, re) : bool =
   e2 = e1 WITH [ x := e1(x) o one(re) ]

AddPre1(es1, a1) : bool =
   LET s1 = old(es1), e1 = ext(es1),
       x = dat(a1), T = tr(a1), at = act(a1) IN

( (R?(at) OR W?(at)) =>
       ((NOT member(T, waiting(e1))) & length(e1(x)) = 0) OR
        request_exists(e1(x), (# tr := T, act := at #))) &

( (unlock?(at) OR  downgrade?(at)) => (NOT member(T, waiting(e1)))) &

( Rrequest?(at) =>
      ((NOT member(T, waiting(e1))) &
       (EXISTS T1 : T1 /= T & member(x, xset(s1)(T1))))) &

( Wrequest?(at) =>
      ((NOT member(T, waiting(e1))) &
       (EXISTS T1 : T1 /= T & (member(x, xset(s1)(T1)) OR
                               member(x, sset(s1)(T1))))))

NewPre1(es1, a1) : bool = AddPre1(es1, a1) &
   (ActionNames2PL?(act(a1)) => Pre2PL(old(es1), a1))

AddEffect1(e1, a1, e2) : bool =
```

```
   LET x = dat(a1), T = tr(a1), at = act(a1) IN

( (R?(at) OR W?(at)) =>
    IF (NOT member(T, waiting(e1))) THEN e2 = e1
    ELSE (request_exists(e1(x), (# tr := T, act := at #)) &
          Remove_request(e1, e2, x, (# tr := T, act := at #))) ENDIF ) &

( (unlock?(at) OR downgrade?(at)) =>  e2 = e1) &

( (Rrequest?(at) OR Wrequest?(at)) =>
      Insert_request(e1, e2, x,
         (# tr := T, act := IF Rrequest?(at) THEN R ELSE W ENDIF #)))

NewEffect1(es1, a1, es2) : bool = AddEffect1(ext(es1), a1, ext(es2)) &
   (IF ActionNames2PL?(act(a1)) THEN Effect2PL(old(es1), a1, old(es2))
    ELSE old(es2) = old(es1) ENDIF)

IMPORTING ExtendProt [Transactions, DataItems, AllActions, R, W,
   ActionNames2PL?, States2PL, Extension1, is2PL, Pre2PL, Effect2PL,
   NewIS1, NewPre1, NewEffect1]

NewS : VAR NewSchedules

CS_Extension1 : THEOREM NewProt(NewS) => Conf_Ser_New(NewS)

END TplExt1



TplExt2 [Transactions, DataItems : TYPE+,
         PR0 : [Transactions -> nat], U0 : nat ] : THEORY
BEGIN

IMPORTING TplExt1 [Transactions, DataItems]

Extension2 : TYPE = [# PR : [Transactions -> nat], U  : nat #]

IMPORTING ExtendStates[ States1, Extension2 ]

States2 : TYPE = ExtendedStates[ States1, Extension2 ]

NewIS2 : States2 = (# old := NewIS1, ext := (# PR := PR0, U := U0 #) #)

x, y : VAR  DataItems
i, j, k, l, m, n   : VAR nat
T, T1, T2   : VAR Transactions
PR1 : VAR [Transactions -> nat]
q : VAR finite_sequence[Requests]
a1, a2 : VAR Actions[AllActions, Transactions, DataItems]
es1, es2 : VAR States2
at : VAR AllActions

ts_val(q, PR1) : setof[nat] =
```

```
   { i | EXISTS k : k < length(q1) & PR1(tr(seq(q1)(k))) = i }

MaxPriority(q1, (PR1 | length(q1) > 0)) : nat = maximum(ts_val(q1, PR1))

AddPre2(es1, a1) : bool =
   LET x = dat(a1), T = tr(a1), at = act(a1) IN
((R?(at) OR W?(at)) & length(ext(old(es1))(x)) > 0) =>
      IF MaxPriority(ext(old(es1))(x), PR(ext(es1))) > U(ext(es1))
      THEN PR(ext(es1))(T) = MaxPriority(ext(old(es1))(x), PR(ext(es1)))
      ELSE T = tr(seq(ext(old(es1))(x))(0)) ENDIF

NewPre2(es1, a1) : bool = NewPre1(old(es1), a1) & AddPre2(es1, a1)

NewEffect2(es1, a1, es2) : bool =
   NewEffect1(old(es1), a1, old(es2)) & ext(es1) = ext(es2)

TruePred?(at) : bool = TRUE

IMPORTING ExtendProt [Transactions, DataItems, AllActions, R, W,
   TruePred?, States1, Extension2, NewIS1, NewPre1, NewEffect1,
   NewIS2, NewPre2, NewEffect2]

NewS : VAR NewSchedules[Transactions, DataItems, AllActions, R, W,
   TruePred?, States1, Extension2, NewIS1, NewPre1, NewEffect1,
   NewIS2, NewPre2, NewEffect2]

NewProt2(NewS) : bool = NewProt[Transactions,DataItems,AllActions,R,W,
   TruePred?, States1, Extension2, NewIS1, NewPre1, NewEffect1,
   NewIS2, NewPre2, NewEffect2](NewS)

Conf_Ser_New2(NewS) : bool =
   Conf_Ser_New[Transactions, DataItems, AllActions, R, W,
      TruePred?, States1, Extension2, NewIS1, NewPre1, NewEffect1,
      NewIS2, NewPre2, NewEffect2](NewS)

CS_Extension2 : THEOREM NewProt2(NewS) => Conf_Ser_New2(NewS)

END TplExt2
```

Chapter 4

Atomicity, Durability and Fault-Tolerant Serializability

In this chapter, the properties of atomicity, durability and conflict serializability are formalized for a fault-tolerant environment, in which both transaction and memory failures (as defined in section 1.1) are considered. We verify these properties for an experimental protocol, integrating a few typical protocols for concurrency control and recovery: strict two-phase locking, undo/redo recovery and two-phase commit. Unlike the protocols from section 3.6, this protocol is too different from the Two Phase Locking protocol to be formalized as its extension. However, we still have been able to reuse many definitions and verification techniques from chapter 3. As in chapter 3, in the main sections we show only mathematical definitions and proofs extracted from PVS definitions and proofs, and only briefly explain how to implement them in PVS. The actual PVS specifications of the protocol and its correctness properties are given at the end of the chapter.

The structure of the chapter is as follows. In section 4.1, we rigorously define decision consistency and the "update in place" read policy, which in section 2.1.1 are shown to represent atomicity and durability. We also define conflict serializability for a fault-tolerant model and adjust our method for the verification of conflict serializability from chapter 3 to this model. Section 4.2 describes the main differences between our protocol and conventional protocols for concurrency control and recovery. Section 4.3 outlines the protocol itself. The modelling of the protocol and its correctness properties in PVS is discussed in section 4.4. The verification of decision consistency, update in place and conflict serializability is presented in sections 4.5, 4.6 and 4.7, respectively. Actual PVS specifications are given in section 4.8.

4.1 Formalization of correctness properties

We use the same notation as in chapter 2 with only a few small changes, aiming at an easier translation of our specifications into PVS. In this chapter, each action $a1$ is represented by a record with two fields: $name(a1)$ is the *name* of $a1$ (e.g., $Read(T, x, v)$, $Commit(T)$, etc.), and $site(a1)$ is the site where $a1$ is executed. For each data item x, $sf(x)$ denotes a site at which x is located. If $name(a1)$ is of the form $Read(T, x, v)$, $Write(T, x, v)$ or $Flush(x)$, the combination of $name(a1)$ and $site(a1)$ should be meaningful. Therefore we require that x is located at $site(a1)$, formally represented by $site(a1) = sf(x)$. However, we may omit the "site" field of read, write and flush actions when its value is not important. As mentioned in section 2.1, a transaction is allowed to write to a data item only once, and values of data items are labelled by identifiers of transactions that produced these

values.

As explained in section 1.5.2, in this chapter we define the correctness properties for *infinite* schedules instead of finite schedules to make it easier to work with them in PVS. $Si(i)$ now represents an action in an *infinite* schedule Si with index i, and unlike chapter 3, we don't have to make sure that $i < length(Si)$. As in chapter 3, $T0$ is a fictitious write-only *initial* transaction which writes the initial values of all the data items and doesn't have any other actions.

4.1.1 Formalization of decision consistency and "update in place"

Henceforth, i, j, k, l, m and n represent variables from the set of natural numbers, and int represents a variable from the set of integers. To formalize decision consistency and the "update in place" read policy, we introduce the following abbreviations:

$$Commits(T, st, i, Si) \stackrel{def}{\iff} Si(i) = (Commit(T), st)$$
$$Commits(T, Si) \stackrel{def}{\iff} \exists st, i : Commits(T, st, i, Si)$$
$$Aborts(T, st, i, Si) \stackrel{def}{\iff} Si(i) = (Abort(T), st)$$
$$Aborts(T, Si) \stackrel{def}{\iff} \exists st, i : Aborts(T, st, i, Si)$$

Definition 4.1
The decisions to commit or abort a transaction are *consistent* in a schedule Si, represented by $DecisionConsistency(Si)$, if

$$DecisionConsistency(Si) \stackrel{def}{\iff} \forall T : Commits(T, Si) \implies \neg Aborts(T, Si)$$

The definition of "update in place" uses the following abbreviations:

$$AbortsBetween(T, x, i, j, Si) \stackrel{def}{\iff} \exists k : i < k \,\&\, k < j \,\&\, Aborts(T, sf(x), k, Si)$$
$$NoWritesBetween(x, int, j, Si) \stackrel{def}{\iff}$$
$$\forall l, T : (int < l \,\&\, l < j \,\&\, Si(l) = Write(T, x, x_T)) \implies AbortsBetween(T, x, l, j, Si)$$

Thus predicate $AbortsBetween(T, x, i, j, Si)$ means that T aborts at site $sf(x)$ at some index between i and j in a schedule Si. Predicate $NoWritesBetween(x, i, j, Si)$ indicates that each transaction T writing to x at some index l between indexes i and j must abort between indexes l and j.

Definition 4.2
A schedule Si satisfies the "update in place" read policy, denoted by $UpdateInPlace0(Si)$, if the following condition holds:

$$\begin{aligned}
&UpdateInPlace0(Si) \stackrel{def}{\iff} \forall T2, T1, x, j : \\
&\quad (Si(j) = Read(T2, x, x_{T0}) \implies NoWritesBetween(x, -1, j, Si)) \,\& \\
&\quad ((Si(j) = Read(T2, x, x_{T1}) \,\&\, T1 \neq T0) \implies \\
&\qquad \exists i : i < j \,\&\, Si(i) = Write(T1, x, x_{T1}) \,\& \\
&\qquad\quad \neg AbortsBetween(T1, x, i, j, Si) \,\& \\
&\qquad\quad NoWritesBetween(x, i, j, Si))
\end{aligned}$$

The informal definition of "update in place" in section 2.1.1 says that each read should obtain the "last non-aborted value" of a data item. As in section 3.1, in the formal definition the values produced

by $T0$ are treated differently from other values. The part of $UpdateInPlace0$ corresponding to $T0$ requires that a transaction $T2$ may read a value of x produced by $T0$ only if all previous writes to x are aborted. The remaining part requires that $T2$ may read a value of x produced by $T1$ ($T1 \neq T0$) only if $T1$ wrote this value before, it is not aborted, and all other values of x written between the write by $T1$ and the read by $T2$ are aborted.

Most protocols, including the protocol considered in this chapter, do not allow to read uncommitted values for reasons discussed in section 2.2.2. Such protocols should satisfy the stronger requirement which we denote by $UpdateInPlace$.

$$CommitsBetween(T, x, i, j, Si) \stackrel{def}{\Longleftrightarrow} \exists k : i < k \,\&\, k < j \,\&\, Commits(T, sf(x), k, Si)$$

$$\begin{aligned}
&UpdateInPlace(Si) \stackrel{def}{\Longleftrightarrow} \forall T2, T1, x, j : \\
&\quad (Si(j) = Read(T2, x, x_{T0}) \Longrightarrow NoWritesBetween(x, -1, j, Si)) \,\& \\
&\quad ((Si(j) = Read(T2, x, x_{T1}) \,\& \, T1 \neq T0) \Longrightarrow \\
&\qquad \exists i : i < j \,\&\, Si(i) = Write(T1, x, x_{T1}) \,\& \\
&\qquad\quad (T1 = T2 \vee CommitsBetween(T1, x, i, j, Si)) \,\& \\
&\qquad\quad NoWritesBetween(x, i, j, Si))
\end{aligned}$$

Predicate $UpdateInPlace$ distinguishes between cases when $T2$ reads a value produced by itself and when $T2$ reads a value produced by another transaction. If a value is produced by another transaction, this value is required to be committed. It is easy to see that if Si satisfies decision consistency, then $UpdateInPlace(Si)$ implies $UpdateInPlace0(Si)$ (assuming, of course, that a transaction does not read any values after it aborts). In our PVS specification, the definition of the "update in place" policy corresponds to predicate $UpdateInPlace$.

Remark. The intuitive notion of durability reflects the fact that a committed value does not disappear until it is over-written by another committed value, and therefore it survives subsequent memory failures. A more formal definition of durability may be given by the following predicate $Durability$.

$$\begin{aligned}
&Durability(Si) \stackrel{def}{\Longleftrightarrow} \forall T2, T, T1, x, i, j : \\
&\quad ((NoWritesBetween(x, -1, j, Si) \,\& Si(j) = Read(T2, x, x_T)) \Longrightarrow T = T0) \,\& \\
&\quad ((Si(i) = Write(T1, x, x_{T1}) \,\& \, i < j \,\&\, CommitsBetween(T1, x, i, j, Si) \,\& \\
&\quad NoWritesBetween(x, i, j, Si) \,\&\, Si(j) = Read(T2, x, x_T)) \Longrightarrow T = T1)
\end{aligned}$$

However, we easily proved in PVS that if Si satisfies decision consistency, then $UpdateInPlace(Si)$ implies $Durability(Si)$. Therefore we don't need a separate notion of durability.

4.1.2 Formalization of serializability

As in section 2.1.2, we denote by $C(Si)$ the *committed projection* of a schedule Si, i.e. a schedule obtained from Si by removing all actions except for reads and writes of committed transactions. In section 2.1.2, Si is said to be conflict serializable, if $C(Si)$ is conflict equivalent to some serial schedule. Since serializability is a safety property, in a more formal definition it is sufficient to consider all finite prefixes of $C(Si)$. If "update in place" is proved for Si, then in $C(Si)$ each read again obtains exactly the last written value of a data item, and therefore the definition of conflict equivalence from section 3.1 is applicable to finite prefixes of $C(Si)$. For a schedule Si, we denote by *prefix(Si, k)* its

prefix of length k.

Definition 4.3
A schedule Si is *fault-tolerant conflict serializable*, denoted by $FTCS0(Si)$, if

$FTCS0(Si) \stackrel{def}{\Longleftrightarrow} \forall k$: *Conf_serializable(prefix(C(Si), k))*

It is known that it is difficult to define projections of *infinite* schedules in PVS, as well as other theorem provers, and to reason about such projections [DGM97]. This is why we adjusted definition 4.3 such that it now uses only committed projection of *finite* schedules. For a *finite* schedule Sf and a set of transactions $Tset$ let $Extr(Sf, Tset)$ represent a schedule obtained from Sf by removing all actions except for reads and writes of transactions in $Tset$. Note the similarity of this projection to the function $Extract$ used in section 3.5. Let $CommitSet(Si)$ denote the set of all transactions that commit in an infinite schedule Si.

Definition 4.4
A schedule Si is *fault-tolerant conflict serializable*, denoted by $FTCS(Si)$, if

$FTCS(Si) \stackrel{def}{\Longleftrightarrow} \forall k$: *Conf_serializable(Extr(prefix(Si, k), CommitSet(Si)))*

It is easy to see that definitions 4.3 and 4.4 are equivalent. In our PVS specification, the definition of fault-tolerant conflict serializability is based on predicate $FTCS$.

Verification of serializability. We adjusted our method for verification of conflict serializability from section 3.2 in a straightforward fashion.

Definition 4.5
A conflict relation $Conflict\_i(Si)$ of a schedule Si is defined as follows: an ordered pair $(T1, T2)$ belongs to $Conflict\_i(Si)$ iff $T1 \neq T2$, $T1 \in CommitSet(Si)$, $T2 \in CommitSet(Si)$ and

- Si includes actions $a1$ and $a2$ by $T1$ and $T2$ respectively
- $a1$ precedes $a2$ in Si
- $a1$ and $a2$ are conflicting, i.e. they are reads or writes of the same data item and at least one of them is write.

Definition 4.6
A timestamp TS is a *conflict-preserving timestamp (CPT)* with respect to schedule Si, denoted by $CPTi(Si)$, iff for each transactions $T1$ and $T2$ the following condition holds:
$CPTi(Si) \stackrel{def}{\Longleftrightarrow} Conflict\_i(Si)(T1, T2) \Longrightarrow TS(T1) < TS(T2)$

Definition 4.7
A schedule Si is *ordered*, denoted by $Ordered\_i(Si)$, iff there is a timestamp TS which is conflict-preserving with respect to Si.
$Ordered\_i(Si) \stackrel{def}{\Longleftrightarrow} \exists TS : CPTi(Si, TS)$

As in section 3.2, we proved that any ordered schedule is conflict serializable.

```
OrdConfInf : THEOREM Ordered_i(Si) => FTCS(Si)
```

Proof of theorem `OrdConfInf`. Let $Sf = Extr(prefix(Si, k), CommitSet(Si))$ for some arbitrary k. We need to prove *Conf_serializable(Sf)*. By theorem `OrdConfEquiv` from section 3.2, it is sufficient to prove $Ordered(Sf)$. By definition of $Ordered\_i$, Si has a conflict preserving timestamp. Let TS denote this timestamp. We prove that TS is also a CPT for Sf. Indeed, suppose that transactions $T1$ and $T2$ are conflicting in Sf. It implies that Sf includes two conflicting actions $a1$ and $a2$ by $T1$ and $T2$ respectively. The definition of $Extr$ implies that Si also includes $a1$ and $a2$, in the same order (the formal proof of this fact in PVS is rather complicated and is done by induction on k). The definition of $Conflict\_i(Si)$ now implies that $T1$ and $T2$ are also conflicting in Si. TS is a CPT for Si, and it implies $TS(T1) < TS(T2)$. We proved that if transactions $T1$ and $T2$ are conflicting in Sf, then $TS(T1) < TS(T2)$. Therefore TS is indeed a CPT for Sf, i.e. Sf is ordered. Note the similarity of the proof to the proof of theorem `ConfNewOld` in section 3.5.

4.2 Main features of the protocol

In this section, we discuss the main differences between our protocol and conventional concurrency control and recovery protocols. The most significant difference is the close integration of concurrency control and recovery in our protocol, which is very beneficial for both. As explained in sections 2.2 and 2.3, locks on data items is one of the main methods of concurrency control, whereas the log is a part of stable memory that is used for recovery from transaction failures and memory failures. In our protocol, we keep some locks in the log, thus preventing their loss in a system failure and helping to ensure serializability. On the other hand, some information about locks is used to manage the log more efficiently. In the rest of the section we explain the more technical differences.

In [BHG87], the basic undo/redo protocol does not guarantee the consistency of distributed commitment. Indeed, in this protocol all transactions executed at a particular site are aborted after a system failure at this site. Note that if a site votes YES in an execution of 2PC deciding on commit or abort of a transaction T, and a system failure happens at this site before it receives the coordinator's decision, then aborting T may contradict the decision made by other sites. To ensure the decision consistency among multiple sites, we made a non-trivial change to basic undo/redo by introducing a *precommit* action. Action $(Precommit(T), st)$ is an atomic action that puts the values written by T into the log maintained at site st and also sends a YES vote in an execution of 2PC deciding on commit or abort of T. This allows to avoid aborting T if a system failure happens before the arrival of the coordinator's decision, because the lost updates of T may now be redone instead of undone, and T may continue its execution. Note that in Strict 2PL *locks* of T may be released only after it commits or aborts. This implies that locks of a precommitted transaction must survive a system failure as well. Therefore $(Precommit(T), st)$ also adds T's *locks* to the log of st, and they must be restored in a situation described above.

In the basic undo/redo algorithm in [BHG87] the log is represented as a sequence of entries of the form $[T, x, v]$, identifying the value v that transaction T wrote into data item x, and such entry is added each time the corresponding write occurs. We would not gain much by using volatile memory if each write had to access stable memory as well. Therefore it is important to minimize the number of values kept in the log. In our protocol most reads and writes are performed on volatile memory and don't change the log, and the transaction's writes are added to the log only shortly before it commits (to be precise, when it precommits). This is obviously sufficient to ensure that the log always contains last committed value of each data item. These values are only used by restart and abort actions.

We represent the log not as a sequence of entries, but as a combination of a few sets and functions, without considering their physical implementation in stable memory. For instance, function $cvalue$

maps each data item to its last committed value. Suppose the current volatile value of x, represented by v, is produced by transaction T. When T commits, we don't add any "entries" to the log, but simply make an assignment $cvalue(x) := v$. Such high-level representation is chosen not only for ease of verification, but also to reflect current trends in the implementation of transaction processing systems. At the time when [BHG87] was written, operating systems were rather primitive and cheap stable memory devices allowed only sequential access to data. As a result, it indeed was cheaper and more convenient to store log as a sequence. However, modern storage devices allow almost random access to data, whereas modern operating systems easily support complex data structures.

During restart after a system failure, the protocol from [BHG87] *scans* the log, i.e. a sequence of entries of the form $[T,\ x,\ v]$, in order to find the set of updated data items and their last committed values. In our protocol the set of updated data items is collected during normal processing, and only the *last* committed value is included in the log at any time. As a result, the implementation of restart and abort actions is remarkably easy compared to [BHG87] and [Kuo96], and a loop is never required. The absence of loops in the modelling of actions greatly simplifies the verification. There is also no need to specify garbage collection, because it is performed automatically.

4.3 Outline of the protocol

Data structure. Let $DataI$ denote the set of data items located at one of the distributed sites st. Our aim is to describe the protocol performed by st. First we define the data structure of the protocol. Volatile memory contains 1) volatile values of some of the data items in $DataI$ and 2) locks, both exclusive ($xlocks$) and shared ($slocks$). In the definition given below, $Trans$ and $Values$ are uninterpreted data types representing transactions and values of data items in $DataI$, respectively. Since we prefer to use only totally defined functions, ε denotes the absence of a value.

Volatile Memory:
1) $vvalue : DataI \longrightarrow Values \cup \{\varepsilon\}$,
2) $locks$:
  a) $xlocks : Trans \longrightarrow setof[DataI]$,
  b) $slocks : Trans \longrightarrow setof[DataI]$

Stable memory contains stable values of all data items in $DataI$ and the log. The log includes the sets of a) active, b) precommitted, c) committed and d) aborted transactions, e) last committed values of all data items in $DataI$, f) precommitted values of some of the data items in $DataI$ (i.e. values written by transactions that precommitted but have not committed or aborted yet), g) exclusive ($sxlocks$) and shared ($sslocks$) locks of precommitted transactions, h) the set of data items that have ever been updated (changed) in volatile memory, and i) a boolean variable indicating that recovery from a system failure is required.

Stable Memory:
1) $svalue : DataI \longrightarrow Values$,
2) log :
  a) $active : setof[Trans]$,
  b) $precommitted : setof[Trans]$,
  c) $committed : setof[Trans]$,
  d) $aborted : setof[Trans]$,

e) $cvalue : DataI \longrightarrow Values$,
f) $pcvalue : DataI \longrightarrow Values \cup \{\varepsilon\}$,
g) $slocks$:
1) $sxlocks : Trans \longrightarrow setof[DataI]$,
2) $sslocks : Trans \longrightarrow setof[DataI]$,
h) $updated : setof[DataI]$,
i) $crashed : boolean$

In the initial state these values are as follows (recall that $T0$ is a fictitious initial transaction):
$\forall x, T$:
$vvalue(x) = \varepsilon$ & $xlocks(T) = \emptyset$ & $slocks(T) = \emptyset$ &
$svalue(x) = arbitrary\ value\ (produced\ by\ T0)$ &
$active = \emptyset$ & $precommitted = \emptyset$ & $committed = \{T0\}$ & $aborted = \emptyset$
$cvalue(x) = svalue(x)$ & $pcvalue(x) = \varepsilon$ &
$sxlocks(T) = \emptyset$ & $sslocks(T) = \emptyset$ &
$updated = \emptyset$ & $crashed = false$

The protocol is specified by a state machine with 8 atomic actions. Below we show the precondition and the effect of each of them (skipping the "site" field, because here it is equal to st for each action). The effect is given in an imperative style close to specifications of PVS. For instance, record of the form $xlocks := \forall T : \emptyset$ indicates that exclusive locks of all transactions are erased, whereas $xlocks(T) := xlocks(T) \cup \{x\}$ means that T locks x in an exclusive mode, and exclusive locks of all other transactions are not changed.

Note that $Read(T, x, v)$ models a read action obtaining the value v. Hence the precondition of $Read(T, x, v)$ mentions that v is obtained properly. As explained in section 2.3, if the volatile value of x is absent, then it must be fetched from stable memory and read after that. Restart action has a parameter $Tset$ indicating the set of transactions that are aborted during restart.

1) *Read(T, x, v)*
Precondition:
$v =$ if $vvalue(x) \neq \varepsilon$ then $vvalue(x)$ else $svalue(x)$,
$\forall T1 : x \in xlocks(T1) \Longrightarrow T1 = T$,
$T \notin precommitted$ & $T \notin committed$ & $T \notin aborted$
Effect:
$vvalue(x) := v$,
$slocks(T) :=$ if $x \notin xlocks(T)$ then $slocks(T) \cup \{x\}$ else $slocks(T)$,
$active := active \cup \{T\}$

2) *Write(T, x, v)*
Precondition:
$\forall T1 : x \notin xlocks(T1)$ & $(x \in slocks(T1) \Longrightarrow T1 = T)$,
$T \notin precommitted$ & $T \notin committed$ & $T \notin aborted$
Effect:
$vvalue(x) := v$,
$xlocks(T) := xlocks(T) \cup \{x\}$,
$slocks(T) := xlocks(T) \setminus \{x\}$,
$active := active \cup \{T\}$,

$updated := updated \cup \{x\}$

3) *Flush(x)*
Precondition:
$vvalue(x) \neq \varepsilon$
Effect:
$svalue(x) := vvalue(x)$

4) *Precommit(T)*
Precondition:
$T \in active \ \& \ T \notin precommitted \ \& \ T \notin committed \ \& \ T \notin aborted$
Effect:
$precommitted := precommitted \cup \{T\}$,
$active := active \setminus \{T\}$,
$pcvalue := \forall x$: if $x \in xlocks(T)$ then $vvalue(x)$ else $pcvalue(x)$,
$sxlocks(T) := xlocks(T),\ sslocks(T) := slocks(T)$

5) *Commit(T)*
Precondition:
$T \in precommitted \ \& \ T \notin active \ \& \ T \notin committed \ \& \ T \notin aborted$
Effect:
$committed := committed \cup \{T\}$,
$precommitted := precommitted \setminus \{T\}$,
$cvalue := \forall x$: if $x \in xlocks(T)$ then $pcvalue(x)$ else $cvalue(x)$,
$pcvalue := \forall x$: if $x \in xlocks(T)$ then ε else $pcvalue(x)$,
$sxlocks(T) := \emptyset,\ sslocks(T) := \emptyset$,
$xlocks(T) := \emptyset,\ slocks(T) := \emptyset$

6) *Abort(T)*
Precondition:
$(T \in active \ \vee \ T \in precommitted) \ \& \ T \notin committed \ \& \ T \notin aborted$
Effect:
$vvalue := \forall x$: if $x \in xlocks(T)$ then $cvalue(x)$ else $vvalue(x)$,
$aborted := aborted \cup \{T\}$,
$active := active \setminus \{T\}$,
$precommitted := precommitted \setminus \{T\}$,
$pcvalue := \forall x$: if $x \in xlocks(T)$ then ε else $pcvalue(x)$,
$sxlocks(T) := \emptyset,\ sslocks(T) := \emptyset$,
$xlocks(T) := \emptyset,\ slocks(T) := \emptyset$

7) *Crash*
Precondition: none
Effect:
$vvalue := \forall x : \varepsilon$,
$xlocks := \forall T : \emptyset,\ slocks := \forall T : \emptyset$,
$crashed := true$

8) *Restart(Tset)*
Precondition:
$\forall T : T \in Tset \Longleftrightarrow status(T) = act$
Effect:
$vvalue := \forall x :$ if $x \in updated$ then
(if $pcvalue(x) \neq \varepsilon$ then $pcvalue(x)$ else $cvalue(x)$)
else ε,
$xlocks := \forall T : sxlocks(T)$,
$slocks := \forall T : sslocks(T)$,
$aborted := active \cup aborted$,
$active := \emptyset$,
$crashed := false$

There is an additional precondition for all actions concerning $crashed$; a restart action can only be performed if $crashed = true$, and for all other actions we require $crashed = false$.

Comment. The precondition of a write action says that x is not locked in an exclusive mode by any transaction, *including* T. This ensures that T can write to x only once, because if it is already holding an exclusive lock on x, then it has already written to x before.

Implementation issues. Note that read and write actions add the corresponding transaction to the set $active$. A write action also changes the set $updated$. Since the sets $active$ and $updated$ are located in the log, read and write actions require accessing stable memory even if these variables are not really changed. Therefore in a reasonable implementation these variables must also be present in volatile memory, and we would access their values in the log only if necessary. We did not implement this, because it would not add much to the model and would only complicate the verification.

Comparison with other models. Unlike [Kuo96] we model restart as atomic, but we consider this a reasonable assumption. In [BHG87] crash and restart actions are basically combined. We could also do this in our model, but decided to separate these actions to preserve compatibility with our results on distributed commitment. In chapter 5 we verify a timed version of 2PC protocol, and its correctness properties depend on exact periods of time during which a site stays crashed (and therefore is unable to communicate with other sites). In this chapter, the modelling of crashes is essentially the same as in chapter 5.

4.3.1 Specification of distributed commitment

Atomicity and serializability cannot be proved unless the properties AC1, AC2 and AC4 of the 2PC protocol are ensured (AC3 is only needed for liveness properties, which are not considered in this chapter). In our model, AC4 can be easily proved for each site. AC1 and AC2 require exchange of messages between sites. We don't want to consider here the details of the communication mechanism. Hence we assume AC1 and AC2 as given, and express them by global predicates on schedules.

Some additional abbreviations are needed to define these predicates. We say that T commits (aborts) at site st in a schedule Si, represented by $Commits(T, st, Si)$ ($Aborts(T, st, Si)$), if Si includes a $(Commit(T), st)$ ($Abort(T), st)$) action, respectively. We say that T *decides* at site st in a schedule Si, represented by $Decides(T, st, Si)$, if either $Commits(T, st, Si)$ or $Aborts(T, st, Si)$. We say that T *is active* at site st in a schedule Si, represented by $Active(T, st, Si)$, if Si in-

cludes either action $(Read(T, x, v), st)$ or action $(Write(T, x, v), st)$ for some x and v. We say that T precommits at site st at index i in a schedule Si, represented by $Precommits(T, st, i, Si)$, if $Si(i) = (Precommit(T), st)$. As explained in section 4.2, a precommit action corresponds to voting YES in an execution of 2PC. If Si is a schedule corresponding to a system execution, then AC1 and AC2 are defined for Si as follows:

$$AC1(Si) = \forall st1, st2, T :$$
$$\quad Decides(T, st1, Si)\ \&\ Decides(T, st2, Si) \Longrightarrow$$
$$\quad\quad (Commits(T, st1, Si) \Longrightarrow Commits(T, st2, Si))$$

$$AC2(Si) = \forall st, T, i : Commits(T, st, i, Si) \Longrightarrow$$
$$\quad \forall st1 : Active(T, st1, Si) \Longrightarrow \exists j : j < i\ \&\ Precommits(T, st1, j, Si)$$

4.4 Specification in PVS

4.4.1 Modelling of values

The correctness properties of our protocol, that were presented in section 4.1, are concerned only with identifiers of transactions that produced values of data items, and not the values themselves. This is why in our PVS specification we *replace* the "real" values by the identifiers of transactions that produced these values. Therefore, a $Write(T1, x, v)$ action is replaced by a $Write(T1, x)$ action that assigns "$T1$" to x. A $Read(T2, x, v)$ action, where v is produced by $T1$, is replaced by a $Read(T2, x, T1)$ action. The definitions of read and write actions in section 4.3, as well the definitions of correctness notions in section 4.1, are adjusted accordingly. E.g., the PVS definition of $UpdateInPlace$ is based on the following mathematical definition (in which $AbortsBetween$ and $CommitsBetween$ are the same as in section 4.1):

$$NoWritesBetween(x, i, j, Si) \stackrel{def}{\Longleftrightarrow}$$
$$\quad \forall l, T : (i < l\ \&\ l < j\ \&\ Si(l) = Write(T, x)) \Longrightarrow AbortsBetween(T, x, l, j, Si)$$

$$UpdateInPlace(Si) \stackrel{def}{\Longleftrightarrow} \forall T2, T1, x, j :$$
$$\quad (Si(j) = Read(T2, x, T0) \Longrightarrow NoWritesBetween(x, -1, j, Si))\ \&$$
$$\quad ((Si(j) = Read(T2, x, T1)\ \&\ T1 \neq T0) \Longrightarrow$$
$$\quad\quad \exists i : i < j\ \&\ Si(i) = Write(T1, x, T1)\ \&$$
$$\quad\quad\quad (T1 = T2 \vee CommitsBetween(T1, x, i, j, Si))\ \&$$
$$\quad\quad\quad NoWritesBetween(x, i, j, Si))$$

It is certainly not unusual to use only "names" of values in definitions and proofs, and not the values themselves. For instance, the same approach is used for multiversion concurrency control in chapter 5 of [BHG87], in which data items are also labelled by identifiers of transactions that produced these values.

4.4.2 New definition of aborts

Parameter $Tset$ of a restart action allows us to more rigorously define aborts of transactions such that this notion covers both "voluntary" aborts and "forced" aborts, i.e. aborts performed by a system during restart. We redefine the abbreviation $Aborts$ from section 4.1.1 as follows:

$$Aborts(T, st, i, Si) \stackrel{def}{\Longleftrightarrow}$$
$$Si(i) = (Abort(T), st) \vee \exists Tset : Si(i) = (Restart(Tset), st)\ \&\ T \in Tset$$

This new definition of aborts does not require changes to any other definitions in this chapter, including the definition of $UpdateInPlace$.

4.4.3 Formalization of runs

We define a *global state* as a function which assigns to each site its local state. Then the behaviour of a system is represented by an infinite sequence of the form $gs_0 \stackrel{a0}{\rightarrow} ... \stackrel{a_{i-1}}{\rightarrow} gs_i \stackrel{a_i}{\rightarrow} gs_{i+1} \stackrel{a_{i+1}}{\rightarrow} ...$, where the gs_i are global states and a_i are actions, for $i \in N$. Sequence r is a *run* iff gs_0 is the correct initial state for each site, each $gs_i(site(a_i))$ satisfies the precondition of a_i and every pair $(gs_i(site(a_i)), gs_{i+1}(site(a_i)))$ corresponds to the effect of a_i.

In PVS, a run r is formalized as a record with two fields: $states(r)$ is an infinite sequence of global states, and $actions(r)$ is an infinite sequence of actions. For the example run above, we have $states(r) = gs_0...\ gs_i gs_{i+1}...$ and $actions(r) = a_0...\ a_{i-1} a_i a_{i+1}....$ Run r is a *correct run*, if schedules accepted by this run satisfy the properties AC1 and AC2 of distributed commitment protocols, i.e. we have $AC1(actions(r))$ and $AC2(actions(r))$, where predicates AC1 and AC2 are taken from section 4.3. To verify a protocol, we need to prove its correctness properties for each schedule that is accepted by some correct run of this protocol. It is interesting to know whether AC1 and AC2 are needed to prove a particular property, and this is why we use runs instead of correct runs in most lemmas, and explicitly mention AC1 and AC2 in these lemmas when they are needed. We will see in sections 4.5, 4.6 and 4.7 that we need AC1 to prove decision consistency, both AC1 and AC2 to prove serializability, and neither of them to prove the "update in place" read policy.

4.4.4 Use of abstract datatypes

The aim of this subsection is to help the reader to understand the PVS specifications in section 4.8. In this chapter, unlike chapter 3, different actions have different parameters. To implement such actions in PVS, we use abstract data types: a powerful PVS construct. Abstract data type `ActNames` in section 4.6.1 defines the possible names of actions. E.g., a read action is represented as follows:

```
ActNames [Transactions, DataItems : TYPE] : DATATYPE
BEGIN

Read(tr: Transactions, dat: DataItems, trw: Transactions) : Read?
...other 7 actions...

END ActNames
```

Here `Read` is a *constructor*, which has three parameters of types `Transactions`, `DataItems` and `Transactions`, respectively. Constructors exhaustively enumerate the elements of the data type. Moreover, `tr`, `dat` and `trw` are *destructors*, which serve as accessors to the parameters of the constructor. Here we have $tr(Read(T2, x, T1)) = T2$, $dat(Read(T2, x, T1)) = x$ and $trw(Read(T2, x, T1)) = T1$. Finally, `Read?` is a *recognizer*, which can be used in tests or in subtypes, and $Read?(Read(T2, x, T1))$ equals `TRUE`. Note that in our PVS specifications we import

the definition of conflict serializability for finite schedules from chapter 3, which is defined for different actions (denoted by triples). This is why in section 4.8.4 actions represented by the datatype `ActNames` are transformed into actions represented by triples (by predicate `transform`).

Abstract data types also allow us to use only totally defined functions in global states by adding two "undefined" values to the range of the functions. E.g., `vvalue` is defined as a function from the set of all data items `DataItems` to abstract data type `TNU` with three constructors: `trans`, `nothing` and `undefined`. Constructor `trans` has one parameter of type `Transactions` with accessor `trid`, whereas `nothing` and `undefined` have no parameters. If for global state *gs*, site *st* and data item *x* we have `vvalue(gs(st))(x) = undefined`, it means that *x* is not located at *st*. If `vvalue(gs(st))(x) = nothing`, it means that *x* is located at *st*, but has no volatile value. Finally, if `vvalue(gs(st))(x) = trans(T)`, it means that *x* is located at *st*, and its volatile value is produced by transaction *T*.

4.5 Verification of decision consistency

The following lemma $AC4Lem$, equivalent to the property AC4 of distributed commitment protocols, expresses that a decision to commit or abort a transaction is consistent *at each site*, and therefore irreversible.

$$\forall r, T, st : Commits(T, st, actions(r)) \Longrightarrow \neg Aborts(T, st, actions(r)) \qquad \text{AC4Lem}$$

It is easy to see that lemma $AC4Lem$ and assumption $AC1$ immediately imply the decision consistency:

$$\forall r, T : AC1(actions(r)) \Longrightarrow DecisionConsistency(actions(r)) \qquad \text{DCLem}$$

4.5.1 Proof of lemma $AC4Lem$

Suppose we have both $Commits(T, st, actions(r))$ and $Aborts(T, st, actions(r))$. It means that there exist indexes i and j such that $actions(r)(i) = (Commit(T), st)$ and $actions(r)(j) = (Abort(T), st)$. Suppose $i < j$. The effect of a commit action implies that in state with index $i + 1$, T belongs to the set of committed transactions. We prove now that a transaction never leaves the set $committed$ after it gets there:

$$\forall r, T, st, i, k : T \in committed(states(r)(i)(st)) \Longrightarrow \\ T \in committed(states(r)(i + k)(st)) \qquad \text{ComLem}$$

The invariant $ComLem$ is proved by induction on k by considering the precondition and effect of each action. By applying $ComLem$ for r, T, st, $i + 1$ and $j - i - 1$, we obtain $T \in committed(states(r)(j)(st))$. Contradiction with the precondition of the abort action that is performed in state $states(r)(j)$, and this completes the proof for the case $i < j$. The case $j < i$ is treated in a similar way.

4.6 Verification of "update in place"

For an arbitrary run r, the predicate $UpdateInPlace$ from section 4.1.1 must be proved for $actions(r)$. The proof is based on a large number of invariants (about 100), and most of these invariants are proved by induction on the index. We only show the most interesting invariants and the most important proof steps.

4.6.1 Reads of initial values

The case when a transaction reads a value produced by the initial transaction $T0$ is represented by lemma $LemI$:

$$\begin{array}{l}\forall r, T2, x, j : name(actions(r)(j)) = Read(T2, x, T0) \Longrightarrow \\ \quad \forall T3, j : (l < j \;\&\; name(actions(r)(l)) = Write(T3, x)) \Longrightarrow \\ \quad\quad \exists m : l < m \;\&\; m < j \;\&\; Aborts(T3, sf(x), m, actions(r))\end{array} \qquad \text{LemI}$$

This lemma is proved using the invariants $LockInv$, $UnlockLem$ and $CommitInv$, the proof of which will be given later. These invariants use additional abbreviations $ControlsValue$ and $InLog$.

$$\begin{array}{l}ControlsValue(T, x, i, r) \stackrel{def}{\Longleftrightarrow} \\ \quad if\ vvalue(states(r)(i)(sf(x)))(x) \neq \varepsilon\ then\ vvalue(states(r)(i)(sf(x)))(x) = T \\ \quad else\ svalue(states(r)(i)(sf(x)))(x) = T\end{array}$$

We say that T *controls the value of* x in state $states(r)(i)$ if $ControlsValue(T, x, i, r) = TRUE$. The precondition of a read action implies that a read of x by any transaction in state $states(r)(i)$ will obtain T if and only if T controls the value of x in state $states(r)(i)$.

$$\begin{array}{l}InLog(T, x, i, r) \stackrel{def}{\Longleftrightarrow} \\ \quad pcvalue(states(r)(i)(sf(x)))(x) = T \vee cvalue(states(r)(i)(sf(x)))(x) = T\end{array}$$

$$\forall T, x, i, r : x \in xlocks(states(r)(i)(sf(x)))(T) \Longrightarrow ControlsValue(T, x, i, r) \qquad \text{LockInv}$$

$$\begin{array}{l}\forall T, x, i, j, r : \\ (x \in xlocks(states(r)(i)(sf(x)))(T) \;\& \\ i < j \;\&\; x \notin xlocks(states(r)(j)(sf(x)))(T) \;\&\; \neg crashed(states(r)(j)(sf(x)))) \Longrightarrow \\ \quad \exists k : i \leq k \;\&\; k < j \;\& \\ \quad\quad ((Commits(T, sf(x), k, actions(r)) \;\&\; x \in xlocks(states(r)(k)(sf(x)))(T)) \vee \\ \quad\quad Aborts(T, sf(x), k, actions(r)))\end{array} \qquad \text{UnlockLem}$$

$$\begin{array}{l}\forall T, T2, x, i, k, r : \\ \quad ((T \in aborted(states(r)(i)(sf(x))) \vee T \in committed(states(r)(i)(sf(x)))) \;\& \\ \quad T2 \neq T \;\&\; Commits(T2, sf(x), i, actions(r)) \;\&\; x \in xlocks(states(r)(i)(sf(x)))(T2) \;\& \\ \quad j > i \;\&\; \neg crashed(states(r)(j)(sf(x)))) \Longrightarrow \\ \quad\quad (\neg ControlsValue(T, x, j, r) \;\&\; \neg InLog(T, x, j, r))\end{array} \qquad \text{CommitInv}$$

The invariants $LockInv$, $UnlockLem$ and $CommitInv$ have a clear intuitive meaning. Invariant

$LockInv$ expresses that each transaction that holds an exclusive lock on x also controls the value of x. Invariant $UnlockLem$ says that a transaction loses an exclusive lock on a data item only when it commits or aborts. Finally, invariant $CommitInv$ expresses that if a transaction commits holding an exclusive lock on x, then all earlier values of x overwritten by this transaction will never appear in stable or volatile memory or in the log.

4.6.1.1 Proof of lemma $LemI$

Assuming $LockInv$, $UnlockLem$ and $CommitInv$, we sketch the proof of lemma $LemI$. Suppose $name(actions(r)(j)) = Read(T2, x, T0)$, $l < j$ and $name(actions(r)(l)) = Write(T3, x)$. We need to prove that $T3$ aborts at some index between l and j. Effect of a write action implies that in state $states(r)(l+1)$, $T3$ holds an exclusive lock on x. Suppose that in state $states(r)(j)$, $T3$ still holds an exclusive lock on x. Invariant $LockInv$ implies that in state $states(r)(j)$, $T3$ also controls the value of x. Contradiction, because a read of x by $T2$ in this state gives $T0$, and $T0 \neq T3$. Thus $x \notin xlocks(states(r)(j)(sf(x)))(T3)$. We now apply invariant $UnlockLem$ for $T3$, x, $l+1$, j and r, and it implies that $T3$ either commits or aborts at some index between $l+1$ and j. If $T3$ aborts, we are done. Suppose $T3$ commits at some index m holding an exclusive lock on x. We know that $T0$ is committed from the beginning of the run. We now apply invariant $CommitInv$ for $T0$, $T3$, x, m, j and r, and obtain that in state $states(r)(j)$, $T0$ does not control the value of x. Again a contradiction, because a read of x by $T2$ in this state gives $T0$.

4.6.1.2 Proof of lemma $LockInv$

The proof is based on 5 additional invariants. Below we present these invariants and briefly describe how they are proved.

$$\forall T, x, k, r : (\forall j : j \leq k-1 \Longrightarrow name(actions(r)(j)) \neq Write(T, x)) \Longrightarrow$$
$$(\neg x \in xlocks(states(r)(k)(sf(x)))(T)\ \&\ \neg x \in sxlocks(states(r)(k)(sf(x)))(T)) \quad \text{LockInv2}$$

Invariant $LockInv2$ expresses that a transaction T can hold an exclusive lock or a stable exclusive lock on x only if it previously wrote to x. It is proved by showing that T can obtain its *first* exclusive lock on x only when it writes to x (note that T can also obtain an exclusive lock on x as a result of a restart action, but only if T held this lock before and it was lost in a system failure).

$$\forall T, x, i, k, r :$$
$$(\neg crashed(states(r)(i)(sf(x)))\ \&\ T \in active(states(r)(i)(sf(x)))\ \&$$
$$x \in xlocks(states(r)(i)(sf(x)))(T)\ \&\ ControlsValue(T, x, i, r)) \Longrightarrow$$
$$(T \in committed(states(r)(i+k)(sf(x))) \vee$$
$$T \in aborted(states(r)(i+k)(sf(x))) \vee$$
$$\forall l : (l \leq j\ \&\ l \leq i+k\ \&\ \neg crashed(states(r)(l)(sf(x)))) \Longrightarrow$$
$$(x \in xlocks(states(r)(l)(sf(x)))(T)\ \&\ ControlsValue(T, x, l, r))) \quad \text{LockInv3}$$

Invariant $LockInv3$ expresses that after a transaction T obtains an exclusive lock on a data item and control of its value, T either holds them or is added to the set of committed or aborted transactions. It is also proved by induction on k, which checks the precondition and effect of all actions.

$$\forall T, x, i, r : T \in committed(states(r)(i)(sf(x))) \Longrightarrow$$
$$(xlocks(states(r)(i)(sf(x)))(T) = \emptyset \;\&\; sxlocks(states(r)(i)(sf(x)))(T) = \emptyset) \qquad \text{LockInv4}$$

$$\forall T, x, i, r : T \in aborted(states(r)(i)(sf(x))) \Longrightarrow$$
$$(xlocks(states(r)(i)(sf(x)))(T) = \emptyset \;\&\; sxlocks(states(r)(i)(sf(x)))(T) = \emptyset) \qquad \text{LockInv5}$$

Invariants $LockInv4$ and $LockInv5$ express that a committed (aborted) transaction cannot hold an exclusive lock or a stable exclusive lock on any data item. They are proved by showing that a transaction T is added to the set *committed* (*aborted*) only when it performs a commit (abort) action, that T releases all its locks when it performs a commit (abort) action, and that T cannot obtain an exclusive lock after that (because a transaction from the set *committed* (*aborted*) is not allowed to perform any actions).

$$\forall st, i, j, r : (\neg crashed(states(r)(i)(st)) \;\&\; i < j \;\&\; crashed(states(r)(j)(st))) \Longrightarrow$$
$$\exists k : i \leq k \;\&\; k < j \;\&\; \neg crashed(states(r)(k)(st)) \;\&$$
$$actions(r)(k) = (Crash, st) \;\&\; states(r)(k+1)(st) = states(r)(j)(st) \qquad \text{LockInv6}$$

Invariant $LockInv6$, which is rather easily proved by induction, expresses that if site st changes its status from non-crashed to crashed between indexes i and j, then the *last* action performed at st between indexes i and j is $Crash$.

Assuming invariants from $LockInv2$ to $LockInv6$, we present the proof of $LockInv$. Suppose $x \in xlocks(states(r)(i)(sf(x)))(T)$. We must prove $ControlsValue(T, x, i, r)$. Lemma $LockInv2$ implies that there exists $j \leq i-1$ such that $name(actions(r)(j)) = Write(T, x)$. If we apply $LockInv3$ for T, j, $i-j$ and r, it is easy to see that all preconditions of this lemma are satisfied. According to $LockInv3$, three cases are possible. Suppose $T \in committed(states(r)(i)(sf(x)))$. By applying $LockInv4$, we obtain a contradiction with $x \in xlocks(states(r)(i)(sf(x)))(T)$. Suppose $T \in aborted(states(r)(i)(sf(x)))$. By applying $LockInv5$, we again obtain a contradiction with $x \in xlocks(states(r)(i)(sf(x)))(T)$. Finally, suppose that the third expression is true, and let $l = i$ in that expression. This gives us $\neg crashed(states(r)(i)(sf(x))) \Longrightarrow ControlsValue(T, x, i, r)$. If $\neg crashed(states(r)(i)(sf(x)))$, we are done. Suppose now $crashed(states(r)(i)(sf(x)))$. It is easy to see that state $states(r)(j)(sf(x))$ is not crashed. By applying lemma $LockInv6$, we obtain that there exists k such that $k < i$, $actions(r)(k) = (Crash, sf(x))$ and $states(r)(k+1)(sf(x)) = states(r)(j)(sf(x))$. The effect of a crash action implies that in state $states(r)(k+1)(sf(x))$, T does not hold any locks. By applying $states(r)(k+1)(sf(x)) = states(r)(j)(sf(x))$, we obtain a contradiction with $x \in xlocks(states(r)(i)(sf(x)))(T)$, and this completes the proof.

4.6.1.3 Proof of lemma $UnlockLem$

The proof is based on 5 additional invariants. Below we show these invariants and briefly describe how they are proved.

$$\forall T, x, i, j, r :$$
$$(x \in xlocks(states(r)(i)(sf(x)))(T) \;\&\; i < j \;\&\; x \notin xlocks(states(r)(j)(sf(x)))(T)) \Longrightarrow$$
$$\exists k : i \leq k \;\&\; k < j \;\&\; x \in xlocks(states(r)(k)(sf(x)))(T) \;\&$$

$$\forall l : (k+1 \leq l \;\&\; l \leq j) \Longrightarrow x \notin xlocks(states(r)(l)(sf(x)))(T) \qquad \text{UnlockLem2}$$

Invariant $UnlockLem2$, which is rather easily proved by induction without even using the definition of our state machine, expresses that if a transaction releases an exclusive lock between indexes i and j, then there is an index between i and j when it releases this lock *for the last time*.

$$\begin{aligned}\forall T, x, i, r : (x \in xlocks(states(r)(i)(sf(x)))(T) \;\&\; x \notin xlocks(states(r)(i+1)(sf(x)))(T)) \Longrightarrow \\ (Commits(T, sf(x), k, actions(r)) \;\vee\; Aborts(T, sf(x), k, actions(r))) \;\vee \\ actions(r)(k) = (Crash, sf(x))\end{aligned} \qquad \text{UnlockLem3}$$

Invariant $UnlockLem3$ expresses that a transaction can lose an exclusive lock only as a result of a commit, abort or crash action. It is proved by looking at the preconditions and effects of all actions.

$$\begin{aligned}\forall T, st, i, k, r : T \in active(states(r)(i)(st)) \Longrightarrow \\ (T \in active(states(r)(i+k)(st)) \;\vee\; T \in precommitted(states(r)(i+k)(st)) \;\vee \\ T \in committed(states(r)(i+k)(st)) \;\vee\; T \in aborted(states(r)(i+k)(st)))\end{aligned} \qquad \text{UnlockLem4}$$

Invariant $UnlockLem4$ expresses that after a transaction becomes active, it always stays in one of the sets $active$, $precommitted$, $committed$ or $aborted$. It is proved by induction on k by looking at the preconditions and effects of all actions.

$$\begin{aligned}\forall T, x, i, r : \\ (T \in precommitted(states(r)(i)(sf(x))) \;\&\; x \in xlocks(states(r)(i)(sf(x)))(T)) \Longrightarrow \\ x \in sxlocks(states(r)(i)(sf(x)))(T)\end{aligned} \qquad \text{UnlockLem5}$$

Invariant $UnlockLem5$ expresses that a precommitted transaction duplicates all its exclusive locks in stable memory. It is proved by showing that a transaction is added to the set $precommitted$ only when it performs a precommit action, that it saves all its exclusive locks in stable memory in a precommit action, and that stable locks of a precommitted transaction cannot be removed until it commits or aborts.

$$\begin{aligned}\forall st, i, j, r : (crashed(states(r)(i)(st)) \;\&\; i < j \;\&\; \neg crashed(states(r)(j)(st))) \Longrightarrow \\ \exists k, Tset : i \leq k \;\&\; k < j \;\&\; crashed(states(r)(k)(st)) \;\& \\ actions(r)(k) = (Recover(Tset), st) \;\&\; states(r)(i)(st) = states(r)(k)(st)\end{aligned} \qquad \text{UnlockLem6}$$

Invariant $UnlockLem6$, which is rather easily proved by induction, expresses that if site st changes its status from crashed to non-crashed between indexes i and j, then the *first* action performed at st between indexes i and j is $Recover$.

Assuming invariants from $LockInv2$ to $LockInv6$ and from $UnlockLem2$ to $UnlockLem6$, we present the proof of $UnlockLem$. Suppose $x \in xlocks(states(r)(i)(sf(x)))(T)$, $i < j$, $x \notin xlocks(states(r)(j)(sf(x)))(T)$ and $\neg crashed(states(r)(j)(sf(x)))$. We need to prove that T either commits or aborts at some index between i and j. By applying lemma $UnlockLem2$, we obtain that there exists index k between i and j that in state $states(r)(k)(sf(x))$, T releases an exclusive lock on x *for the last time.* According to lemma $UnlockLem3$, three cases are possible. If $Commits(T, sf(x), k, actions(r))$ or $Aborts(T, sf(x), k, actions(r))$, we are done. Suppose $actions(r)(k) = (Crash, sf(x))$.

By applying lemma $LockInv2$ for index k, we obtain that there exists some l such that $l < k$ and $name(actions(r)(l)) = Write(T, x)$. The effect of a write action gives us $T \in active(states(r)(l+1)(sf(x)))$. Lemma $UnlockLem4$ implies that in state $states(r)(k)(sf(x))$, T is in one of the sets $active$, $precommitted$, $committed$ or $aborted$. If T belongs to $committed$ or $aborted$, by applying lemma $LockInv4$ or $LockInv5$ we obtain a contradiction with $x \in xlocks(states(r)(k)(sf(x)))(T)$. Suppose $T \in precommitted(states(r)(k)(sf(x)))$. The effect of a crash action implies $T \in precommitted(states(r)(k+1)(sf(x)))$. Lemma $UnlockLem5$ implies $x \in sxlocks(states(r)(k+1)(sf(x)))(T)$. By applying lemma $UnlockLem6$ for indexes $k+1$ and j, we obtain that there exist m and $Tset$ such that $m \geq k+1$, $m < j$, $actions(r)(m) = (Recover(Tset), sf(x))$ and $states(r)(k+1)(sf(x)) = states(r)(m)(sf(x))$. This implies $T \in precommitted(states(r)(m)(sf(x)))$ and $x \in sxlocks(states(r)(m)(sf(x)))(T)$. The effect of a restart action now implies $x \in xlocks(states(r)(m+1)(sf(x)))(T)$, i.e. locks of T are restored from the stable memory. Contradiction, because in state $states(r)(k)(sf(x))$, T released its exclusive lock on x *for the last time*.

Finally, suppose $T \in active(states(r)(k)(sf(x)))$. The effect of a crash action implies $T \in active(states(r)(k+1)(sf(x)))$. We again apply lemma $UnlockLem6$ for indexes $k+1$ and j, and obtain that there exist m and $Tset$ such that $m \geq k+1$, $m < j$, $actions(r)(m) = (Recover(Tset), sf(x))$ and $states(r)(k+1)(sf(x)) = states(r)(m)(sf(x))$.
Thus $T \in active(states(r)(m)(sf(x)))$. The precondition of a restart action implies $T \in Tset$. The definition of transaction aborts from section 4.4 now implies $Aborts(T, sf(x), m, actions(r))$, and this completes the proof of lemma $UnlockLem$.

4.6.1.4 Proof of lemma $CommitInv$

Suppose $T \in aborted(states(r)(i)(sf(x)))$ or $T \in committed(states(r)(i)(sf(x))))$, $T2 \neq T$, $Commits(T2, sf(x), i, actions(r))$, $x \in xlocks(states(r)(i)(sf(x)))(T2)$, $j > i$ and $\neg crashed(states(r)(j)(sf(x)))$. We need to prove $\neg ControlsValue(T, x, j, r)$ and $\neg InLog(T, x, j, r)$. By applying lemma $LockInv$. we obtain $ControlsValue(T2, x, i, r)$. This and the effect of a commit action imply $ControlsValue(T2, x, i+1, r)$, which implies $\neg ControlsValue(T, x, i+1, r)$. It is also easy to see that the effect of a commit action implies $\neg InLog(T, x, i+1, r)$, and that T still belongs to one of the sets $aborted$ or $committed$ in state $states(r)(i+1)(sf(x))$. To complete the proof, it is now sufficient to apply the following invariant $CommitInv2$ for $l = i+1$ and $n = j-i-1$, which is proved by induction on n:

$$\begin{array}{l}\forall T, x, i, k, r : \\ \quad ((T \in aborted(states(r)(l)(sf(x))) \vee T \in committed(states(r)(l)(sf(x)))) \ \& \\ \quad \neg ControlsValue(T, x, l, r) \ \& \ \neg InLog(T, x, l, r) \ \& \\ \quad \neg crashed(states(r)(l+n)(sf(x)))) \Longrightarrow \\ \qquad (\neg ControlsValue(T, x, l+n, r) \ \& \ \neg InLog(T, x, l+n, r)) \qquad \text{CommitInv2}\end{array}$$

4.6.2 Reads of non-initial values

The case when a transaction reads a value that is not produced by the initial transaction $T0$ is represented by lemma $LemNI$:

$$\forall r, T2, T1, x, j : name(actions(r)(j)) = Read(T2, x, T1) \ \& \ T1 \neq T0) \Longrightarrow$$

$$\exists i : i < j \;\&\; name(actions(r)(i)) = Write(T1, x) \;\&\; (T1 = T2 \vee CommitsBetween(T1, x, i, j, actions(r))) \;\&\; NoWritesBetween(x, i, j, actions(r))) \qquad \text{LemNI}$$

In addition to invariants established in section 4.6.1, the proof of lemma $LemNI$ uses two other invariants $ValueInv$ and $AbortInv$. Below we show these invariants and briefly describe how they are proved.

$$\forall T, x, k, r : (T \neq T0 \;\&\; \forall j : j \leq k-1 \Longrightarrow name(actions(r)(j)) \neq Write(T, x)) \Longrightarrow (\neg ControlsValue(T, x, k, r) \;\&\; \neg InLog(T, x, k, r)) \qquad \text{ValueInv}$$

Invariant $ValueInv$ expresses that a transaction can control the value of a data item or its log value only if it previously wrote to that data item. It is proved by induction on k, by showing that stable, precommitted or committed value of x is equal to T only if volatile value of x is equal to T in some previous state, and that volatile value of x becomes equal to T *for the first time* only as a result of a write action of T on x.

$$\forall T, x, i, k, r : (Aborts(T, sf(x), i, actions(r)) \;\&\; \neg crashed(states(r)(i+1+k)(sf(x)))) \Longrightarrow (\neg ControlsValue(T, x, i+1+k, r) \;\&\; \neg InLog(T, x, i+1+k, r)) \qquad \text{AbortInv}$$

Invariant $AbortInv$ expresses that an aborted transaction does not control the value of a data item or its log value in any non-crashed state. We don't give the proof of its invariant, because it is rather similar to the proof of invariant $CommitInv$.

4.6.2.1 Proof of lemma $LemNI$

Assuming invariants $ValueInv$ and $AbortInv$, we present the proof of lemma $LemNI$. Suppose $name(actions(r)(j)) = Read(T2, x, T1)$ and $T1 \neq T0$. Let's denote $Si = actions(r)$. The precondition of a read action implies $ControlsValue(T1, x, j, r)$. Lemma $ValueInv$ implies that there exists i such that $i < j$ and $name(Si(i)) = Write(T1, x)$. If $T1 \neq T2$, we must prove $CommitsBetween(T1, x, i, j, Si)$. The effect of a write action implies that in state $states(r)(i+1)$, x is locked by $T1$ in an exclusive mode. The preconditions of write actions imply that in state $states(r)(j)$, x is not locked in an exclusive mode by $T1$. By applying lemma $UnlockLem$ from section 4.6.1, we obtain that there exists index k such that $k \geq i+1$, $k < j$ and either $Commits(T1, sf(x), k, Si)$ or $Aborts(T1, sf(x), k, Si)$. If $Commits(T1, sf(x), k, Si)$, we are done. Suppose $Aborts(T1, sf(x), k, Si)$. By applying lemma $AbortInv$, we obtain a contradiction with $ControlsValue(T1, x, j, r)$.

It remains to prove $NoWritesBetween(x, i, j, Si))$. Suppose $i < l, l < j$ and $name(Si(l)) = Write(T3, x)$. We must prove $AbortsBetween(T3, x, l, j, Si)$. By applying the same techniques as in section 4.6.1, we prove that $T1$ does not release its exclusive lock on x between indexes i and k, and it implies $l > k$. By applying lemma $UnlockLem$ again, we obtain that there exists index m such that $m \geq l+1, m < j$ and either $Commits(T3, sf(x), m, Si)$ and $x \in xlocks(states(r)(m)(sf(x)))(T3)$, or $Aborts(T3, sf(x), m, Si)$. If $Aborts(T3, sf(x), m, Si)$, we are done. Suppose $Commits(T3, sf(x), m, Si)$ and $x \in xlocks(states(r)(m)(sf(x)))(T3)$. It is easy to see that in state $states(r)(m)$, $T1$ belongs to the set *committed*. By applying lemma $CommitInv$ for $T1$ and $T3$, we obtain a contradiction with $ControlsValue(T1, x, j, r)$, and this completes the proof of lemma $LemNI$.

4.7 Verification of serializability

We use our method from section 4.1.2 to verify conflict serializability for our protocol, and define a timestamp as follows: for a schedule Si, $TSp(Si)$ assigns to each transaction committed in Si the index of its first commit action, and assigns 0 to all other transactions. It is sufficient to prove that TSp it is a conflict preserving timestamp for each schedule accepted by our protocol. The proof uses the properties AC1 and AC2.

$$\forall r : (AC1(actions(r)) \ \&\ AC2(actions(r))) \Longrightarrow CPTi(actions(r), TSp(actions(r))) \quad \text{TSpCPT}$$

To prove lemma $TSpCPT$, suppose $Conflict\_i(actions(r))(T1, T2)$ for arbitrary r, $T1$ and $T2$. Let's denote $Si = actions(r)$. We need to prove $TSp(Si)(T1) < TSp(Si)(T2)$. Two cases are possible.

Case 1. A write action conflicts with a following write or read action in Si. Then $Si(i) = (Write(T1, x), sf(x))$ and either $Si(j) = (Write(T2, x), sf(x))$ or $Si(j) = (Read(T2, x), sf(x))$ for some x, i and j such that $i < j$. The effect of a write action implies that in state $states(r)(i+1)$, x is locked by $T1$ in an exclusive mode. The preconditions of write and read actions imply that in state $states(r)(j)$, x is not locked in an exclusive mode by $T1$. By applying lemma $UnlockLem$ from section 4.6.1, we obtain that there exists index k such that $k \geq i+1$, $k < j$ and either $Commits(T1, sf(x), k, Si)$ or $Aborts(T1, sf(x), k, Si)$. Suppose $Aborts(T1, sf(x), k, Si)$. The definition of $Conflict\_i$ implies $Commits(T1, Si)$. By applying lemma $DCLem$ from section 4.5 and the property $AC1$, we obtain a contradiction with decision consistency for Si. Suppose $Commits(T1, sf(x), k, Si)$. The definition of TSp implies $TSp(Si)(T1) \leq k$. The property $AC2$ and the fact that $T2$ also commits in Si imply that there exists an index l such that $Precommits(T2, sf(x), l, Si)$. Using the effect of a precommit action and the preconditions of write and read actions, we prove that a transaction does not perform any write or read actions at any site after it precommits at this site. Therefore $l > j$. By applying AC2 again, we obtain that $T2$ can commit at *any* site only after l. The definition of TSp now imply $TSp(Si)(T2) > l$, and finally $TSp(Si)(T2) > j$. Therefore $TSp(Si)(T1) < TSp(Si)(T2)$, and this completes the proof of case 1.

Case 2. A read action conflicts with a following write action in Si. Then $Si(i) = (Read(T1, x), sf(x))$ and $Si(j) = (Write(T2, x), sf(x))$ for some x, i and j such that $i < j$. The effect of a read action implies that in state $states(r)(i+1)$, x is locked by $T1$ in a shared or exclusive mode. The precondition of write action implies that in state $states(r)(j)$, x is not locked in a shared or exclusive mode by $T1$. We now use an additional lemma, similar to lemma $UnlockLem$ in section 4.6.1.

$$\begin{array}{l}
\forall T, x, i, j, r : \\
((x \in slocks(states(r)(i)(sf(x)))(T) \ \vee\ x \in xlocks(states(r)(i)(sf(x)))(T)) \ \& \\
i < j \ \& \ x \notin slocks(states(r)(j)(sf(x)))(T) \ \& \ x \notin xlocks(states(r)(j)(sf(x)))(T) \ \& \\
\neg crashed(states(r)(j)(sf(x)))) \Longrightarrow \\
\qquad \exists k : i \leq k \ \& \ k < j \ \& \\
\qquad\quad (Commits(T, sf(x), k, actions(r)) \ \vee\ Aborts(T, sf(x), k, actions(r))) \quad \text{SUnlockLem}
\end{array}$$

Lemma $SUnlockLem$ implies that there exists index k such that $k \geq i+1$, $k < j$ and either $Commits(T1, sf(x), k, Si)$ or $Aborts(T1, sf(x), k, Si)$. By applying the same reasoning as in

case 1, we consequently obtain $Commits(T1, sf(x), k, Si)$, $TSp(Si)(T1) \leq k$ and $TSp(Si)(T2) > j$. Therefore $TSp(Si)(T1) < TSp(Si)(T2)$, and this completes the proof of case 2 and lemma $TSpCPT$.

4.7.0.2 Proof of lemma SUnlockLem

To prove lemma $SUnlockLem$, we need a number of additional invariants about shared locks of committed, aborted and precommitted transactions.

$$\forall T, x, i, r : T \in committed(states(r)(i)(sf(x))) \Longrightarrow$$
$$(slocks(states(r)(i)(sf(x)))(T) = \emptyset \;\&\; sslocks(states(r)(i)(sf(x)))(T) = \emptyset) \qquad \text{SLockInv4}$$

$$\forall T, x, i, r : T \in aborted(states(r)(i)(sf(x))) \Longrightarrow$$
$$(slocks(states(r)(i)(sf(x)))(T) = \emptyset \;\&\; sslocks(states(r)(i)(sf(x)))(T) = \emptyset) \qquad \text{SLockInv5}$$

$$\forall T, x, i, r :$$
$$(T \in precommitted(states(r)(i)(sf(x))) \;\&\; \neg crashed(states(r)(i)(sf(x))) \;\&$$
$$x \in slocks(states(r)(i)(sf(x)))(T)) \Longrightarrow x \in sslocks(states(r)(i)(sf(x)))(T) \qquad \text{SUnlockLem5}$$

Invariants $SLockInv4$ and $SLockInv5$, similar to invariants $LockInv4$ and $LockInv5$ from section 4.6.1.2, express that a committed (aborted) transaction cannot hold a shared lock or a stable shared lock on any data item. Invariant $SUnlockLem5$, similar to invariant $UnlockLem5$ from section 4.6.1.3, expresses that a precommitted transaction duplicates all its shared locks in stable memory. After invariants $SLockInv4$, $SLockInv5$ and $SUnlockLem5$ are proved, we can finish the proof by using the same techniques and invariants as in the proof of lemma $SUnlockLem$ in section 4.6.1.3. This is why we don't give a complete proof of lemma $SUnlockLem$.

4.8 PVS specifications

4.8.1 Actions and schedules

```
ActNames [Transactions, DataItems : TYPE] : DATATYPE
BEGIN

Read(tr : Transactions, vr : DataItems, twrt : Transactions) : Read?
Write(tw : Transactions, vw : DataItems) : Write?
Flush(vf : DataItems) : Flush?
Precommit(tpc : Transactions) : Precommit?
Commit(tc : Transactions) : Commit?
Abort(ta : Transactions) : Abort?
Crash : Crash?
Restart(Tset : setof[Transactions]) : Restart?

END ActNames



Schedules : THEORY
BEGIN
```

```
Transactions, DataItems, Sites : TYPE+

T0 : Transactions
sf : [DataItems -> Sites]

IMPORTING ActNames [Transactions, DataItems]

Actions : TYPE = [# act : ActNames, site : Sites #]

Schedules0 : TYPE = sequence[Actions]

st, site1, st1, st2 : VAR Sites
T, T1, T2, T3 : VAR Transactions
x, y, z : VAR DataItems
i, j, k, l, m : VAR nat
Si0 : VAR Schedules0

Sched1(Si0) : bool = FORALL i :
 (Read?(act(Si0(i))) => sf(vr(act(Si0(i)))) = site(Si0(i))) &
 (Write?(act(Si0(i))) => sf(vw(act(Si0(i)))) = site(Si0(i))) &
 (Flush?(act(Si0(i))) => sf(vf(act(Si0(i)))) = site(Si0(i)))

Sched2(Si0) : bool = FORALL i :
 (Read?(act(Si0(i))) => tr(act(Si0(i))) /= T0) &
 (Write?(act(Si0(i))) => tw(act(Si0(i))) /= T0) &
 (Commit?(act(Si0(i))) => tc(act(Si0(i))) /= T0) &
 (Precommit?(act(Si0(i))) => tpc(act(Si0(i))) /= T0) &
 (Abort?(act(Si0(i))) => ta(act(Si0(i))) /= T0) &
 (Restart?(act(Si0(i))) => (NOT Tset(act(Si0(i)))(T0)))

Schedules : TYPE = { Si0 | Sched1(Si0) & Sched2(Si0) }

Si, Si1, Si2 : VAR Schedules

commits(T, st, i, Si) : bool =
   Commit?(act(Si(i))) & tc(act(Si(i))) = T & site(Si(i)) = st

commits(T, st, Si) : bool = EXISTS i : commits(T, st, i, Si)

aborts(T, st, i, Si) : bool =
   ((Abort?(act(Si(i))) & ta(act(Si(i))) = T) OR
    (Restart?(act(Si(i))) & Tset(act(Si(i)))(T))) &
       site(Si(i)) = st

aborts(T, st, Si) : bool = EXISTS i : aborts(T, st, i, Si)

decides(T, st, Si) : bool = commits(T, st, Si) OR aborts(T, st, Si)

Active(T, Si) : bool = (T = T0) OR
   EXISTS i : (Read?(act(Si(i))) & tr(act(Si(i))) = T) OR
              (Write?(act(Si(i))) & tw(act(Si(i))) = T)
```

```
Commits(T, (Si | Active(T, Si))) : bool =
   (T = T0) OR EXISTS st : commits(T, st, Si)

Aborts(T, (Si | Active(T, Si))) : bool = NOT Commits(T, Si)

Read(T2, x, T1, Si, i) : bool =
   Read?(act(Si(i))) & tr(act(Si(i))) = T2 &
   vr(act(Si(i))) = x & twrt(act(Si(i))) = T1

Read(T2, x, Si, i) : bool =
   Read?(act(Si(i))) & tr(act(Si(i))) = T2 & vr(act(Si(i))) = x

Write(T1, x, Si, j) : bool =
   Write?(act(Si(j))) & tw(act(Si(j))) = T1 & vw(act(Si(j))) = x

END Schedules
```

4.8.2 States

```
TNU [Transactions : TYPE] : DATATYPE
BEGIN

trans(trid : Transactions) : trans?

nothing : nothing?

undefined : undefined?

END TNU


States [Transactions, DataItems : TYPE+] : THEORY
BEGIN

IMPORTING TNU[Transactions]

Locks : TYPE = [# xlocks : [Transactions -> setof[DataItems]],
                  slocks : [Transactions -> setof[DataItems]] #]

Log : TYPE = [#  active       : setof[Transactions],
                 precommitted : setof[Transactions],
                 committed    : setof[Transactions],
                 aborted      : setof[Transactions],
                 cvalue       : [DataItems -> TNU],
                 pcvalue      : [DataItems -> TNU],
                 slocks       : Locks,
                 updated      : setof[DataItems],
                 crashed      : bool #]

States : TYPE = [# vvalue : [DataItems -> TNU],
                   locks  : Locks,
                   svalue : [DataItems -> TNU],
                   log    : Log #]
```

```
END States
```

4.8.3 Atomicity and durability

```
AtomDur : THEORY
BEGIN

IMPORTING Schedules

st, site1, st1, st2 : VAR Sites
T, T1, T2, T3 : VAR Transactions
x, y, z : VAR DataItems
i, j, k, l, m : VAR nat
int1 : VAR int
Si, Si1, Si2 : VAR Schedules

DecisionConsistency(Si) : bool = FORALL (T | Active(T, Si)):
   (Commits(T, Si) => FORALL st: (NOT aborts(T, st, Si))) &
   (Aborts(T, Si) => FORALL st: (NOT commits(T, st, Si)))

CommitsBetween(T, x, i, j, Si) : bool =
   EXISTS k : i < k & k < j & commits(T, sf(x), k, Si)

AbortsBetween(T, x, i, j, Si) : bool =
   EXISTS k : i < k & k < j & aborts(T, sf(x), k, Si)

NoWritesBetween(x, int1, j, Si) : bool =
   FORALL l, T : (int1 < l & l < j & Write(T, x, Si, l)) =>
      AbortsBetween(T, x, l, j, Si)

UpdateInPlace(Si) : bool = FORALL T2, T1, x, j:
   (Read(T2, x, T0, Si, j) => NoWritesBetween(x, -1, j, Si)) &
   ((Read(T2, x, T1, Si, j) & T1 /= T0) =>
        EXISTS i: i < j & Write(T1, x, Si, i) &
                  (T1 = T2 OR CommitsBetween(T1, x, i, j, Si)) &
                  NoWritesBetween(x, i, j, Si))

Durability(Si) : bool = FORALL T2, T, T1, x, i, j:
 ((NoWritesBetween(x, -1, j, Si) & Read(T2, x, T, Si, j)) => T = T0) &
 ((Write(T1, x, Si, i) & i < j & CommitsBetween(T1, x, i, j, Si) &
   NoWritesBetween(x, i, j, Si) & Read(T2, x, T, Si, j)) => T = T1)

AbortRead(Si) : bool = FORALL T, x, i, j:
   (aborts(T, sf(x), i, Si) & j > i) => (NOT Read(T, x, Si, j))

UpdDurab : LEMMA
   (UpdateInPlace(Si) & DecisionConsistency(Si) & AbortRead(Si)) =>
       Durability(Si)

END AtomDur
```

4.8.4 Fault-tolerant serializability and method for its verification

```
FTCS : THEORY
BEGIN

AtomicActions : TYPE = {R, W}

IMPORTING Schedules, fin_seq[Actions], min,
   AcycOrd [Transactions, DataItems, AtomicActions, R, W ]

act1 : VAR Actions
Si : VAR Schedules
S : VAR finite_sequence[Actions]
T, T1, T2 : VAR Transactions
x, y : VAR DataItems
i, j, k, n : VAR nat
trset : VAR setof[Transactions]
TS : VAR [Transactions -> nat]

transform(act1 | Read?(act(act1)) OR Write?(act(act1))) : Actions1 =
   IF Read?(act(act1))
   THEN (# act := R, tr := tr(act(act1)), vari := vr(act(act1)) #)
   ELSE (# act := W, tr := tw(act(act1)), vari := vw(act(act1)) #) ENDIF

Extr0(act1, trset) : Schedules1 =
   IF ((Read?(act(act1)) & trset(tr(act(act1)))) OR
   (Write?(act(act1)) & trset(tw(act(act1))))) THEN one(transform(act1))
   ELSE empty_seq ENDIF

Extr(S, trset) : RECURSIVE Schedules1 =
   IF length(S) = 0 THEN empty_seq
   ELSE Extr(prefix(S), trset) o Extr0(last(S), trset) ENDIF
      MEASURE length(S)

prefix(Si, n) : finite_sequence[Actions] =
   (# length := n, seq := (LAMBDA (k : below[n]): Si(k)) #)

CommitSet(Si) : setof[Transactions] =
   LAMBDA T : Active(T, Si) & Commits(T, Si) & T /= T0

FTCS(Si) : bool =
   FORALL n : Conf_serializable(Extr(prefix(Si, n), CommitSet(Si)))

Conflict_i(Si)(T1, T2) : bool =
   (T1 /= T2) & CommitSet(Si)(T1) & CommitSet(Si)(T2) &
   EXISTS i, j, x : i < j &
      ((Write(T1, x, Si, i) & Write(T2, x, Si, j)) OR
       (Write(T1, x, Si, i) & Read(T2, x, Si, j)) OR
       (Read(T1, x, Si, i) & Write(T2, x, Si, j)))

CPTi(Si, TS) : bool =
   FORALL T1, T2 : Conflict_i(Si)(T1, T2) => TS(T1) < TS(T2)
```

```
Ordered_i(Si) : bool = EXISTS TS : CPTi(Si, TS)

OrdConfInfLem : LEMMA
   CPTi(Si, TS) => CPT(Extr(prefix(Si, n), CommitSet(Si)), TS)

OrdConfInf : THEOREM Ordered_i(Si) => FTCS(Si)

ComIndexes(T, Si) : setof[nat] =
   { i | Commit?(act(Si(i))) & tc(act(Si(i))) = T }

TSp(Si) : Timestamp =
   LAMBDA T : IF nonempty?(ComIndexes(T, Si))
              THEN minimum(ComIndexes(T, Si))
              ELSE 0 ENDIF

TSpLem : LEMMA CPTi(Si, TSp(Si)) => FTCS(Si)

END FTCS
```

4.8.5 State machine for the protocol

```
Runs : THEORY
BEGIN

IMPORTING Schedules, States[Transactions, DataItems]

GlobalStates : TYPE = [Sites -> States]

gs, gs0, gs1, gs2 : VAR GlobalStates
site, site1, st, st1, st2 : VAR Sites
T, T1, T2, T3 : VAR Transactions
x, y, z : VAR DataItems
i, j, k, l, m : VAR nat
tsset :VAR setof[Transactions]
a1, a2, a3 : VAR Actions

is : GlobalStates = LAMBDA site :
 (# vvalue  := LAMBDA x : IF sf(x) = site THEN nothing
                          ELSE undefined ENDIF,
    locks := (# xlocks := LAMBDA T : emptyset[DataItems],
                slocks := LAMBDA T : emptyset[DataItems] #),
    svalue  := LAMBDA x : IF sf(x) = site THEN trans(T0)
                          ELSE undefined ENDIF,
    log   := (# cvalue       := LAMBDA x : IF sf(x) = site
                                           THEN trans(T0)
                                           ELSE undefined ENDIF,
                pcvalue      := LAMBDA x : IF sf(x) = site
                                           THEN nothing
                                           ELSE undefined ENDIF,
                updated      := LAMBDA x : FALSE,
                active       := emptyset[Transactions],
                committed    := singleton[Transactions](T0),
                aborted      := emptyset[Transactions],
```

```
                precommitted := emptyset[Transactions],
                slocks       :=
                   (# xlocks := LAMBDA T : emptyset[DataItems],
                      slocks := LAMBDA T : emptyset[DataItems] #),
                crashed      := FALSE #) #)

Pre(gs1, a1) : bool = IF Restart?(act(a1))
                      THEN crashed(log(gs1(site(a1)))) = TRUE
                      ELSE crashed(log(gs1(site(a1)))) = FALSE ENDIF

FlushEffect(gs1, gs2, x, site) : bool =
   trans?(vvalue(gs1(site))(x)) &
   gs2 = gs1 WITH [site := gs1(site) WITH
      [svalue := svalue(gs1(site)) WITH [x := vvalue(gs1(site))(x)],
       vvalue := vvalue(gs1(site)) WITH [x := nothing] ]]

CommitEffect(gs1, gs2, T, site) : bool =
   (NOT (committed(log(gs1(site)))(T) OR aborted(log(gs1(site)))(T))) &
   (NOT active(log(gs1(site)))(T)) & precommitted(log(gs1(site)))(T) &
   gs2 = gs1 WITH [site := gs1(site) WITH
      [log := log(gs1(site)) WITH
          [cvalue := LAMBDA x : IF xlocks(locks(gs1(site)))(T)(x)
                                THEN trans(T)
                                ELSE cvalue(log(gs1(site)))(x) ENDIF,
           pcvalue := LAMBDA x : IF xlocks(locks(gs1(site)))(T)(x)
                                 THEN nothing
                                 ELSE pcvalue(log(gs1(site)))(x) ENDIF,
           committed := add(T, committed(log(gs1(site)))),
           precommitted := remove(T, precommitted(log(gs1(site)))),
           slocks := slocks(log(gs1(site))) WITH
              [xlocks := xlocks(slocks(log(gs1(site)))) WITH
                         [T := emptyset[DataItems]],
               slocks := slocks(slocks(log(gs1(site)))) WITH
                         [T := emptyset[DataItems]] ]],
        locks := locks(gs1(site)) WITH
              [xlocks := xlocks(locks(gs1(site))) WITH
                         [T := emptyset[DataItems]],
               slocks := slocks(locks(gs1(site))) WITH
                         [T := emptyset[DataItems]] ]]]

AbortEffect(gs1, gs2, T, site) : bool =
   (NOT (committed(log(gs1(site)))(T) OR aborted(log(gs1(site)))(T))) &
   (active(log(gs1(site)))(T) OR precommitted(log(gs1(site)))(T)) &
   gs2 = gs1 WITH [site := gs1(site) WITH
      [vvalue := LAMBDA x : IF xlocks(locks(gs1(site)))(T)(x)
                            THEN cvalue(log(gs1(site)))(x)
                            ELSE vvalue(gs1(site))(x) ENDIF,
       log := log(gs1(site)) WITH
          [pcvalue := LAMBDA x : IF xlocks(locks(gs1(site)))(T)(x)
                                 THEN nothing
                                 ELSE pcvalue(log(gs1(site)))(x) ENDIF,
           aborted := add(T, aborted(log(gs1(site)))),
           active := remove(T, active(log(gs1(site)))),
```

```
           precommitted := remove(T, precommitted(log(gs1(site)))),
           slocks := slocks(log(gs1(site))) WITH
              [xlocks := xlocks(slocks(log(gs1(site)))) WITH
                         [T := emptyset[DataItems]],
               slocks := slocks(slocks(log(gs1(site)))) WITH
                         [T := emptyset[DataItems]] ]],
        locks := locks(gs1(site)) WITH
              [xlocks := xlocks(locks(gs1(site))) WITH
                         [T := emptyset[DataItems]],
               slocks := slocks(locks(gs1(site))) WITH
                         [T := emptyset[DataItems]] ]]]

PrecommitEffect(gs1, gs2, T, site) : bool =
   (NOT (committed(log(gs1(site)))(T) OR aborted(log(gs1(site)))(T))) &
   (NOT precommitted(log(gs1(site)))(T)) & active(log(gs1(site)))(T) &
   gs2 = gs1 WITH [site := gs1(site) WITH
      [log := log(gs1(site)) WITH
          [pcvalue := LAMBDA x : IF xlocks(locks(gs1(site)))(T)(x)
                                 THEN trans(T)
                                 ELSE pcvalue(log(gs1(site)))(x) ENDIF,
           precommitted := add(T, precommitted(log(gs1(site)))),
           active := remove(T, active(log(gs1(site)))),
           slocks := slocks(log(gs1(site))) WITH
              [xlocks := xlocks(slocks(log(gs1(site)))) WITH
                         [T := xlocks(locks(gs1(site)))(T)],
               slocks := slocks(slocks(log(gs1(site)))) WITH
                         [T := slocks(locks(gs1(site)))(T)] ]]]]

ReadEffect(gs1, gs2, T2, x, T1, site) : bool =
   (NOT (committed(log(gs1(site)))(T2) OR aborted(log(gs1(site)))(T2))) &
   (NOT precommitted(log(gs1(site)))(T2)) &
   (IF trans?(vvalue(gs1(site))(x)) THEN T1 = trid(vvalue(gs1(site))(x))
    ELSE (trans?(svalue(gs1(site))(x)) &
          T1 = trid(svalue(gs1(site))(x))) ENDIF) &
   (FORALL T3 : T3 /= T2 => (NOT xlocks(locks(gs1(site)))(T3)(x))) &
   gs2 = gs1 WITH [site := gs1(site) WITH
      [vvalue := vvalue(gs1(site)) WITH [x := trans(T1)],
       log := log(gs1(site)) WITH
                 [active := add(T2, active(log(gs1(site))))],
       locks := locks(gs1(site)) WITH
          [slocks := slocks(locks(gs1(site))) WITH
             [T2 := IF xlocks(locks(gs1(site)))(T2)(x) THEN
                    slocks(locks(gs1(site)))(T2)
                    ELSE add(x, slocks(locks(gs1(site)))(T2)) ENDIF] ]]]

WriteEffect(gs1, gs2, T, x, site) : bool =
   (NOT (committed(log(gs1(site)))(T) OR aborted(log(gs1(site)))(T))) &
   (NOT precommitted(log(gs1(site)))(T)) &
   (FORALL T3 : (NOT xlocks(locks(gs1(site)))(T3)(x)) &
                (T3 /= T => (NOT slocks(locks(gs1(site)))(T3)(x)))) &
   gs2 = gs1 WITH [site := gs1(site) WITH
      [vvalue := vvalue(gs1(site)) WITH [x := trans(T)],
       log := log(gs1(site)) WITH
```

```
          [updated := updated(log(gs1(site))) WITH [x := TRUE],
           active := add(T, active(log(gs1(site))))],
       locks := locks(gs1(site)) WITH
          [xlocks := xlocks(locks(gs1(site))) WITH
                      [T := add(x, xlocks(locks(gs1(site)))(T))],
           slocks := slocks(locks(gs1(site))) WITH
                      [T := remove(x, slocks(locks(gs1(site)))(T))] ]]]

CrashEffect(gs1, gs2, site) : bool =
   gs2 = gs1 WITH [site := gs1(site) WITH
      [vvalue  := LAMBDA x : IF sf(x) = site THEN nothing
                             ELSE vvalue(gs1(site))(x) ENDIF,
       log := log(gs1(site)) WITH [ crashed := TRUE ],
       locks := (# xlocks := LAMBDA T : emptyset[DataItems],
                   slocks := LAMBDA T : emptyset[DataItems] #) ]]

RecoverEffect(gs1, gs2, tsset, site) : bool =
   tsset = active(log(gs1(site))) &
   gs2 = gs1 WITH [site := gs1(site) WITH
      [vvalue  := LAMBDA x :
          IF sf(x) = site THEN
             IF updated(log(gs1(site)))(x) THEN
                 IF (trans?(pcvalue(log(gs1(site)))(x)) &
                    precommitted(log(gs1(site)))
                       (trid(pcvalue(log(gs1(site)))(x))))
                 THEN pcvalue(log(gs1(site)))(x)
                 ELSE cvalue(log(gs1(site)))(x) ENDIF
             ELSE nothing ENDIF
        ELSE vvalue(gs1(site))(x) ENDIF,
       log := log(gs1(site)) WITH
          [aborted :=
             union(active(log(gs1(site))), aborted(log(gs1(site)))),
           active := emptyset[Transactions],
           crashed := FALSE ],
       locks := locks(gs1(site)) WITH
          [xlocks := LAMBDA T : xlocks(slocks(log(gs1(site))))(T),
           slocks := LAMBDA T : slocks(slocks(log(gs1(site))))(T)] ]]

Effect(gs1, a1, (gs2 | Pre(gs1, a1))) : bool =
   CASES act(a1) OF

Read(T2, x, T1): ReadEffect(gs1, gs2, T2, x, T1, site(a1)),
Write(T, x):     WriteEffect(gs1, gs2, T, x, site(a1)),
Flush(x):        FlushEffect(gs1, gs2, x, site(a1)),
Precommit(T):    PrecommitEffect(gs1, gs2, T, site(a1)),
Commit(T):       CommitEffect(gs1, gs2, T, site(a1)),
Abort(T):        AbortEffect(gs1, gs2, T, site(a1)),
Crash:           CrashEffect(gs1, gs2, site(a1)),
Restart(tsset):  RecoverEffect(gs1, gs2, tsset, site(a1))

   ENDCASES

PreRuns : TYPE = [# states  : sequence[GlobalStates],
```

```
                    actions : Schedules #]

Runs : TYPE = { pr : PreRuns | states(pr)(0) = is &
   (FORALL i : Pre(states(pr)(i), actions(pr)(i)) &
               Effect(states(pr)(i),actions(pr)(i),states(pr)(i + 1)))}

r : VAR Runs

ValueExists : LEMMA trans?(svalue(states(r)(i)(sf(x)))(x)) &
                    trans?(cvalue(log(states(r)(i)(sf(x))))(x))

END Runs
```

4.8.6 Two-Phase Commit

```
TPC : THEORY
BEGIN

IMPORTING Schedules

st, st1, st2 : VAR Sites
T, T1, T2, T3 : VAR Transactions
i, j, k, l, m : VAR nat
Si, Si1, Si2 : VAR Schedules

AC1(Si) : bool = FORALL st1, st2, T:
   (decides(T, st1, Si) & decides(T, st2, Si)) =>
       (commits(T, st1, Si) IFF commits(T, st2, Si))

precommits(T, st, i, Si) : bool =
   Precommit?(act(Si(i))) & tpc(act(Si(i))) = T & site(Si(i)) = st

active(T, st, Si) : bool =
   EXISTS i : ((Read?(act(Si(i))) & tr(act(Si(i))) = T) OR
               (Write?(act(Si(i))) & tw(act(Si(i))) = T))  &
                   site(Si(i)) = st

fn : VAR [Sites -> nat]

AC2(Si) : bool = FORALL st1, T, i:
   commits(T, st1, i, Si) =>
      EXISTS fn: FORALL st: active(T, st, Si) =>
         (fn(st) < i & precommits(T, st, fn(st), Si))
END TPC
```

4.8.7 Main lemmas and theorems

```
Proofs : THEORY
BEGIN

IMPORTING AtomDur, FTCS, Runs, TPC

r, r1, r2 : VAR Runs
```

```
DCLem : LEMMA AC1(actions(r)) => DecisionConsistency(actions(r))

UIPLem : LEMMA UpdateInPlace(actions(r))

TSpCPT : LEMMA
   (AC1(actions(r)) & AC2(actions(r))) =>
       CPTi(actions(r), TSp(actions(r)))

SLemma : LEMMA
   (AC1(actions(r)) & AC2(actions(r))) => FTCS(actions(r))

Main : THEOREM
   (AC1(actions(r)) & AC2(actions(r))) =>
       (DecisionConsistency(actions(r)) & UpdateInPlace(actions(r)) &
        FTCS(actions(r)))

END Proofs
```

Chapter 5

Distributed Atomic Commitment

In this chapter, we present the verification of the non-blocking commitment protocol of Babaoglu and Toueg [BT93a], combined with our own termination protocol for crashed participants. This protocol assumes a *fail-silent* model of crashes, i.e. a participant interacts with other participants either correctly or not at all. A crashed participant (i.e. a participant that is currently not correct) is only allowed to perform some internal activity (e.g., recovering from a memory failure). It is also assumed that communication failures between participants do not happen. For this protocol and communication model, all safety and liveness properties have been verified. As in the previous chapter, the protocol is specified by combination of state machines and global assertions. In the main sections, only mathematical definitions and proofs are presented that are based on PVS specifications and proofs. The actual PVS specifications are given at the end of the chapter.

This chapter is organized as follows. In section 5.1, we describe the communication mechanism used by our protocol. In section 5.2, we present an informal description of the protocol of Babaoglu and Toueg and our termination protocol. We also explain the error in the termination protocol of Babaoglu and Toueg. Section 5.3 gives an overview of our specification method. In the next three sections the method is presented in more detail. Section 5.4 explains the modelling of participants and the coordinator using state machines, whereas section 5.5 describes the modelling of the communication mechanism using global assertions. Section 5.6 presents the specification of the correctness properties including our formalization of the termination property. The verification of these properties is described in section 5.7. Section 5.8 contains the actual PVS specifications.

5.1 Communication mechanism

Our model of communication is *synchronous* in the sense that there are known bounds for both relative speeds of processes and message delays. There are two ways to disseminate a message: send/receive and reliable broadcast. If a message has been disseminated to some group of processes *without* using reliable broadcast, then this action is called $send$, and a corresponding action at the destination is called $receive$. Each message is received within δ time units after being sent, including the time of processing it at the sending and receiving processes. If we use a reliable broadcast mechanism, the two actions are called $broadcast$ and $deliver$ respectively. Each message is delivered within Δ_b time units after being broadcasted and we have $\Delta_b \geq \delta$.

Suppose a process p disseminates a message m to all processes in a set of processes G using either send/receive or broadcast/deliver. If p sends m, it can be informally specified as follows (using the style of Babaoglu and Toueg with some small changes):

S-Validity If p is correct, then all correct processes in G eventually receive m.
S-Integrity Each process in G receives m at most once, and only if p actually sends m.
S-Timeliness If p initiates the sending of m at real-time t, then no process in G receives m after real-time $t + \delta$.

To obtain the specification of reliable broadcast by p from the specification of send/receive by p, we must replace send by broadcast, receive by deliver, δ by Δ_b, and add the *uniform agreement* property, expressing that either all correct processes in G deliver m, or none of them:

B-Validity If p is correct, then all correct processes in G eventually deliver m.
B-Integrity Each process in G delivers m at most once, and only if p actually broadcasts m.
B-Timeliness If p initiates the broadcast of m at real-time t, then no process in G delivers m after real-time $t + \Delta_b$.
B-UniformAgreement If any process (correct or not) in G delivers m, then all correct processes in G eventually deliver m.

Recall from section 2.4 that if in some ACP the coordinator crashes while broadcasting a decision and this decision is delivered only to participants that crash later, then this ACP leads to blocking. It is easy to see that this is the only scenario that may lead to blocking of correct participants. Since such executions are prevented by the uniform agreement property, it is not surprising that the combination of ACP-BT and RBC indeed leads to a correct non-blocking ACP.

5.2 Outline of the protocol

5.2.1 Pseudo code used

To present the protocols informally, we use a Pascal-like pseudo code with the usual sequential control flow structures which is taken from [BT93a] with some small changes. We cite its description from [BT93a]:

> We denote concurrent activities as tasks separated by "//" enclosed within **cobegin** and **coend**. Communication is accomplished through the **send** and **receive** statements by supplying the message and the destination/source process name. ... We use **send** m **to** G as a shorthand for sending message m one at a time to each process that is a member of the set G. Note that we make no assumption about the indivisibility of this operation. In particular, the sender may crash after having sent to some but not all members of the destination set. The receiver of a message may synchronize its execution with the receipt of a message in one of two ways. The **wait-for** statement is used to block the receiver until the receipt of a particular message. If the message may arrive at unspecified times and should be received without blocking the receiver, then the **upon** statement is appropriate. ... When the specified event occurs, execution proceeds with the body of the respective statement. In case of a blocking wait, an optional timeout may be set to trigger at a particular (local) time using the **set-timeout-to** statement. The timeout value in effect is that set by the most recent **set-timeout-to** before the execution of a **wait-for** statement. If the event being waited for does not occur by the specified time, then the **on-timeout** clause of the **wait-for** statement is executed rather than its body. The body and the timeout clause of **wait-for** are mutually exclusive.

Moreover, **broadcast**(m, G) is used to express the broadcast of message m to all members of the set G.

5.2.2 The ACP-BT protocol

For verification purposes the *separation of concerns* concept is applied: only the specification and not the implementation of certain facilities is used. This concerns especially communication, leader election and reliable storage.

For the same reason we clarify and simplify the assumptions about how this protocol starts as compared to [BT93a]. In our model, the transaction begins at real time t_{start}. By real time $C_{know}(p)$ ($C_{know}(p) > t_{start}$) each participant p is assumed to "know" about the operations of the transaction, its full set of participants, and whether it is willing and able to make the transaction's updates permanent.

Participants of the transaction execute some protocol to elect the *coordinator* of this transaction. We are not interested in how the coordinator is elected; it is only assumed that exactly one participant assumes the role of coordinator by real time Ct_{start} such that $Ct_{start} \leq t_{start} + \Delta_c$, where Δ_c is some constant (an "upper bound" on electing a coordinator). For any p, we also require $C_{know}(p) < Ct_{start}$. This leads to the chain of inequalities $t_{start} < C_{know}(p) < Ct_{start} \leq t_{start} + \Delta_c$ for any p. Each participant p starts executing the participant's part of ACP-BT at $C_{know}(p)$, whereas the coordinator starts executing the coordinator's part at Ct_{start}.

The protocol, shown in Fig. 5.1, consists of two concurrent tasks, one executed by the coordinator (task 1) and the other executed by all participants, including the coordinator (task 2). The coordinator starts by sending a $vote\_request$ message to all participants. A participant has to vote when it receives this message. If it votes NO, it can unilaterally decide **abort**. If it votes YES, it waits for the coordinator's decision. Phase 1 ends when the coordinator collects all votes and decides (as already explained in section 2.4).

In Phase 2, the coordinator broadcasts the decision to all participants. Each participant decides upon delivering this decision message. If a participant does not receive a $vote\_request$ message or a decision message before the corresponding time-out expires, it can still decide **abort**, without becoming blocked! Time-out values are chosen in an appropriate way which ensures that if there are no crashes in the system, then each message is received before its time-out expires. To make termination of a transaction possible even at a participant that crashes, variable $state$, which is kept in a stable storage, is used (this issue is discussed in more detail in the next subsection).

5.2.3 The termination protocol

A participant recovering from a crash must execute a termination protocol to conclude all the transactions that were active at this participant at the time of the crash. It can only achieve termination with the help of a log kept by this participant in stable storage (which is called *DT-log* in [BT93a]). The termination protocol of Babaoglu and Toueg represents this DT-log as a sequence of records and describes its management in detail. For instance, it requires each participant p to add a **start** record to the DT-log at real time $C_{know}(p)$, and a **yes** (or **no**) record before sending a YES (or NO) vote to the coordinator. The protocol specifies actions that must be taken if a particular sequence of log records is discovered upon recovery. For instance, a participant must decide **abort** if the part of DT-log regarding transaction is equal to <**start, no**>.

The first difference between our termination protocol and "standard" termination protocols (for instance, the protocol of Babaoglu and Toueg) is a higher level of abstraction in representing the DT-log. Without considering the tedious details of how the log is actually stored and managed, we just assume that a recovered process remembers the value of the variable $state$ at the moment when it crashed, as well as the values of some other variables that are especially important for reaching a decision.

```
procedure atomic_commitment(transaction, participants)
    cobegin
    % Task 1: Executed by the coordinator
        send vote_request to all participants
        set-timeout-to local_clock + 2 * δ
        wait-for (receipt of vote messages from all participants)
            if (all votes are YES) then
                broadcast(commit, participants)
            else broadcast(abort, participants)
        on-timeout broadcast(abort, participants)

//

    % Task 2: Executed by all participants (including the coordinator)
        state := Initial
        set-timeout-to C_know + Δ_c + δ
        wait-for (receipt of vote_request from coordinator)
            if (vote = NO) then state := Aborted else state := Waiting
            send vote to coordinator
            if (vote = NO) then decide abort
            else
                set-timeout-to C_know + Δ_c + 2 * δ + Δ_b
                wait-for (delivery of decision message)
                    (if decision message is abort) then
                        state := Aborted; decide abort
                    else state := Committed; decide commit
                on-timeout state := Aborted; decide abort
        on-timeout state := Aborted; decide abort
    coend
end
```

Figure 5.1: The ACP-BT protocol.

If some participant[1] p discovers upon recovery that $state = Initial$ or $state = Aborted$, it decides **abort** as a result of recovery, thus completing termination. If $state = Committed$, it decides **commit** instead. If $state = Waiting$, it cannot decide independently, so it starts the *requester's component* of the protocol, shown in Fig. 5.2, by sending `help` message to all other participants. If some other participant q is operational, it replies to this message according to the *responder's component*, shown in Fig. 5.3, by sending $reply\_commit$ if $state(q) = Committed$, $reply\_abort$ if $state(q) = Aborted$, $reply\_willing$ if $state(q) = Willing$, and nothing otherwise.

If p then receives $reply\_commit$ or $reply\_abort$, it decides **commit** or **abort** respectively. If it receives $reply\_willing$ from some participant q, it adds q to the set $willing$, representing participants that are also willing to commit. The set $willing$ is also kept in stable storage. In general, all variables except $state$ and $willing$ are kept in volatile storage and may not survive a crash. If all other par-

[1]Like [BT93b], we assume that the coordinator does not recover from crashes (for simplicity reasons, and also because the coordinator is not responsible for making any updates; it is the task of a participant located at the coordinator's site).

```
procedure terminate_transaction(p)
% Executed by recovered participant p
    case state of
        Initial or Aborted:    state := Aborted; decide abort
        Committed:    decide commit
        Waiting or Willing:
            if Waiting then willing := {p}; state := Willing
            while (undecided) do
                send help to all participants not in willing
                set-timeout-to 2 * δ
                wait-for receipt of reply message
                    case reply from q of
                        reply_commit:    state := Committed; decide commit
                        reply_abort:     state := Aborted; decide abort
                        reply_willing:
                            willing := add(q, willing);
                            if willing = all participants
                                then state := Committed; decide commit
                    esac
                on-timeout skip
            od
    esac
end
```

Figure 5.2: The requester's component of the termination protocol

ticipants eventually send $reply\_willing$ to p (which implies that the most malicious *total failure* has occurred, i.e. all participants crashed while waiting for the coordinator's decision), it decides **commit**. This is different from the termination protocol of Babaoglu and Toueg where, if all participants vote YES and then crash while waiting for the **commit** decision, they all decide **abort** during termination; the result is an unnecessary decision to abort a transaction. If p does not decide within $2 * \delta$ time units after sending the request for help, it repeats the request, and this process can continue indefinitely (without further assumptions).

If p crashes again while executing the termination protocol, it will "remember" the set $willing$ upon recovery, and it will not have to contact participants from this set again. This appears to greatly facilitate termination, because some of the willing participants may crash again, and may never re-

```
upon (receipt of help message from p)
        case state of
            Committed:          send reply_commit to p
            Aborted:            send reply_abort to p
            Willing:            send reply_willing to p
            Initial or Waiting: skip
        esac
```

Figure 5.3: The responder's component of the termination protocol

spond to new requests for help. Indeed, in the next section we will show, that our decision to keep the set $willing$ in stable storage makes the assumptions under which a transaction should eventually terminate much weaker.

5.2.4 Error found

The responder's component of the termination protocol of Babaoglu and Toueg makes no distinction between those participants that voted YES and are still waiting for the coordinator's decision, and those that recovered and are willing to commit. All such participants send $reply\_willing$ (in our notation) to a request for help. We show that this may lead to inconsistent decisions.

Consider the following scenario, in which the coordinator is correct and only one participant crashes. Suppose all participants send YES votes to the coordinator at real time $t1$, and all correct ones deliver the **commit** decision at real time $t1 + 1.75 * \delta$ (this is allowed by one of the possible implementations of reliable broadcast, namely UTRB1 in [BT93a]).

Suppose now that some participant p crashes shortly after sending a YES vote and recovers by real time $t1 + 0.5 * \delta$. Then p immediately starts its termination protocol by sending $help$ messages to all other participants. Suppose all of them receive this message by real time $t1 + \delta$. According to the responder's component, they all send $reply\_willing$. Suppose p receives all these replies by real time $t1 + 1.5 * \delta$. Therefore p decides **abort** by real time $t1 + 1.5 * \delta$ (remember the difference with our protocol, where the decision would be **commit**). After that, all other participants deliver the **commit** decision at real time $t1 + 1.75 * \delta$ and they all decide **commit**. Thus p decides **abort** and all other participants decide **commit**. The proof of correctness in [BT93b] ignores such scenarios.

The most obvious way to correct the error is to introduce an additional log record, which would let us distinguish the two states described above.

5.3 Overview of the formal specification

In our model, the coordinator and participants are represented using state machines. From Figures 5.1, 5.2 and 5.3 it can be seen that the actions of the protocol are completely determined by the events (such as crash and recover), the messages and the contents of the variables $state$ and $willing$. The states are therefore chosen in accordance with the contents of these variables. We also introduce 3 additional states: $ComCrashed$, $AbCrashed$ and $WCrashed$. For example, separation was needed between the $ComCrashed$ state and the $AbCrashed$ state to model that a process crashing in the state *Aborted* ($Committed$) recovers to the state *Aborted* ($Committed$).

From the pseudo code all actions associated with a state can be determined. Six actions need to be considered: $send$, $broadcast$, $read$, $crash$, $recover$ and $timeout$, where the first three have a message as a parameter. This determines the possible transitions between the states and the event or message that provokes them. In Figures 5.4 and 5.5 the possible transitions are drawn and labeled with associated actions. These figures do not include the so-called *pseudostates*, i.e. states which a process leaves immediately without performing any activity in them, that are present in our PVS model. E.g., receiving of a help request and sending a reply to this request are "separated" by a pseudostate in our model. We introduced pseudostates to ensure that each action corresponds to exactly one event.

In general, states are represented in PVS by a record with a number of fields. The main fields are:

- $name$, to represent the "name" of the state (e.g., $Initial$, $Committed$, etc.);
- $ident$ is the identifier of the corresponding process;

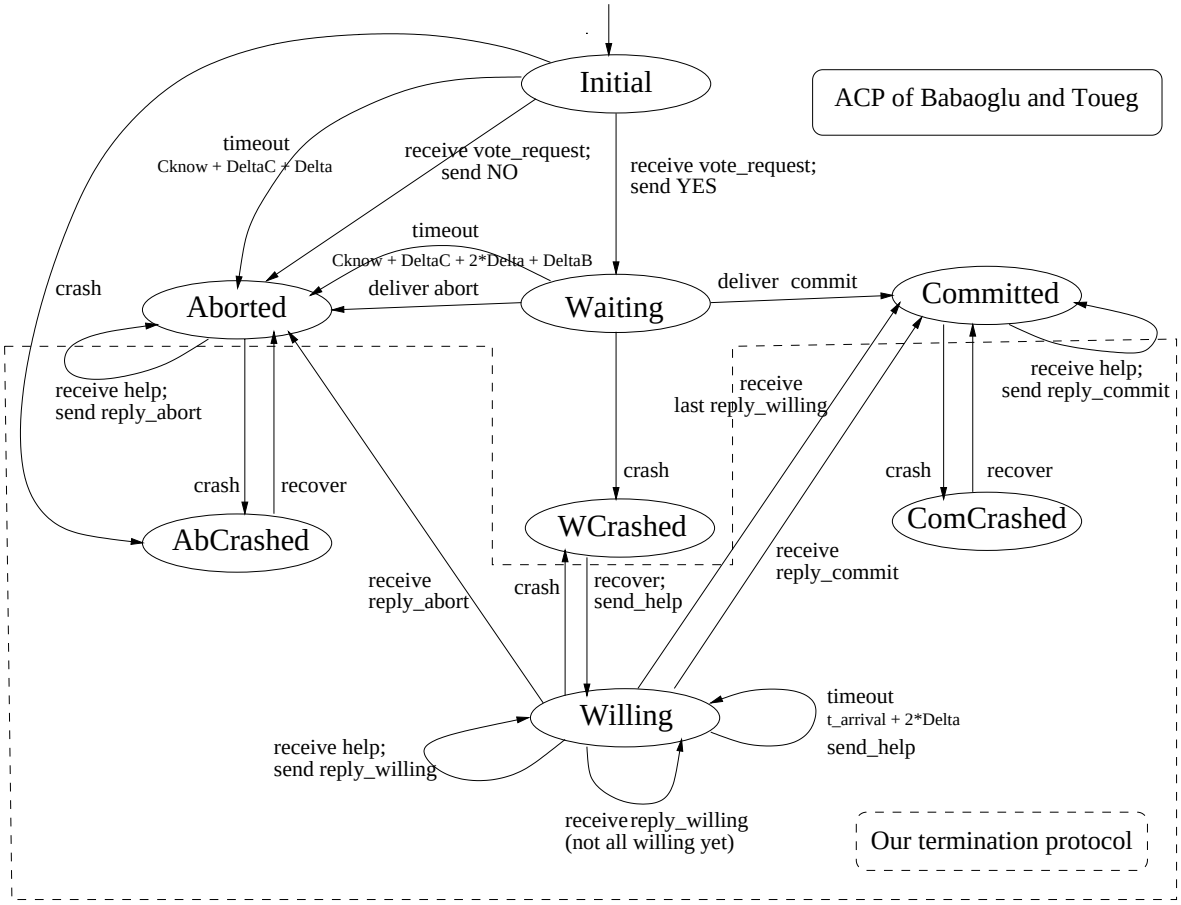

Figure 5.4: Protocol performed by each participant.

- $tout$ is the time-out of the state (i.e. the time moment before which some action should be taken);
- $wait$ is the set of messages the process is waiting for.

Further details about our state machines are given in section 5.4.

The protocol performed by a particular process, such as coordinator and participant, is specified in PVS by defining the precondition of each action as a predicate on the state and also the effect of each action in terms of a state transition. Let $time(a)$ denote the *execution time* of action a. Also let $TimeS(ps)$ denote the *starting time* of process ps.

Then a run of a process is represented by an infinite sequence of the form $s_0 \overset{a_0}{\rightarrow} s_1 \overset{a_1}{\rightarrow} \ldots \overset{a_{i-1}}{\rightarrow} s_i \overset{a_i}{\rightarrow} s_{i+1} \overset{a_{i+1}}{\rightarrow} \ldots$, where the s_i are states and the a_i are executed actions, for $i \in N$. A run of process ps should satisfy a number of conditions:

- each s_i satisfies the precondition of a_i;
- every the pair (s_i, s_{i+1}) corresponds to the effect of a_i;
- $time(a_0) \geq TimeS(ps)$;
- $time(a_{i+1}) \geq time(a_i)$;
- $time(a_i) \leq tout(s_i)$;
- non-Zeno behavior (only finitely many actions in finite time).

We formalize non-Zeno behavior as follows: for any real number L, a run must include an action with execution time greater than L.

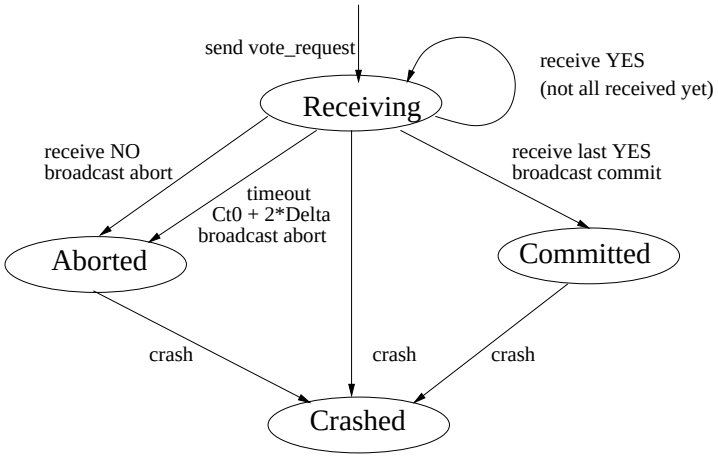

Figure 5.5: Protocol performed by the coordinator.

In this model, we use infinite runs to represent the behavior of a participant that never completes its termination protocol due to failures of others. Therefore it would be possible to include both finite and infinite runs in our model, as in an I/O automata framework [Lyn96]. However, we decided that it would be difficult to work in PVS with both finite and infinite runs, especially because finite runs lead to a large number of additional type-correctness conditions (see section 1.5.2). Therefore, we restrict ourself to infinite runs. To implement this, in each of the "final" states (e.g., $Committed$, $AbCrashed$ etc.), where a process may not have to take any action, a fictitious time-out action is introduced, which moves a process to the same state after some fixed unit of time elapses. In this way each finite run is transformed to an infinite run.

We also associate with each process a record called $Available$, which consists of two sets $received$ and $delivered$. For a process ps, set $received(ps)$ includes all messages received by ps plus their receiving time, and set $delivered(ps)$ includes all messages delivered at ps plus their delivery time. For a message ms and time t, we say that ms is *available* to ps at t if either $(ms, t) \in received(ps)$ or $(ms, t) \in delivered(ps)$.

Clearly the record $Available$ is related to send and broadcast actions of other processes, as specified by the communication properties given in section 5.5. The read action is used to get an available message according to the rules that are also specified in 5.5.

A *complete run* of a process is a pair consisting of its run and an $Available$ record. Finally, the behaviour of the whole system is represented by a *complete system run*, which is a function that assigns to each process a complete run.

For a complete system run r and a process ps, let $run(r(ps))$ denote the *incomplete* run of ps in r where the $Available$ record has been omitted, and let $received(r(ps))$ and $delivered(r(ps))$ denote the corresponding fields of the $Available$ record of ps in r. Moreover, let $states(r(ps))$ and $actions(r(ps))$ denote the sequence of states and the sequence of actions in the incomplete run $run(r(ps))$, respectively.

5.4 Processes by state machines

In this section, we provide additional details on our modelling of participants and the coordinator using state machines. We only explain the main ideas and illustrate them by a few transitions in our machines. The complete specifications are not given here because they would consume too much space. We refer for further details to the PVS specifications at the end of this chapter.

5.4.1 Pseudostates

As mentioned in the previous section, the main field of each state in our machines is *name*. In Figures 5.4 and 5.5, 8 state names for each participant and 4 state names for the coordinator are shown. Note that in the PVS specification the state names for the coordinator are preceded by *C* to distinguish them from the state names for participants. Besides these names, there are also a few names for *pseudostates* of participants and the coordinator, i.e. states which a process leaves immediately without performing any activity in them. Each participant has 5 names for the following pseudostates:

- Names $AbHelp$, $ComHelp$ and $WHelp$ indicate pseudostates that are used to send a reply to a $help$ message that is received in states $Aborted$, $Committed$ or $Willing$, respectively. E.g., receiving of a $help$ request moves a participant from state $Committed$ to state $ComHelp$, and after that it immediately sends a reply to this request and moves back to state $Committed$.
- A pseudostate with name $Deciding$ is used to send a YES or NO reply to a $vote\_request$ message from the coordinator.
- A pseudostate with name $WSending$ is used to send a $help$ request to other participants immediately after timeout in state $Willing$ expires.

The names for pseudostates of the coordinator are as follows: $CInitial$ is the initial state of the coordinator which it leaves immediately by sending a $vote\_request$ message; $CAborting$ and $CCommitting$ are pseudostates used to broadcast an **abort** or **commit** message, respectively.

5.4.2 Crash and recover actions

Specifications of crash and recover actions depend on whether a state is "normal", "crashed" or a pseudostate. As Figure 5.4 shows, each participant has a transition for a crash action in each of the "normal" states $Initial$, $Waiting$, $Aborted$, $Committed$ and $Willing$. This transition has no time bounds. Each participant has a transition for a recover action in each of the "crashed" states $AbCrashed$, $ComCrashed$ and $WCrashed$. This transition has a lower time bound Δ_r (which is here equal to the length of the fictitious time-out period in each of the "final" states) to ensure that a participant does not crash and recover too often. Crash and recover actions are not specified in pseudostates because these states are left immediately. For the coordinator, a crash action is specified in "normal" states $CReceiving$, $CAborted$ and $CCommitted$, but a recover action is not specified in state $CCrashed$, because we assume that the coordinator does not recover.

5.4.3 Complete structure of states

Besides the fields $name$, $ident$, $tout$ and $wait$, that have been already explained in the previous section, each state also has a few additional fields. The coordinator uses only the first of these fields, and participants use these fields only for the termination protocol:

- $noinfo$, which for the coordinator represents the set of participants from which a vote has not yet been received, and for a participant it represents the complement of the set $willing$, i.e. the set of participants from which a reply to a $help$ message has not yet been received (during the execution of the termination protocol);
- $requests$, the natural number indicating the number of times a participant sent a help request, which is only used for the proofs;

- $dest$, used only in "help" pseudostates $ComHelp$, $AbHelp$ and $WHelp$, which represents a participant from which a $help$ message has just been received (and to which a reply must be sent immediately);
- $oldtout$, used only in "help" pseudostates, which represents the timeout of the state which the participant has just left after receiving a $help$ message (and to which it will immediately return);
- $repeats$, a natural number that used to define a lower time bound on a recover action in each of the "crashed" states of a participant. We assign 0 to $repeats$ when a participant gets into a particular crashed states for the first time, and it is incremented by 1 after each fictitious time-out action in this state. This time-out action has a period Δ_r, so by imposing a precondition $repeats > 0$ on a recover action, we ensure that it has a lower time bound Δ_r.

5.4.4 Structure of messages

As mentioned in the previous section, each state has the field $wait$ which specifies the set of messages the process is waiting for. In our model, each message is represented by a record with the following fields:

- $name$, to represent the contents of the message: participants can send messages with names yes, no, $help$, $reply\_commit$, $reply\_abort$ and $reply\_willing$, whereas the coordinator can send messages with names $vote\_request$, $commit$ and $abort$;
- $sender$, to indicate the process that sent or broadcasted the message;
- $dest$ to indicate set of processes to which the message has been sent or broadcasted;
- $ident$ is the unique identifier of the message.

5.4.5 Examples of a few transitions

To illustrate our use of state machines, we show the initial state of a participant and two transitions in its state machine. The initial state of a participant $p1$ is as follows (where $Coord$ represents the coordinator and $\{Participants\}$ represents the set of all participants):

$$
\begin{aligned}
&name = Initial \;\&\; ident = p1 \;\&\; tout = C_{know}(p1) + \Delta_c + \delta \;\&\\
&wait = \{ms \mid name(ms) = vote\_request \;\&\; sender(ms) = Coord \;\&\\
&\qquad\qquad dest(ms) = \{Participants\}\} \;\&\\
&noinfo = \{Participants\} \setminus \{p1\} \;\&\; requests = 0 \;\&\\
&dest = p1 \;\&\; oldtout = 0 \;\&\; repeats = 0
\end{aligned}
$$

The field $tout$ specifies the time moment before which a $vote\_request$ message from the coordinator must be received, as shown in Figure 5.4. The values of fields $dest$ and $oldtout$ are chosen arbitrarily here, because they will be reassigned later anyway (just before getting into a "help" pseudostate). The values of all other fields are self-explanatory. If a participant $p1$ is in a state $ps1$ such that $name(ps1) = Initial$, and it performs an action $a1$, then the precondition and effect of this action can be expressed as follows (in an imperative style close to specifications of PVS):

$$
\begin{aligned}
&(read?(act(a1)) \;\&\\
&name := Deciding \;\&\; tout := time(a1) \;\&\; wait := \emptyset) \vee
\end{aligned}
$$

$$
\begin{aligned}
&(timeout?(act(a1)) \;\&\; time(a1) = tout(ps1) \;\&\\
&name := Aborted \;\&\; tout := time(a1) + \Delta_r \;\&
\end{aligned}
$$

$wait := \{ms \mid name(ms) = help \,\&\, sender(ms) \neq Coord\}) \vee$

$(crash?(act(a1)) \,\&$
$name := AbCrashed \,\&\, tout := time(a1) + \Delta_r \,\&\, wait := \emptyset)$

Each action has two fields $time$ and act, representing its execution time and actual action. Like in chapter 4, we specify actual actions by abstract datatypes. The main motivation for this is the fact that different actions have different parameters: some include a message and some do not (see section 4.4.4 for information about abstract datatypes in PVS). Here $read?$, $timeout?$ and $crash?$ correspond to PVS *recognizers*, which can be used in tests or in subtypes.

If $a1$ gets $p1$ into state $Deciding$, we assign $tout := time(a1)$ to ensure that $Deciding$ is indeed a pseudostate, i.e. must be left immediately. If $a1$ is a time-out action, we require $time(a1) = tout(ps1)$ to ensure that it is taken exactly when the time-out of state $ps1$ expires. This time-out action has effect $tout := time(a1) + \Delta_r$ to provide a fictitious time-out action in state $Aborted$ after Δ_r time units.

The effects of read and crash actions on the set $wait$ imply that a process is not supposed to read any messages in states $Deciding$ and $AbCrashed$. The effect of timeout action implies that a process will read only $help$ messages in state $Aborted$. Note that the precondition of a read action in state $Initial$ does not require that a message being read belongs to the set $wait(ps1)$. This is because such precondition is already included into one of the communication properties, namely integrity of a read action, which will be presented in section 5.5.2.

If $p1$ is in a state $ps1$ such that $name(ps1) = Deciding$, and it performs an action $a1$, then the precondition and effect of this action are as follows:

$(act(a1) = send(ms) \,\&$
$name(ms) = yes \,\&\, sender(ms) = ident(ps1) \,\&\, dest(ms) = \{Coord\} \,\&$
$name := Waiting \,\&\, tout := C_{know}(ident(ps1)) + \Delta_c + 2 * \delta + \Delta_b \,\&$
$wait := \{ms \mid (name(ms) = commit \vee name(ms) = abort) \,\&$
$sender(ms) = Coord \,\&\, dest(ms) = \{Participants\}\}) \vee$

$(act(a1) = send(ms) \,\&$
$name(ms) = no \,\&\, sender(ms) = ident(ps1) \,\&\, dest(ms) = \{Coord\} \,\&$
$name := Aborted \,\&\, tout := time(a1) + \Delta_r \,\&$
$wait := \{ms \mid name(ms) = help \,\&\, sender(ms) \neq Coord\})$

To send a message, we must specify its fields $name$, $sender$ and $dest$. The field $ident$ is not specified, because we are not interested how identifiers are assigned to messages. We only require that each message has an unique identifier, which will be expressed by the predicate $MUnique$ in section 5.5.1.

5.5 Communication mechanism by assertions

All properties of the communication mechanism are formalized by a few predicates on complete system runs (11 in total). They are presented in the following 4 subsections. A few abbreviations are used: $Send(ps, ms, t, i, r)$ stands for "process ps sends message ms in complete system run r, and this action has index i and execution time t". Abbreviation $Crash(ps, t, i, r)$ stands for "process ps

crashes in complete system run r, and this action has index i and execution time t". Formally they are defined as follows:

$$Send(ps, ms, t, i, r) \stackrel{def}{\Longleftrightarrow} actions(r(ps))(i) = (send(ms), t)$$

$$Crash(ps, t, i, r) \stackrel{def}{\Longleftrightarrow} actions(r(ps))(i) = (crash, t)$$

The abbreviations $Broadcast(ps, ms, t, i, r)$, $Read(ps, ms, t, i, r)$ and $Recover(ps, t, i, r)$ are defined in a similar way.

5.5.1 Uniqueness of messages

By predicate $MUnique$, we formalize the already mentioned requirement that any message that has ever been sent or broadcasted has a unique identifier:

$$\begin{array}{l} MUnique(r) \stackrel{def}{\Longleftrightarrow} \forall ps1, ps2, ms1, ms2, t1, t2, i1, i2 : \\ \quad (Send(ps1, ms1, t1, i1, r) \ \vee \ Broadcast(ps1, ms1, t1, i1, r)) \ \& \\ \quad (Send(ps2, ms2, t2, i2, r) \ \vee \ Broadcast(ps2, ms2, t2, i2, r)) \ \& \\ \quad ident(ms1) = ident(ms2) \Longrightarrow \\ \qquad (ps1 = ps2 \ \& \ ms1 = ms2 \ \& \ t1 = t2 \ \& \ i1 = i2) \end{array}$$

5.5.2 Read action: validity and integrity

We explain our assumptions about reading of messages that have been received or delivered by a process. In our communication model, messages are not refused by correct processes, but not all received or delivered messages change the state of a process. A process reacts only to messages which it considers "relevant".

For instance, suppose that some participant p sends a NO vote to the coordinator and moves to the state $Aborted$. After that, the coordinator broadcasts the **abort** decision, which is eventually delivered by all participants including p. The delivery of this message is recorded in the $Available$ record of p. However, since p has already reached a decision, it will consider this **abort** message irrelevant and will take no action in response to it. Therefore there is no need to specify a read of this message in the $Aborted$ state. The separation of delivery and read actions reduces the number of transitions that have to be specified in each state, and therefore simplifies the analysis of a state machine.

If a process is in some state st, it accepts messages that belong to $wait(st)$ by performing a $read$ action. Correctness of a read action is formalized by two predicates: validity and integrity. *Validity*, formalized below by predicate $RValidity$, says that if a message ms is available at t and a process is waiting for ms, i.e. it is in a state st such that $ms \in wait(st)$, then this process should read ms at t.

$$\begin{array}{l} RValidity(r) \stackrel{def}{\Longleftrightarrow} \forall ps, ms, i, t : \\ \quad ((ms, t) \in received(r(ps)) \ \vee \ (ms, t) \in delivered(r(ps))) \ \& \\ \quad (if \ i > 0 \ then \ time(actions(run(r(ps)))(i-1)) < t \ else \ TimeS(ps) < t) \ \& \\ \quad time(actions(run(r(ps)))(i)) \geq t \ \& \\ \quad ms \in wait(states(run(r(ps)))(i)) \Longrightarrow \\ \qquad Read(ps, ms, t, i, r) \end{array}$$

In this formal definition, two timing conditions are added to determine the exact index i of a read action. The action with index i is performed in the state with index i, so the first timing condition expresses that we "arrive" to this state before time t. If $i > 0$, this arrival corresponds to performing the action with index $i-1$, and if $i = 0$ (i.e. the state is initial), it happens at time $TimeS(ps)$, where $TimeS$ is a staring time of a process as defined in section 5.3. The second timing condition says that a process does not leave the state with index i before time t (which corresponds to performing the action with index i).

To ensure that this definition of validity is non-contradictory, we should allow at most one message with a particular timestamp to be available. This restriction is introduced below by predicate $MAvail$. It must be ensured by the communication medium and not the process itself. Considering that in our model time is represented by a set of real numbers, such ordering of messages should not be too difficult to achieve.

$$
\begin{aligned}
&MAvail(r) \overset{def}{\Longleftrightarrow} \forall ps, ms1, ms2, t : \\
&\quad ((ms1, t) \in received(r(ps1)) \ \vee \ (ms1, t) \in delivered(r(ps1))) \ \& \\
&\quad ((ms2, t) \in received(r(ps1)) \ \vee \ (ms2, t) \in delivered(r(ps1))) \Longrightarrow \\
&\qquad ms1 = ms2
\end{aligned}
$$

Integrity of a read action, formalized below by predicate $RIntegrity$, expresses that we can read ms at t only if ms is available at t. This implies that a received or delivered message is lost forever if it has not been read immediately.

$$
\begin{aligned}
&RIntegrity(r) \overset{def}{\Longleftrightarrow} \forall ps, ms, i, t : \\
&\quad Read(ps, ms, t, i, r) \Longrightarrow \\
&\qquad ((ms, t) \in received(r(ps)) \ \vee \ (ms, t) \in delivered(r(ps))) \ \& \\
&\qquad ms \in wait(states(r(ps))(i))
\end{aligned}
$$

5.5.3 Specification of send/receive

Since a crashed process does not receive or deliver any messages, we must distinguish crashed and operational processes in order to define the communication properties correctly. Informally, we say that a process is crashed between each pair of crash and recover actions and correct otherwise. The formal definition of a correct process is somewhat more complicated and is given below by predicate $Correct$. This predicate says that a process ps is correct in the closed interval $[t1, t2]$ if and only if is does not crash within this interval, and each crash action that happens before $t1$ is followed by a recover action that also happens before $t1$.

$$
\begin{aligned}
&Correct(ps, t1, t2, r) \overset{def}{\Longleftrightarrow} t2 > t1 \ \& \\
&\quad \forall t, i : Crash(ps, t, i, r) \Longrightarrow \\
&\qquad (t < t1 \ \vee \ t > t2) \ \& \\
&\qquad (t < t1 \Longrightarrow \exists t3, j : t3 < t1 \ \& \ j > i \ \& \ Recover(ps, t3, j, r))
\end{aligned}
$$

In this definition of $Correct$, the relative order of crash and recover actions is expressed by comparing their indices i and j and not execution times t and $t3$, which are allowed to be equal.

Below we formalize the validity, integrity and timeliness properties of send/receive that were spec-

ified in section 5.1. We found it natural to combine the validity and timeliness properties into a single property, represented by predicate $SValidity$ below. In general, when message ms is sent to the set of processes $dest(ms)$, we don't assume indivisibility of this action. We model the sending of ms only to a subset of $dest(ms)$ (due to a crash) as a send action immediately followed by a crash action with the same execution time. Therefore, predicate $SValidity$ requires ms to be received by *each* member of $dest(ms)$ only if a send action is *not* immediately followed by a crash action with the same execution time.

$$SValidity(r) \stackrel{def}{\Longleftrightarrow} \forall ps1, ps2, ms, t, i :$$
$$Send(ps1, ms, t1, i, r) \ \& \ \neg Crash(ps1, t1, i+1, r) \ \&$$
$$ps2 \in dest(ms) \ \& \ Correct(ps2, t1, t1+\delta, r) \Longrightarrow$$
$$\exists t2 : t2 > t1 \ \& \ t2 < t1+\delta \ \& \ (ms, t2) \in received(r(ps2))$$

At the same time, the integrity property have been naturally split into two properties: predicate $SIntegrity1$ says that every received message ms has been sent by its sender $sender(ms)$ some time ago (recall from section 5.4 that $sender(ms)$ is just a *field* of message m), and predicate $SIntegrity2$ says that every message can be received only once.

$$SIntegrity1(r) \stackrel{def}{\Longleftrightarrow} \forall ps1, ms, t1 :$$
$$(ms, t1) \in received(r(ps1)) \Longrightarrow$$
$$\exists i, t2 : t2 < t1 \ \& \ Send(sender(ms), ms, t2, i, r)$$

$$SIntegrity2(r) \stackrel{def}{\Longleftrightarrow} \forall ps1, ms, t1, t2 :$$
$$(ms, t1) \in received(r(ps1)) \ \& \ (ms, t2) \in received(r(ps1)) \Longrightarrow$$
$$t1 = t2$$

5.5.4 Specification of reliable broadcast

Using the abbreviation $Correct$ from the previous subsection, we formalize the validity, integrity, timeliness and uniform agreement properties of reliable broadcast that were specified in section 5.1. The predicates for the validity, integrity and timeliness are very similar to the predicates for the corresponding properties of send/receive from the previous subsection:

$$BValidity(r) \stackrel{def}{\Longleftrightarrow} \forall ps1, ps2, i, ms, t1 :$$
$$Broadcast(ps1, ms, t1, i, r) \ \& \ \neg Crash(ps1, t1, i+1, r) \ \&$$
$$ps2 \in dest(ms) \ \& \ Correct(ps2, t1, t1+\Delta_B, r) \Longrightarrow$$
$$\exists t2 : t2 > t1 \ \& \ t2 < t1+\Delta_B \ \& \ (ms, t2) \in delivered(r(ps2))$$

$$BIntegrity1(r) \stackrel{def}{\Longleftrightarrow} \forall ps1, ms, t1 :$$
$$(ms, t1) \in delivered(r(ps1)) \Longrightarrow$$
$$\exists i, t2 : t2 < t1 \ \& \ Broadcast(sender(ms), ms, t2, i, r)$$

$$BIntegrity2(r) \stackrel{def}{\Longleftrightarrow} \forall ps1, ms, t1, t2 :$$
$$(ms, t1) \in delivered(r(ps1)) \ \& \ (ms, t2) \in delivered(r(ps1)) \Longrightarrow$$
$$t1 = t2$$

In the formal definition of uniform agreement, the broadcasting process has been introduced to

make the timing properties more precise. Note that in the predicate $BUnifAgreement$ below we no longer require that a broadcast action cannot immediately be followed by a crash action with the same execution time. Indeed, even if a crash happens in the middle of a broadcast, the delivery of a message by some process $p3$ guarantees that all correct processes will also deliver this message.

$$\begin{aligned}
&BUnifAgreement(r) \stackrel{def}{\Longleftrightarrow} \forall ps1, ps2, i, ms, t1 : \\
&\quad Broadcast(ps1, ms, t1, i, r)\ \&\\
&\quad (\exists ps3, t3 : (ms, t3) \in delivered(r(ps3)))\ \&\\
&\quad ps2 \in dest(ms)\ \&\ Correct(ps2, t1, t1 + \Delta_B, r) \Longrightarrow\\
&\qquad \exists t2 : t2 > t1\ \&\ t2 < t1 + \Delta_B\ \&\ (ms, t2) \in delivered(r(ps2))
\end{aligned}$$

5.6 Specification of correctness properties

In this section, we formalize the correctness properties of our protocol that have been already informally defined in section 2.4.1. Like the properties of communication mechanism, they are also expressed as predicates on complete system runs. The formalization uses both states and actions of complete system runs and is fairly straightforward.

The following definitions are introduced: we say that a participant *p commits* in a complete system run, if it eventually reaches one of the *committed states* in this run; *p aborts*, if it eventually reaches one of the *aborted states*. For this particular protocol, the set of committed states consists of all states with one of the names $Committed$, $ComCrashed$ or $ComHelp$; the set of aborted states consists of all states with one of the names $Aborted$, $AbCrashed$ or $AbHelp$. We say that *p decides*, if it either commits or aborts.

$$\begin{aligned}
&ComState(st) \stackrel{def}{\Longleftrightarrow}\\
&\quad name(st) = Committed\ \vee\ name(st) = ComCrashed\ \vee\ name(st) = ComHelp
\end{aligned}$$

$$Commits(p1, r) \stackrel{def}{\Longleftrightarrow} \exists i : ComState(states(r(p1))(i))$$

$$AbState(st) \stackrel{def}{\Longleftrightarrow} name(st) = Aborted\ \vee\ name(st) = AbCrashed\ \vee\ name(st) = AbHelp$$

$$Aborts(p1, r) \stackrel{def}{\Longleftrightarrow} \exists i : AbState(states(r(p1))(i))$$

$$Decides(p1, r) \stackrel{def}{\Longleftrightarrow} Commits(p1, r)\ \vee\ Aborts(p1, r)$$

5.6.1 Properties AC1-AC5

Using these abbreviations, AC1 is represented as

$$\begin{aligned}
&AC1(r) \stackrel{def}{\Longleftrightarrow} \forall p1, p2 :\\
&\quad Decides(p1, r)\ \&\ Decides(p2, r) \Longrightarrow\\
&\qquad (Commits(p1, r) \Longleftrightarrow Commits(p2, r))
\end{aligned}$$

Using abbreviation $Send$ from the previous chapter and a new abbreviation $EachYes$, expressing

that each participant sends its YES vote to the coordinator, the property AC2 is formulated.

$$EachYes(r) \stackrel{def}{\Longleftrightarrow} \forall p1 : \exists i : \\ act(actions(r(p1))(i)) = send(ms)\ \&\ \\ name(ms) = yes\ \&\ sender(ms) = p1\ \&\ dest(ms) = \{Coord\}$$

$$AC2(r) \stackrel{def}{\Longleftrightarrow} (\exists p1 : Commits(p1, r)) \Longrightarrow EachYes(r)$$

The definition of AC3 uses abbreviation $NoCrashes$, expressing that each participant in a complete system run is *always correct*, i.e. never performs a crash action.

$$AlwaysCorrect(p1, r) \stackrel{def}{\Longleftrightarrow} \forall t, i : \neg Crash(p1, t, i, r)$$

$$NoCrashes(r) \stackrel{def}{\Longleftrightarrow} \forall p1 : AlwaysCorrect(p1, r)$$

$$AC3(r) \stackrel{def}{\Longleftrightarrow} EachYes(r)\ \&\ NoCrashes(r) \Longrightarrow \forall p1 : Commits(p1, r)$$

In our model, the decision to commit or abort corresponds to getting to a committed or an aborted state, respectively, so the property AC4 is represented as follows:

$$AC4(r) \stackrel{def}{\Longleftrightarrow} \forall p1, i, j : \\ (ComState(states(r(p1))(i))\ \&\ j > i \Longrightarrow ComState(states(r(p1))(j)))\ \&\ \\ (AbState(states(r(p1))(i))\ \&\ j > i \Longrightarrow AbState(states(r(p1))(j)))$$

The definition of AC5 uses abbreviations $Decides$ and $AlwaysCorrect$ that have been already introduced above.

$$AC5(r) \stackrel{def}{\Longleftrightarrow} \forall p1 : AlwaysCorrect(p1, r) \Longrightarrow Decides(p1, r)$$

5.6.2 Property AC6

The formalization of the *termination property* AC6 is by far the most interesting and complicated. In order to reach a decision, the original formulation requires *all* participants to stay operational simultaneously during some sufficiently long period of time. Our property, however, requires only that *each pair* of participants is operational simultaneously for sufficiently long time. This implies termination even for executions in which at most two participants are operational at any moment of time during the execution of the termination protocol. To define this more formally, we introduce the notion of *strongly non-blocking* protocols. We say that an ACP with recovery is strongly non-blocking if it has *uncertainty bound* and *communication delay*. The intuitive meaning of these additional notions is as follows: uncertainty bound B is the time by which each correct participant must decide, and communication delay D is the length of the period of time which is sufficient for a correct participant to learn the state of any other correct participant and not loose this information in the future as a result of crashes. The formal definition of the property can be given as follows: ACP with recovery is strongly non-blocking if there exist real numbers B and D such that the following property holds:

AC6S: If a participant p recovers, and after that for each other participant q

there is a time interval $[t_q,\ t_q + D]$, where $t_q \geq B$, such that both p and q stay correct during this interval, then p eventually decides.

We have proved that our protocol satisfies AC6S with $B = Ct_{start} + \Delta_c + 2 * \delta + \Delta_b$ and $D = 4 * \delta$. It can be determined from Figure 5.1 that each correct participant p decides by the time $C_{know}(p) + \Delta_c + 2 * \delta + \Delta_b$, and we mentioned in section 5.2.2 that for any p, we have $C_{know}(p) < Ct_{start}$. Therefore, each correct participant indeed must decide by the time B. Communication delay D is the time needed to send a $help$ message to another participant and get a reply.

If we eliminate the restriction $t_q \geq B$ from the definition of $AC6S$, then the communication delay will greatly increase. Indeed, if a participant is still waiting for the coordinator's decision, it will not reply to requests for help from recovered participants. Therefore if some participant recovers "too early", it may have to wait longer to get replies to its $help$ messages.

The fact that $D = 4 * \delta$ for our protocol may seem strange considering that it takes at most $2 * \delta$ to get a reply to a message. However, it is easy to show that communication delay must be greater than $2 * \delta$ because the periods of correctness are not synchronized. Indeed, suppose that a participant $p1$ recovers at time $t1$ and a participant $p2$ recovers at time $t1 + \epsilon$. Participant $p1$ immediately sends a $help$ message to $p2$. Even if ϵ is very small, this message may still arrive before the recovery of $p2$ and will be lost. Participant $p1$ sends a new $help$ message to $p2$ at time $t1 + 2 * \delta$, so even if $p2$ is correct during the interval $[t1 + \epsilon, t1 + 4 * \delta]$, then $p1$ is guaranteed to get a reply from $p2$ only by the time $t1 + 4 * \delta$. Therefore, if $p1$ crashes again shortly before $t1 + 4 * \delta$, then it may not receive the reply from $p2$, even though $p1$ and $p2$ simultaneously stayed correct during almost $4 * \delta$ time units.

The mathematical representation of the property AC6S is relatively complicated, but still rather similar to its less formal definition (here tf is a function that assigns to each participant except $p1$ the time when its correctness period begins):

$$
\begin{array}{l}
AC6S(r) \stackrel{def}{\Longleftrightarrow} \\
\quad \exists B, D: \\
\quad\quad \forall p1: \\
\quad\quad\quad (\exists i, t, tf: \\
\quad\quad\quad\quad Recover(p1, i, t, r)\ \& \\
\quad\quad\quad\quad (\forall p2: p2 \neq p1 \Longrightarrow \\
\quad\quad\quad\quad\quad tf(p2) \geq t\ \& \\
\quad\quad\quad\quad\quad tf(p2) \geq B\ \& \\
\quad\quad\quad\quad\quad Correct(p2, tf(p2), tf(p2) + D, r)\ \& \\
\quad\quad\quad\quad\quad Correct(p1, tf(p2), tf(p2) + D, r))) \\
\quad\quad\quad \Longrightarrow Decides(p1, r)
\end{array}
$$

This improved property is achieved by our termination protocol because of our decision to keep the set $willing$ in stable storage. It is easy to see that if this set is in volatile storage, then there is no B and D such that the protocol is strongly non-blocking with respect to B and D. Indeed, suppose that all participants vote YES and then crash while waiting for the coordinator's decision. Next assume that participants p and q recover and stay operational for a very long time. Suppose p learns that q is also willing to commit and then they both crash again. If the set $willing$ is in volatile storage, then p would lose the information that q is also willing to commit. If q never recovers, then p will never learn the state of q again and will never decide, even if all participants except q recover and stay operational forever.

It is easy to determine from Figure 5.4, that a decision to commit based on the set $willing$ can only happen in an execution, in which a *total failure* has occurred, i.e. no participant always remained correct. The property AC6S improves the chances of termination for such executions by keeping the set $willing$ in stable instead of volatile storage. Since total failures in most systems are rare, one may ask, whether this additional complexity is justified for real-world systems. In our opinion, it indeed may be justified for real-time fault-tolerant systems, in which the time needed to complete a transaction is a crucial resource, whereas stable storage is fast and cheap. However, such a complicated question may only be resolved by simulation and experiments. An interesting example of such experimental evaluation is [GHR97], where it is shown that even some version of Three-Phase Commit protocol (see section 2.4.2), despite all complexity and overheads of this protocol, performs better then 2PC for some realistic environments.

5.7 Verification of correctness properties

Assuming the communication properties specified in section 5.5, we must prove correctness properties defined in section 5.6 for each complete system run. To prove that the protocol satisfies a correctness property, we combine operational reasoning about state machines and reasoning in terms of assertions. It turned out that it is rather easy to use assertions in the proofs. However, the analysis of our state machines is rather tedious, because states are parameterized and their number is relatively large (for instance, for a participant we have 13 state names and about 30 transitions between states depending on state names). Although this analysis has been partly automated, we still had to interactively prove a large number of PVS lemmas (over 250). We prefer to spare the reader from boring details, so we show only the main ideas of each proof. The proofs are given in the order of increasing difficulty.

5.7.1 Verification of AC4

The proof of AC4 is fairly easy. By considering all transitions from committed states, we prove by induction that after a participant gets into one of the committed states, it always stays in one of the committed states. The same fact is proved for aborted states.

5.7.2 Verification of AC5

The proof of AC5 is only slightly more complicated. The initial state (with index 0) of any participant $p1$ has name $Initial$. By considering transitions that are possible in state $Initial$, we prove that the next state (with index 1) has name $Aborted$, $ACrashed$ or $Deciding$. By definition of $Decides$, in the first two cases we are done. If $name(states(r(p1))(1)) = Deciding$, we prove that $name(states(r(p1))(2)) = Aborted$ or $name(states(r(p1))(2)) = Waiting$. In the first case we are finished again. If $name(states(r(p1))(2)) = Waiting$, we prove that either the next state is $Committed$ or $Aborted$, or $p1$ performs a crash action. By definition of $AlwaysCorrect$, the last case is impossible, and this completes the proof.

5.7.3 Verification of AC3

Suppose all participants send vote YES to the coordinator and no failures occur. The state machine of the coordinator implies that it sends a $vote\_request$ message to each participant at real time Ct_{start}. The validity assertion for the $send$ action implies that each participant receives this message within interval $(Ct_{start}, Ct_{start} + \delta)$. Each participant sends its YES vote immediately after

reading a $vote\_request$ message, so each YES vote is sent within $(Ct_{start}, Ct_{start} + \delta)$. After sending $vote\_request$, the coordinator moves into state $CReceiving$. The validity of $send$ action implies that the coordinator receives all YES votes within $(Ct_{start}, Ct_{start} + 2 * \delta)$, i.e. before the time-out in state $Receiving$ is triggered.

We also prove that the coordinator cannot leave state $Receiving$ by reading a NO vote from some participant. This follows from the fact that no participant sends a NO vote (which is deduced from the participants state machine) and the integrity of read actions. Since the coordinator receives YES votes from all participants before the time-out expires, and it is waiting for these votes, the validity of read actions implies that the coordinator reads all votes. Reading of a YES vote from a participant removes this participant from the set $noinfo$. We prove by induction on the number of votes that have been read, that $noinfo$ at any moment includes exactly those participants whose votes have not yet been read. So just before the last vote from some participant $p1$ is read, $noinfo$ includes exactly one participant $p1$. The state machine now implies that the reading of this last vote moves the coordinator into state $Committing$. In this state, it has no choice but to broadcasts **commit**.

After voting YES, each participant moves into state $Waiting$. We prove that it cannot read an **abort** decision in this state, since then the integrity assertions for read and broadcast actions would imply that the coordinator broadcasted **abort**. This is impossible, because it broadcasted **commit**. It also cannot crash, because no failures occur. Validity of broadcast implies that a participant delivers the **commit** decision in state $Waiting$ by time $Ct_{start} + \Delta_b + 2 * \delta$, i.e. before the time-out expires. Finally, validity of the read action implies that it reads this decision and commits.

5.7.4 Verification of AC2

Suppose some participant $p1$ commits in r. In the proof of $EachYes(r)$, we need to consider 3 subcases: $p1$ commits by reading **commit** message from the coordinator; $p1$ commits by reading $reply\_willing$ messages from all other participants; $p1$ commits by reading $reply\_commit$ message from some other participant. We formalized these subcases by predicates $Commits1(p1, r)$, $Commits2(p1, r)$ and $Commits3(p1, r)$, respectively.

$$Commits1(p1, r) =$$
$$\exists i : name(states(r(p1))(i)) = Waiting \ \& \ name(states(r(p1))(i+1)) = Committed$$

$$Commits2(p1, r) =$$
$$\exists i : name(states(r(p1))(i)) = Willing \ \&$$
$$act(actions(r(p1))(i)) = read(ms) \ \& \ name(ms) = reply\_willing \ \&$$
$$name(states(r(p1))(i+1)) = Committed$$

$$Commits3(p1, r) =$$
$$\exists i : name(states(r(p1))(i)) = Willing \ \&$$
$$act(actions(r(p1))(i)) = read(ms) \ \& \ name(ms) = reply\_commit \ \&$$
$$name(states(r(p1))(i+1)) = Committed$$

It is easy to prove that they cover all possible ways to commit:

$$\forall p1, r : Commits(p1, r) \iff (Commits1(p1, r) \ \vee \ Commits2(p1, r) \ \vee \ Commits3(p1, r))$$

Let's consider each of these subcases.

1) $Commits1(p1, r)$. If $p1$ commits by reading **commit** message, integrity of read and broadcast actions imply that the coordinator must have broadcasted **commit** in state $Committing$. Reusing some lemmas from the proof of AC3, we prove that the coordinator could get into state $Committing$ only by reading YES votes from all participants. Integrity of read and send actions implies that each participant must have sent a YES vote to the coordinator, and this completes the proof.

2) $Commits2(p1, r)$. If $p1$ commits by reading $reply\_willing$ messages from all other participants, integrity of read and send actions imply that all other participants sent $reply\_willing$ messages to $p1$. They could only do it in state $WHelp$, and by a (very tedious) analysis of the state machine we can prove that a participant can get from state $Initial$ to state $WHelp$ only passing through a transition from state $Deciding$ to state $Waiting$. This is exactly the transition which sends a YES vote to the coordinator.

3) $Commits3(p1, r)$. Let $Min(r)$ denote the minimal execution time of all actions of all participants in r, in which a participant moves from state $Willing$ to state $Committed$ by reading a $reply\_commit$ message. The definition of $Commits3(p1, r)$ easily implies that $Min(r)$ exists. Let $p2$ denote a participant corresponding to $Min(r)$, i.e. a participant that reads a $reply\_commit$ message at time $Min(r)$. Integrity of read and send actions implies that some other participant $p3$ sends $reply\_commit$ to $p2$ at some time t such that $t < Min(r)$. It can send such reply only from state $ComHelp$, so $p3$ has committed. We can have one of three cases $Commits1(p3, r)$, $Commits2(p3, r)$ or $Commits3(p3, r)$. If we assume $Commits3(p3, r)$, we can easily obtain a contradiction with the definition of $Min(r)$. But we already proved above that both $Commits1(p3, r)$ and $Commits2(p3, r)$ imply $EachYes(r)$. This completes the proof of case 3 and property AC2.

5.7.5 Verification of AC1

If some participant $p1$ aborts in r, there can be two subcases: $p1$ moves into state $Aborted$ or $AbCrashed$ from one of the states $Initial$, $Deciding$ or $Waiting$, or $p1$ moves into state $Aborted$ from state $Willing$. We formalized these subcases by predicates $Aborts1(p1, r)$ and $Aborts2(p1, r)$, respectively, and is easy to prove that they cover all possible ways to abort.

$$
\begin{aligned}
&Aborts1(p1, r) = \\
&\quad \exists i : ((name(states(r(p1))(i)) = Initial \ \vee \ name(states(r(p1))(i)) = Deciding \vee \\
&\quad name(states(r(p1))(i)) = Waiting) \ \& \ name(states(r(p1))(i+1)) = Aborted) \vee \\
&\qquad (name(states(r(p1))(i)) = Initial \ \& \ name(states(r(p1))(i+1)) = AbCrashed)
\end{aligned}
$$

$$
\begin{aligned}
&Aborts2(p1, r) = \\
&\quad \exists i : name(states(r(p1))(i)) = Willing \ \& \ name(states(r(p1))(i+1)) = Aborted
\end{aligned}
$$

$$
\forall p1, r : Aborts(p1, r) \iff (Aborts1(p1, r) \ \vee \ Aborts2(p1, r))
$$

Suppose we have both $Commits(p1, r)$ and $Aborts(p2, r)$. To prove AC1, we need to obtain a contradiction. By applying the already proved property AC2, we obtain $EachYes(r)$. Moreover, without any loss of generality we can assume $Aborts1(p2, r)$ and either $Commits1(p1, r)$ or $Commits2(p1, r)$. Indeed, we already proved in the previous subsection that $Commits3(p1, r)$ implies that there exists some other participant $p1'$ such that either $Commits1(p1', r)$ or

$Commits2(p1', r)$. Using the same technique, we can prove that $Aborts2(p2, r)$ implies that there exists some other participant $p2'$ such that $Aborts1(p2', r)$. Therefore we can only consider the following two subcases.

1) $Commits1(p1, r)$ and $Aborts1(p2, r)$. We already know that $p2$ must have voted YES, thus moving into state $Waiting$. As in the case 1 of the previous subsection, we prove that the coordinator broadcasted **commit**. This message was delivered by at least one participant, namely $p1$. By applying the *uniform agreement* property of broadcast, we can prove that $p2$ delivers the **commit** message before the time-out in state $Waiting$ expires. Participant $p2$ cannot crash in state $Waiting$, because it would contradict $Aborts1(p2, r)$. It also cannot deliver **abort**, because the coordinator could not broadcast **abort**. Therefore the validity of a read action implies that in state $Waiting$, $p2$ has no choice but to read the **commit** message and commit. This obviously contradicts $Aborts1(p2, r)$.

2) $Commits2(p1, r)$ and $Aborts1(p2, r)$. Applying the same reasoning as in case 2 of the previous subsection, we prove that $p2$ sent a $reply\_willing$ message to $p1$ from state $WHelp$. We can now easily obtain a contradiction with $Aborts1(p2, r)$. Indeed, by analyzing our state machine we can prove that a participant cannot get into one of the states $Initial$, $Deciding$ or $Waiting$ from state $WHelp$, so it surely cannot abort in one of these states. This completes the proof of case 2 and property AC1.

5.7.6 Verification of AC6S

We give a sketch of the proof of our improved property AC6S. As already mentioned, our protocol satisfies AC6S with $B = Ct_{start} + \Delta_c + 2 * \delta + \Delta_b$ and $D = 4 * \delta$. To prove this, we assume that a participant $p1$ recovers at time t and all preconditions of AC6S are satisfied for $p1$, i.e. for any other participant $p2$ there exists time $tf(p2)$ such that $tf(p2) \geq t$, $tf(p2) \geq Ct_{start} + \Delta_c + 2 * \delta + \Delta_b$ and both $p1$ and $p2$ stay correct within interval $[tf(p2), tf(p2) + 4 * \delta]$.

We must prove that under these assumptions $p1$ will decide. If $p1$ recovers from states $ComCrashed$ or $AbCrashed$, it has already decided. It remains to consider the case when $p1$ recovers from state $WCrashed$ to state $Willing$. In this case, $p1$ immediately starts sending $help$ requests to other participants with a period $2 * \delta$. Therefore, for any other participant $p2$, at least one $help$ request will be sent to $p2$ within interval $[tf(p2), tf(p2) + 2 * \delta]$. Validity of a send action implies that $p2$ will receive this request within interval $[tf(p2), tf(p2) + 3 * \delta]$. By time $tf(p2)$, participant $p2$ is no longer in states $Initial$ or $Waiting$, so it will be willing to send to $p1$ one of the possible replies $reply\_commit$, $reply\_abort$ or $reply\_willing$. Participant $p1$ will receive this reply within interval $[tf(p2), tf(p2) + 4 * \delta]$. By this time, it is either already decided or still in state $Willing$. If $p1$ is still in state $Willing$, this means that $p1$ is still waiting for a reply from $p2$. Therefore, $p1$ will read this reply and remove $p2$ from the set $noinfo$.

To complete the proof, we consider the maximum M of the elements of the form $tf(p) + 4 * \delta$, where p is any participant different from $p1$. We prove that by time M, $p1$ either received $reply\_commit$ or $reply\_abort$ from some participant, or all other participants have been removed from the set $noinfo$. In both cases, $p1$ decides by time M.

5.7.7 Example of lemma

When we tried to implement the proof of AC6S given above in PVS, we faced some difficulties with details of the proof (not PVS specific), and it took some weeks to overcome them. It would be too

tedious to present here all details of our verification. However, to improve the reader's understanding of our methods, we give an example of an important lemma in the proof of AC6S. This lemma states sufficient conditions under which a participant will receive a reply to its help request, and is formulated as follows:

$$\begin{array}{l}\forall p1, p2, t, i, r : actions(r(p1))(i) = (send(ms), t) \;\&\; t \geq Ct_{start} + \Delta_c + 2 * \delta + \Delta_b \;\& \\ name(ms) = help \;\&\; sender(ms) = p1 \;\&\; p2 \in dest(ms) \;\& \\ Correct(p1, t, t + 2 * \delta, r) \;\&\; Correct(p2, t, t + \delta, r) \;\& \\ SValidity(r) \;\&\; RValidity(r) \Longrightarrow \\ \quad \exists t2, ms2 : t < t2 \;\&\; t2 < t + 2 * \delta \;\&\; (ms2, t2) \in received(r(p1)) \;\& \\ \quad (name(ms2) = reply\_commit \;\vee\; name(ms2) = reply\_abort \;\vee \\ \qquad name(ms2) = reply\_willing) \;\& \\ \quad sender(ms2) = p2 \;\&\; dest(ms2) = \{p1\} \end{array} \qquad \text{ReplyLem}$$

The proof of lemma $ReplyLem$ proceeds as follows. Assumption $SValidity$ implies that that $p2$ will receive ms, i.e. there exists $t1$ such that $t < t1$, $t1 < t + \delta$ and $(ms, t1) \in received(r(p2))$. But how can we prove that $p2$ will actually read ms? Assumption $SValidity$ requires that we not only prove that $p2$ will be willing to read help request ms, but also give the exact index of the read action. To determine this index, we first compute the maximum of indices of actions of $p2$ whose executions time is less than $t1$:

$$maxind(p2, t1, r) = maximum(\{i \mid time(actions(r(p2))(i)) < t1\})$$

By applying the assumption about non-Zeno behaviour from section 5.3 and the fact that executions times are non-decreasing, we can easily prove that $maxind(p2, t1, r)$ always exists. By definition of $maxind(p2, t1, r)$, state with index $maxind(p2, t1, r)+1$ is the last state that $p2$ enters before $t1$. So if we assume that in state with index $maxind(p2, t1, r) + 1$ participant $p2$ is waiting for ms and apply $SValidity$ for $p2$, ms, $maxind(p2, t1, r)+1$ and $t1$, we obtain that $p2$ will indeed read ms. To prove that in state with index $maxind(p2, t1, r) + 1$ participant $p2$ is waiting for help request ms, we use the following lemma, the proof of which is discussed later:

$$\begin{array}{l}\forall p2, t3, t4, t1, r : t3 < t4 \;\&\; Correct(p2, t3, t4, r) \;\& \\ t3 < t1 \;\&\; t1 < t4 \;\&\; t1 \geq Ct_{start} + \Delta_c + 2 * \delta + \Delta_b \Longrightarrow \\ \quad name(states(r(p2))(maxind(p2, t1, r) + 1)) = Committed \;\vee \\ \quad name(states(r(p2))(maxind(p2, t1, r) + 1)) = Aborted \;\vee \\ \quad name(states(r(p2))(maxind(p2, t1, r) + 1)) = Willing \end{array} \qquad \text{IndLem}$$

By applying lemma $IndLem$ with corresponding parameters, we obtain that state with index $maxind(p2, t1, r)+1$ has one of the names $Committed$, $Aborted$ or $Willing$. The definition of our state machine implies that ms belongs to the set $wait(states(r(p2))(maxind(p2, t1, r) + 1))$, and this completes the proof that $p2$ will read ms.

The proof of lemma $ReplyLem$ can now be finished relatively easily. The action of $p2$ with index $maxind(p2, t1, r) + 1$ will read ms and move $p2$ into one of the states $ComHelp$, $AbHelp$ or $WHelp$, depending on the name of the state where it is performed. The specification of these 3 states implies that in each of them $p2$ will send a corresponding reply to $p1$ by performing a send action with index $maxind(p2, t1, r) + 2$. This action will have the same execution time $t1$ such that $t < t1$ and $t1 < t + \delta$. By applying $SValidity$ for $p2$, we obtain that $p1$ will indeed obtain one of the 3

possible replies from $p2$ within interval $[t,\ t + 2 * \delta]$.

Proof of lemma IndLem. This lemma is proved by the "brute force" method: we eliminate all 10 state names of a participant that are different from names $Committed$, $Aborted$ and $Willing$. For example, to show that if time $t1$ belongs to a period of correctness, then the last state $p2$ enters before $t1$ cannot be state $ComCrashed$, we prove the following lemma:

$$\forall p2, t3, t4, t1, r : t3 < t4 \ \& \ Correct(p2, t3, t4, r) \ \& \ t3 < t1 \ \& \ t1 < t4 \Longrightarrow$$
$$name(states(r(p2))(maxind(p2, t1, r) + 1)) \neq ComCrashed$$

These 10 state names form 3 groups, and the proof of the corresponding lemma for states from the same group is almost the same:

- "Crashed" states $ComCrashed$, $AbCrashed$ and $WCrashed$. For these states, we perform some manipulations with the definition of the predicate $Correct$ and our state machine, and finally show that if each crash action is followed by a recover action, then the resulting state cannot be one of the crashed states.
- "Early" states $Initial$ and $Waiting$. Using the fact that both of these states have timeouts that are less than $Ct_{start} + \Delta_c + 2 * \delta + \Delta_b$, we prove that both of them will be left before that time.
- "Pseudostates" $Deciding$, $WSending$, $AbHelp$, $ComHelp$ and $WHelp$. The proof is based on the fact that all of these states are left immediately, i.e. by performing an action with the same execution time as the previous action. Therefore, an action leading to one of these states cannot possibly have the maximal index.

5.8 PVS specifications

5.8.1 Time, actions, preruns

```
Time : THEORY
BEGIN

Time : TYPE = real

tstart : Time

t : VAR Time

delta, DeltaB, DeltaC, DeltaR : { t | t > 0 }

Ctstart : { t | tstart < t & t <= tstart + DeltaC }

END Time



ActionNames [Messages : TYPE] : DATATYPE
BEGIN

send(msend : Messages) : send?
broadcast(mbroad : Messages) : broadcast?
```

```
read(mread : Messages) : read?
crash : crash?
recover : recover?
timeout : timeout?

END ActionNames



PreRuns [Processes, StateNames, MessageNames, Rest : TYPE ] : THEORY
BEGIN

IMPORTING Time

Ident : TYPE

Messages : TYPE = [#  name   : MessageNames,
                      sender : Processes,
                      dest   : setof[Processes],
                      ident  : Ident #]

IMPORTING ActionNames [Messages]

Actions : TYPE = [# act : ActionNames, time : Time #]

States : TYPE = [# name  : StateNames,
                   ident : Processes,
                   tout  : Time,
                   wait  : pred[Messages],
                   rest  : Rest #]

PreRuns0 : TYPE = [# states  : sequence[States],
                     actions : sequence[Actions] #]

i, j, k : VAR nat
pre0 : VAR PreRuns0
L : VAR Time

NonZeno(pre0) : bool =
   FORALL L : EXISTS i : time(actions(pre0)(i)) > L

PreRuns1 : TYPE = { pre0 : PreRuns0 |
   (FORALL i : time(actions(pre0)(i + 1)) >= time(actions(pre0)(i))) &
   (FORALL i : time(actions(pre0)(i)) <= tout(states(pre0)(i))) &
   NonZeno(pre0) }

Avail : TYPE = [# received : [[Messages, Time] -> bool],
                  delivered : [[Messages, Time] -> bool] #]

ComplPreRuns : TYPE = [# run : PreRuns1, avail  : Avail #]

ComplGloPreRuns : TYPE = [Processes -> ComplPreRuns]
```

```
pre : VAR ComplGloPreRuns
ps : VAR Processes
t, t1, t2, t3 : VAR Time

Correct(ps, t1, (t2 | t2 > t1), pre) : bool = FORALL t, i:
   actions(run(pre(ps)))(i) = (# act := crash, time := t #) =>
      ((t < t1 OR t > t2) &
       (t < t1 =>
           EXISTS t3, j: t3 < t1 & j > i &
                     actions(run(pre(ps)))(j) =
                        (# act := recover, time := t3 #)))

AlwaysCorrect(ps, pre) : bool =
   FORALL i : NOT crash?(act(actions(run(pre(ps)))(i)))

END PreRuns
```

5.8.2 Communication properties

```
Communication [Processes, StateNames, MessageNames, Rest : TYPE,
     (IMPORTING PreRuns [Processes, StateNames, MessageNames, Rest])
     TimeS : [Processes -> Time] ] : THEORY
BEGIN

ps, ps1, ps2, ps3 : VAR Processes
i, j, k : VAR nat
ms, ms1, ms2 : VAR Messages
t, t1, t2, t3 : VAR Time
pre : VAR ComplGloPreRuns

MUnique(pre) : bool = FORALL ps1, ps2, i, j :
   ((send?(act(actions(run(pre(ps1)))(i))) OR
     broadcast?(act(actions(run(pre(ps1)))(i)))) &
    (send?(act(actions(run(pre(ps2)))(j))) OR
     broadcast?(act(actions(run(pre(ps2)))(j)))) &
    ((send?(act(actions(run(pre(ps1)))(i))) &
      send?(act(actions(run(pre(ps2)))(j)))) =>
         ident(msend(act(actions(run(pre(ps1)))(i)))) =
         ident(msend(act(actions(run(pre(ps2)))(j))))) &
    ((broadcast?(act(actions(run(pre(ps1)))(i))) &
      broadcast?(act(actions(run(pre(ps2)))(j)))) =>
         ident(mbroad(act(actions(run(pre(ps1)))(i)))) =
         ident(mbroad(act(actions(run(pre(ps2)))(j))))) =>
            (ps1 = ps2 & i = j)

Mavail(pre) : bool = FORALL ps, ms1, ms2, t :
   ((received(avail(pre(ps)))(ms1, t) OR
        delivered(avail(pre(ps)))(ms1, t)) &
    (received(avail(pre(ps)))(ms2, t) OR
        delivered(avail(pre(ps)))(ms2, t))) =>
        ms1 = ms2

RValidity(pre) : bool = FORALL ps, ms, i, t :
```

```
   ((received(avail(pre(ps)))(ms, t) OR
        delivered(avail(pre(ps)))(ms, t)) &
    (IF i > 0 THEN time(actions(run(pre(ps)))(i - 1)) < t
     ELSE TimeS(ident(states(run(pre(ps)))(0))) < t
     ENDIF) &
    time(actions(run(pre(ps)))(i)) >= t &
    wait(states(run(pre(ps)))(i))(ms)) =>
       actions(run(pre(ps)))(i) = (# act := read(ms), time := t #)

RIntegrity(pre) : bool = FORALL ps, ms, i, t :
   actions(run(pre(ps)))(i) = (# act := read(ms), time := t #) =>
      ((received(avail(pre(ps)))(ms, t) OR
           delivered(avail(pre(ps)))(ms, t)) &
       wait(states(run(pre(ps)))(i))(ms))

ReadCorrect(pre) : bool =
   MUnique(pre) & Mavail(pre) & RValidity(pre) & RIntegrity(pre)

SValidity(pre) : bool = FORALL ps1, ps2, ms, t1, i :
   (actions(run(pre(ps1)))(i) = (# act := send(ms), time := t1 #) &
    (NOT actions(run(pre(ps1)))(i + 1) =
            (# act := crash, time := t1 #)) &
    dest(ms)(ps2) & Correct(ps2, t1, t1 + delta, pre)) =>
       EXISTS t2 : t2 > t1 & t2 < t1 + delta &
              received(avail(pre(ps2)))(ms, t2)

SIntegrity1(pre) : bool = FORALL ps, ms, t :
   received(avail(pre(ps)))(ms, t) =>
      EXISTS i, t2 : t2 < t & actions(run(pre(sender(ms))))(i) =
                              (# act := send(ms), time := t2 #)

SIntegrity2(pre) : bool = FORALL ps, ms, t1, t2 :
   (received(avail(pre(ps)))(ms, t1) &
    received(avail(pre(ps)))(ms, t2)) => t1 = t2

SendCorrect(pre) : bool =
   SValidity(pre) & SIntegrity1(pre) & SIntegrity2(pre)

BValidity(pre) : bool = FORALL ps1, ps2, ms, t1, i :
   (actions(run(pre(ps1)))(i) = (# act := broadcast(ms), time := t1 #) &
    (NOT actions(run(pre(ps1)))(i + 1) =
            (# act := crash, time := t1 #)) &
    dest(ms)(ps2) & Correct(ps2, t1, t1 + DeltaB, pre)) =>
       EXISTS t2 : t2 > t1 & t2 < t1 + DeltaB &
                   delivered(avail(pre(ps2)))(ms, t2)

BIntegrity1(pre) : bool = FORALL ps, ms, t :
   delivered(avail(pre(ps)))(ms, t) =>
      (EXISTS i, t2 : t2 < t & actions(run(pre(sender(ms))))(i) =
                               (# act := broadcast(ms), time := t2 #))

BIntegrity2(pre) : bool = FORALL ps, ms, t1, t2 :
   (delivered(avail(pre(ps)))(ms, t1) &
```

```
    delivered(avail(pre(ps)))(ms, t2)) => t1 = t2

BUnifAgreement(pre) : bool = FORALL ps1, ps2, ms, t1, i :
   (actions(run(pre(ps1)))(i) = (# act := broadcast(ms), time := t1 #) &
    (EXISTS ps3, t3 : delivered(avail(pre(ps3)))(ms, t3)) &
    dest(ms)(ps2) & Correct(ps2, t1, t1 + DeltaB, pre)) =>
       EXISTS t2 : t2 > t1 & t2 < t1 + DeltaB &
                   delivered(avail(pre(ps2)))(ms, t2)

UTRB(pre) : bool =
   BValidity(pre) & BIntegrity1(pre) & BIntegrity2(pre) &
   BUnifAgreement(pre)

Communication(pre) : bool =
   ReadCorrect(pre) & SendCorrect(pre) & UTRB(pre)

END Communication
```

5.8.3 Correctness properties

```
Names : THEORY
BEGIN

Processes : TYPE+

Coord : Processes

Participants : TYPE = { ps : Processes | NOT ps = Coord }

p1, p2 : VAR Participants

IMPORTING Time

CknowType : TYPE = { f1 : [Participants -> Time] |
                        FORALL p1 : f1(p1) > tstart & f1(p1) < Ctstart }

Cknow : CknowType

IMPORTING cardinal_props [Participants, Participants]

PartFinite : AXIOM is_finite_type [Participants]

N : nat = card(fullset[Participants])

Naxiom : AXIOM N > 1

StateNames : TYPE =
   { Initial, Deciding, Waiting,
     Committed, ComHelp, ComCrashed,
     Aborted, AbHelp, AbCrashed,
     WCrashed, WSending, Willing, WHelp,
     CInitial, CReceiving, CAborting, CCommitting,
     CAborted, CCommitted, CCrashed }
```

```
MessageNames : TYPE =
   { vote_request, yes, no, commit, abort,
     help, reply_commit, reply_abort, reply_willing }

Rest : TYPE = [# noinfo   : setof[Participants],
                 requests : nat,
                 dest     : Participants,
                 oldtout  : Time,
                 repeats  : nat #]

IMPORTING PreRuns [Processes, StateNames, MessageNames, Rest]

ps : VAR Processes
sn : VAR StateNames
st : VAR States

TimeS : [Processes -> Time] = LAMBDA ps :
   IF ps = Coord THEN Ctstart ELSE Cknow(ps) ENDIF

PStateNames(sn) : bool =
   (Initial?(sn) OR Deciding?(sn) OR Waiting?(sn) OR
    Committed?(sn) OR ComHelp?(sn) OR ComCrashed?(sn) OR
    Aborted?(sn) OR AbHelp?(sn) OR AbCrashed?(sn) OR
    WCrashed?(sn) OR WSending?(sn) OR Willing?(sn) OR WHelp?(sn))

PStateSet(st) : bool = PStateNames(name(st))

CStateNames(sn) : bool =
   (CInitial?(sn) OR CReceiving?(sn) OR
    CAborting?(sn) OR CCommitting?(sn) OR
    CCommitted?(sn) OR CAborted?(sn) OR CCrashed?(sn))

CStateSet(st) : bool = CStateNames(name(st))

END Names



Properties : THEORY

BEGIN
IMPORTING Names, PreRuns [Processes, StateNames, MessageNames, Rest]

pre : VAR ComplGloPreRuns
st : VAR States
i, j, k : VAR nat
t1 : VAR Time
p1, p2 : VAR Participants
ps : VAR Processes

ComState(st) : bool =
   Committed?(name(st)) OR ComHelp?(name(st)) OR ComCrashed?(name(st))
```

```
AbState(st) : bool =
   Aborted?(name(st)) OR AbHelp?(name(st)) OR AbCrashed?(name(st))

Commits(p1, pre) : bool = EXISTS i : ComState(states(run(pre(p1)))(i))

Aborts(p1, pre) : bool = EXISTS i : AbState(states(run(pre(p1)))(i))

Decides(p1, pre) : bool = Commits(p1, pre) OR Aborts(p1, pre)

AC1(pre) : bool = FORALL p1, p2 :
   Decides(p1, pre) & Decides(p2, pre) =>
      (Commits(p1, pre) IFF Commits(p2, pre))

EachYes(pre) : bool = FORALL p1 : EXISTS i :
   (send?(act(actions(run(pre(p1)))(i))) &
    name(msend(act(actions(run(pre(p1)))(i)))) = yes &
    sender(msend(act(actions(run(pre(p1)))(i)))) = p1 &
    dest(msend(act(actions(run(pre(p1)))(i)))) = singleton(Coord))

AC2(pre) : bool = (EXISTS p1 : Commits(p1, pre)) => EachYes(pre)

NoCrashes(pre) : bool = FORALL ps : AlwaysCorrect(ps, pre)

AC3(pre) : bool =
   (EachYes(pre) & NoCrashes(pre)) => FORALL p1 : Commits(p1, pre)

AC4(pre) : bool = FORALL p1, i, j :
   ((ComState(states(run(pre(p1)))(i)) & j > i) =>
        ComState(states(run(pre(p1)))(i))) &
   ((AbState(states(run(pre(p1)))(i)) & j > i) =>
        AbState(states(run(pre(p1)))(i)))

AC5(pre) : bool = FORALL p1 : AlwaysCorrect(p1, pre) => Decides(p1, pre)

tf : VAR [Participants -> Time]
B : VAR Time
D : VAR { t1 | t1 > 0 }

AC6(pre) : bool =
   EXISTS B, D :
      FORALL p1 :
         (EXISTS i, tf :
            recover?(act(actions(run(pre(p1)))(i))) &
            (FORALL p2 : p2 /= p1 =>
                tf(p2) >= time(actions(run(pre(p1)))(i)) &
                tf(p2) >= B &
                Correct(p1, tf(p2), tf(p2) + D, pre) &
                Correct(p2, tf(p2), tf(p2) + D, pre)))
          => Decides(p1, pre)

ACAll(pre) : bool =
   AC1(pre) & AC2(pre) & AC3(pre) & AC4(pre) & AC5(pre) & AC6(pre)
```

```
END Properties
```

5.8.4 Protocol performed by each participant

```
PartDef : THEORY
BEGIN

IMPORTING Names

st, st1, st2 : VAR States
pst, pst1, pst2 : VAR (PStateSet)
a1, a2 : VAR Actions
p1, p2 : VAR Participants
pset : VAR setof[Participants]
ms : VAR Messages

v_request(ms) : bool =
   name(ms) = vote_request &
   sender(ms) = Coord & dest(ms) = fullset[Participants]

decision(ms) : bool =
   (name(ms) = commit OR name(ms) = abort) &
    sender(ms) = Coord & dest(ms) = fullset[Participants]

help_request(ms) : bool =
   name(ms) = help & (NOT sender(ms) = Coord)

state_info(pset)(ms) : bool =
   (name(ms) = reply_commit OR name(ms) = reply_abort OR
       name(ms) = reply_willing) &
   (NOT sender(ms) = Coord) & pset(sender(ms))

InitialS(p1) : (PStateSet) =
   (# name  := Initial,
      ident := p1,
      tout  := Cknow(p1) + DeltaC + delta,
      wait  := v_request,
      rest  := (# noinfo   := remove(p1, fullset[Participants]),
                  requests := 0,
                  dest     := p1,
                  oldtout  := 0,
                  repeats  := 0 #) #)

PartPre0(pst1, a1) : bool =
   (timeout?(act(a1)) => time(a1) = tout(pst1)) &
   (recover?(act(a1)) => repeats(rest(pst1)) > 0)

PartPre(st1, a1) : bool = PStateSet(st1) & PartPre0(st1, a1)

IEffect(pst1, a1, pst2) : bool =
   (read?(act(a1)) &
    pst2 = pst1 WITH [name := Deciding ,
```

```
                    tout := time(a1),
                    wait := LAMBDA ms : FALSE ]) OR

   (timeout?(act(a1)) &
    pst2 = pst1 WITH [name := Aborted ,
                    tout := time(a1) + DeltaR,
                    wait := help_request ]) OR

   (crash?(act(a1)) &
    pst2 = pst1 WITH [name := AbCrashed ,
                    tout := time(a1) + DeltaR,
                    wait := LAMBDA ms : FALSE ])

DEffect(pst1, a1, pst2) : bool =
   (send?(act(a1)) & name(msend(act(a1))) = yes &
    sender(msend(act(a1))) = ident(pst1) &
    dest(msend(act(a1))) = singleton(Coord) &
    (NOT ident(pst1) = Coord) &
    pst2 = pst1 WITH [name := Waiting,
              tout := Cknow(ident(pst1)) + DeltaC + 2*delta + DeltaB,
                    wait := decision ]) OR

   (send?(act(a1)) & name(msend(act(a1))) = no &
    sender(msend(act(a1))) = ident(pst1) &
    dest(msend(act(a1))) = singleton(Coord) &
    pst2 = pst1 WITH [name := Aborted,
                    tout := time(a1) + DeltaR, wait := help_request ])

WEffect(pst1, a1, pst2) : bool =
   (read?(act(a1)) & name(mread(act(a1))) = commit &
    pst2 = pst1 WITH [name := Committed,
                    tout := time(a1) + DeltaR,
                    wait := help_request ]) OR

   (read?(act(a1)) & name(mread(act(a1))) = abort &
    pst2 = pst1 WITH [name := Aborted,
                    tout := time(a1) + DeltaR,
                    wait := help_request ]) OR

   (timeout?(act(a1)) &
    pst2 = pst1 WITH [name := Aborted ,
                    tout := time(a1) + DeltaR,
                    wait := help_request ]) OR

   (crash?(act(a1)) &
    pst2 = pst1 WITH [name := WCrashed ,
                  tout := time(a1) + DeltaR, wait := LAMBDA ms : FALSE])

CEffect(pst1, a1, pst2) : bool =
   (read?(act(a1)) &
    (NOT sender(mread(act(a1))) = Coord) &
    pst2 = pst1 WITH [name := ComHelp ,
                    tout := time(a1),
```

```
                      wait := LAMBDA ms : FALSE,
                      rest := rest(pst1) WITH
                         [dest := sender(mread(act(a1))),
                          oldtout := tout(pst1) ] ]) OR

   (timeout?(act(a1)) &
    pst2 = pst1 WITH [tout := time(a1) + DeltaR,
                      wait := help_request ]) OR

   (crash?(act(a1)) &
    pst2 = pst1 WITH [name := ComCrashed ,
                      tout := time(a1) + DeltaR,
                      wait := LAMBDA ms : FALSE ])

CHEffect(pst1, a1, pst2) : bool =
   (send?(act(a1)) & name(msend(act(a1))) = reply_commit &
    sender(msend(act(a1))) = ident(pst1) &
    dest(msend(act(a1))) = singleton[Processes](dest(rest(pst1))) &
    pst2 = pst1 WITH [name := Committed,
                      tout := oldtout(rest(pst1)),
                      wait := help_request ])

CCEffect(pst1, a1, pst2) : bool =
   (recover?(act(a1)) &
    pst2 = pst1 WITH [name := Committed,
                      tout := time(a1) + DeltaR,
                      wait := help_request,
                      rest := rest(pst1) WITH [ repeats := 0 ] ]) OR

   (timeout?(act(a1)) &
    pst2 = pst1 WITH [tout := time(a1) + DeltaR,
                      rest := rest(pst1) WITH
                         [ repeats := repeats(rest(pst1)) + 1 ] ])

AEffect(pst1, a1, pst2) : bool =
   (read?(act(a1)) & (NOT sender(mread(act(a1))) = Coord) &
    pst2 = pst1 WITH [name := AbHelp ,
                      tout := time(a1),
                      wait := LAMBDA ms : FALSE,
                      rest := rest(pst1) WITH
                         [dest := sender(mread(act(a1))),
                          oldtout := tout(pst1) ] ]) OR

   (timeout?(act(a1)) &
    pst2 = pst1 WITH [tout := time(a1) + DeltaR,
                      wait := help_request ]) OR

   (crash?(act(a1)) &
    pst2 = pst1 WITH [name := AbCrashed,
                      tout := time(a1) + DeltaR,
                      wait := LAMBDA ms : FALSE ])

AHEffect(pst1, a1, pst2) : bool =
```

```
   (send?(act(a1)) & name(msend(act(a1))) = reply_abort &
    sender(msend(act(a1))) = ident(pst1) &
    dest(msend(act(a1))) = singleton[Processes](dest(rest(pst1))) &
    pst2 = pst1 WITH [name := Aborted,
                      tout := oldtout(rest(pst1)),
                      wait := help_request ])

ACEffect(pst1, a1, pst2) : bool =
   (recover?(act(a1)) &
    pst2 = pst1 WITH [name := Aborted,
                      tout := time(a1) + DeltaR, wait := help_request,
                      rest := rest(pst1) WITH [repeats := 0 ] ]) OR

   (timeout?(act(a1)) &
    pst2 = pst1 WITH [tout := time(a1) + DeltaR,
                      rest := rest(pst1) WITH
                         [repeats := repeats(rest(pst1)) + 1 ] ])

WCEffect(pst1, a1, pst2) : bool =
   (recover?(act(a1)) &
    pst2 = pst1 WITH [name := WSending,
                      tout := time(a1),
                      rest := rest(pst1) WITH [repeats := 0 ] ]) OR

   (timeout?(act(a1)) &
    pst2 = pst1 WITH [tout := time(a1) + DeltaR,
                      rest := rest(pst1) WITH
                         [repeats := repeats(rest(pst1)) + 1 ] ])

WSEffect(pst1, a1, pst2) : bool =
   (send?(act(a1)) & name(msend(act(a1))) = help &
    sender(msend(act(a1))) = ident(pst1) &
    dest(msend(act(a1))) = noinfo(rest(pst1)) &
    pst2 = pst1 WITH [name := Willing,
                      tout := time(a1) + 2*delta,
                      wait := LAMBDA ms :
                              state_info(noinfo(rest(pst1)))(ms) OR
                              help_request(ms),
                      rest := rest(pst1) WITH
                         [requests := requests(rest(pst1)) + 1 ] ])

WLEffect(pst1, a1, pst2) : bool =
   (read?(act(a1)) & name(mread(act(a1))) = reply_commit &
    (NOT sender(mread(act(a1))) = Coord) &
    pst2 = pst1 WITH [name := Committed,
                      tout := time(a1) + DeltaR,
                      wait := help_request ]) OR

   (read?(act(a1)) & name(mread(act(a1))) = reply_abort &
    (NOT sender(mread(act(a1))) = Coord) &
    pst2 = pst1 WITH [name := Aborted,
                      tout := time(a1) + DeltaR,
                      wait := help_request ]) OR
```

```
   (read?(act(a1)) & name(mread(act(a1))) = reply_willing &
    (NOT sender(mread(act(a1))) = Coord) &
    noinfo(rest(pst1))(sender(mread(act(a1)))) &
       (IF remove(sender(mread(act(a1))), noinfo(rest(pst1))) =
             emptyset[Participants]
        THEN
        pst2 = pst1 WITH [name := Committed,
                          tout := time(a1) + DeltaR,
                          wait := help_request ]
        ELSE
        pst2 = pst1 WITH
           [wait := LAMBDA ms :
               state_info(remove(sender(mread(act(a1))),
                             noinfo(rest(pst1))))(ms) OR
               help_request(ms),
            rest := rest(pst1) WITH
               [noinfo := remove(sender(mread(act(a1))),
                             noinfo(rest(pst1))) ] ]
        ENDIF )) OR

   (read?(act(a1)) & name(mread(act(a1))) = help &
    (NOT sender(mread(act(a1))) = Coord) &
    pst2 = pst1 WITH [name := WHelp ,
                      tout := time(a1), wait := LAMBDA ms : FALSE,
                      rest := rest(pst1) WITH
                         [dest := sender(mread(act(a1))),
                          oldtout := tout(pst1) ] ]) OR

   (timeout?(act(a1)) &
    pst2 = pst1 WITH [name := WSending,
                      tout := time(a1), wait := LAMBDA ms : FALSE ]) OR

   (crash?(act(a1)) &
    pst2 = pst1 WITH [name := WCrashed ,
                      tout := time(a1) + DeltaR,
                      wait := LAMBDA ms : FALSE ])

RHEffect(pst1, a1, pst2) : bool =
   (send?(act(a1)) & name(msend(act(a1))) = reply_willing &
    sender(msend(act(a1))) = ident(pst1) &
    dest(msend(act(a1))) = singleton[Processes](dest(rest(pst1))) &
    pst2 = pst1 WITH [name := Willing,
                      tout := oldtout(rest(pst1)),
                      wait := LAMBDA ms :
                                 state_info(noinfo(rest(pst1)))(ms) OR
                                 help_request(ms) ])

PartEffect0(pst1, a1, pst2) : bool =
   CASES name(pst1) OF

Initial : IEffect(pst1, a1, pst2),
Deciding : DEffect(pst1, a1, pst2),
```

```
Waiting : WEffect(pst1, a1, pst2),
Committed : CEffect(pst1, a1, pst2),
ComHelp : CHEffect(pst1, a1, pst2),
ComCrashed : CCEffect(pst1, a1, pst2),
Aborted : AEffect(pst1, a1, pst2),
AbHelp : AHEffect(pst1, a1, pst2),
AbCrashed : ACEffect(pst1, a1, pst2),
WCrashed : WCEffect(pst1, a1, pst2),
WSending : WSEffect(pst1, a1, pst2),
Willing : WLEffect(pst1, a1, pst2),
WHelp : RHEffect(pst1, a1, pst2)

   ENDCASES

PartEffect(st1, a1, st2) : bool =
   PStateSet(st1) & PStateSet(st2) & PartEffect0(st1, a1, st2)

END PartDef
```

5.8.5 Protocol performed by the coordinator

```
CoordDef : THEORY
BEGIN

IMPORTING Names

st1, st2 : VAR States
cst, cst1, cst2 : VAR (CStateSet)
a1, a2 : VAR Actions
p1, p2 : VAR Participants
pset : VAR setof[Participants]
ms : VAR Messages

vote(pset)(ms) : bool =
  (name(ms) = yes OR name(ms) = no) &
  (NOT sender(ms) = Coord) & pset(sender(ms)) &
   dest(ms) = singleton(Coord)

p0 : Participants

CInitialS  : (CStateSet) =
   (# name := CInitial,
      ident := Coord,
      tout := Ctstart,
      wait := LAMBDA ms : FALSE,
      rest := (# noinfo   := fullset[Participants],
                 oldtout  := 0,
                 dest     := p0,
                 requests := 0,
                 repeats  := 0 #) #)

CoordPre0(cst1, a1) : bool =
   timeout?(act(a1)) => time(a1) = tout(cst1)
```

```
CoordPre(st1, a1) : bool = CStateSet(st1) & CoordPre0(st1, a1)

InitialE(cst1, a1, cst2) : bool =
   (send?(act(a1)) & name(msend(act(a1))) = vote_request &
    sender(msend(act(a1))) = Coord &
    dest(msend(act(a1))) = fullset[Participants] &
    cst2 = cst1 WITH [name := CReceiving,
                      tout := time(a1) + 2*delta,
                      wait := vote(noinfo(rest(cst1))) ])

ReceivingE(cst1, a1, cst2) : bool =
   (read?(act(a1)) & name(mread(act(a1))) = yes &
    (NOT sender(mread(act(a1))) = Coord) &
    (IF remove(sender(mread(act(a1))), noinfo(rest(cst1))) =
          emptyset[Participants]
     THEN
     cst2 = cst1 WITH [name := CCommitting,
                       tout := time(a1), wait := LAMBDA ms : FALSE]
     ELSE
     cst2 = cst1 WITH [wait := vote(remove(sender(mread(act(a1))),
                                           noinfo(rest(cst1)))),
                       rest := rest(cst1) WITH
                          [noinfo := remove(sender(mread(act(a1))),
                                            noinfo(rest(cst1))) ] ]
     ENDIF )) OR

   (read?(act(a1)) & name(mread(act(a1))) = no &
    cst2 = cst1 WITH [name := CAborting,
                      tout := time(a1), wait := LAMBDA ms : FALSE]) OR

   (timeout?(act(a1)) &
    cst2 = cst1 WITH [name := CAborting,
                      tout := time(a1), wait := LAMBDA ms : FALSE]) OR

   (crash?(act(a1)) &
    cst2 = cst1 WITH [name := CCrashed,
                      tout := time(a1) + DeltaR,
                      wait := LAMBDA ms : FALSE])

CommittingE(cst1, a1, cst2) : bool =
   (broadcast?(act(a1)) & name(mbroad(act(a1))) = commit &
    sender(mbroad(act(a1))) = Coord &
    dest(mbroad(act(a1))) = fullset[Participants] &
    cst2 = cst1 WITH [name := CCommitted, tout := time(a1) + DeltaR])

AbortingE(cst1, a1, cst2) : bool =
   (broadcast?(act(a1)) & name(mbroad(act(a1))) = abort &
    sender(mbroad(act(a1))) = Coord &
    dest(mbroad(act(a1))) = fullset[Participants] &
    cst2 = cst1 WITH [name := CAborted, tout := time(a1) + DeltaR ])

CommittedE(cst1, a1, cst2) : bool =
```

```
   (timeout?(act(a1)) &
    cst2 = cst1 WITH [tout := time(a1) + DeltaR ]) OR

   (crash?(act(a1)) &
    cst2 = cst1 WITH [name := CCrashed, tout := time(a1) + DeltaR ])

AbortedE(cst1, a1, cst2) : bool =
   (timeout?(act(a1)) &
    cst2 = cst1 WITH [tout := time(a1) + DeltaR ]) OR

   (crash?(act(a1)) &
    cst2 = cst1 WITH [name := CCrashed, tout := time(a1) + DeltaR ])

CrashedE(cst1, a1, cst2) : bool =
   (timeout?(act(a1)) &
    cst2 = cst1 WITH [tout := time(a1) + DeltaR ])

CoordEffect0(cst1, a1, cst2) : bool =
   CASES name(cst1) OF

CInitial : InitialE(cst1, a1, cst2),
CReceiving : ReceivingE(cst1, a1, cst2),
CCommitting : CommittingE(cst1, a1, cst2),
CAborting : AbortingE(cst1, a1, cst2),
CCommitted : CommittedE(cst1, a1, cst2),
CAborted : AbortedE(cst1, a1, cst2),
CCrashed : CrashedE(cst1, a1, cst2)

   ENDCASES

CoordEffect(st1, a1, st2) : bool =
   CStateSet(st1) & CStateSet(st2) & CoordEffect0(st1, a1, st2)

END CoordDef
```

5.8.6 Definition of runs and main theorem

```
Protocols [Processes, StateNames, MessageNames, Rest : TYPE,
   (IMPORTING PreRuns [Processes, StateNames, MessageNames, Rest])
   TimeS : [Processes -> Time],
   Initial : [Processes -> States],
   Pre : [Processes -> [States, Actions -> bool]],
   Effect : [Processes -> [States, Actions, States -> bool] ] ] :
      THEORY
BEGIN

pre : VAR ComplGloPreRuns
ps : VAR Processes
i, j, k : VAR nat

Protocols(pre) : bool = FORALL ps :
   states(run(pre(ps)))(0) = Initial(ps) &
   time(actions(run(pre(ps)))(0)) >=
```

```
      TimeS(ident(states(run(pre(ps)))(0))) &
   FORALL i : ident(states(run(pre(ps)))(i)) = ps &
              Pre(ps)(states(run(pre(ps)))(i),
                      actions(run(pre(ps)))(i)) &
              Effect(ps)(states(run(pre(ps)))(i),
                         actions(run(pre(ps)))(i),
                         states(run(pre(ps)))(i + 1))

END Protocols



Main : THEORY
BEGIN

IMPORTING PartDef, CoordDef, Properties,
   Communication [Processes, StateNames, MessageNames, Rest, TimeS]

ps : VAR Processes
st1, st2 : VAR States
a1, a2 : VAR Actions
pre : VAR ComplGloPreRuns

Initial(ps) : States =
   IF ps = Coord THEN CInitialS ELSE InitialS(ps) ENDIF

Pre(ps)(st1, a1) : bool =
   IF ps = Coord THEN CoordPre(st1, a1) ELSE PartPre(st1, a1) ENDIF

Effect(ps)(st1, a1, st2) : bool =
   IF ps = Coord THEN CoordEffect(st1, a1, st2)
   ELSE PartEffect(st1, a1, st2) ENDIF

IMPORTING Protocols [Processes, StateNames, MessageNames, Rest,
                     TimeS, Initial, Pre, Effect]

Runs : TYPE = { pre | Protocols(pre) }

r : VAR Runs

Main : THEOREM Communication(r) => ACAll(r)

END Main
```

Chapter 6

Conclusions

During the last 25 years, there has been a rapid growth of the literature on database concurrency control and recovery. Since the notion of serializability and the Two Phase Locking protocol were introduced in [EGLT76], hundreds of papers with new algorithms and correctness notions have been published. During each decade, algorithms in this area address new challenges, such as the need for greater performance, fault-tolerance and real-time. Subsequently, they are also becoming increasingly complex, and their correctness becomes difficult to ensure.

Formal specification and verification is a widely recognized method to improve the understanding of computer protocols and to find errors in them. A large number of protocols has been verified in the areas of access to shared variables (e.g., mutual exclusion protocols), distributed consensus and network protocols [LL90]. Most of the verification methods are based on some version of Hoare logic [Hoa69, Hoo91], temporal logic [MP91, MP95, Lam94], automata [Lyn96, AD94], process algebra [BW90, J.C90], or a combination of these methods.

During the last decade, one of the main trends is the use of some form of mechanical support for verification, and a large number of interesting protocols have been verified using proof checkers. E.g., an I/O automata framework has been (partially) formalized in the PVS, Coq and Larch systems and applied to the verification of a leader election protocol [Gri00], an audio control protocol [HSV94] and a reliable communication protocol [SAGG+93], respectively. The specification language μCRL has been encoded in the Coq, PVS and Isabelle systems and used to verify a bounded retransmission protocol [GvdP96], a distributed summation protocol [GMS97] and the SLIP protocol [GMvdP98], respectively. PVS has also been successfully applied to the verification of the Fisher mutual exclusion protocol [Sha93], a clock synchronization algorithm [RvH89], a steam boiler system [VH96] and a group membership protocol [Hoo97].

However, there has been relatively little work on the formal verification of concurrency control and recovery protocols [BHG87, Pap86, LMWF94, Bro92, KS92, LS92, Kuo96, JZ92, KST00, DF93, Sch96]. These verifications often treat only very simplified versions of protocols, and usually do not use mechanical support.

In this thesis, we have tried to fill this gap and achieved a certain success. A number of correctness notions for such protocols have been formalized in the higher-order logic of PVS and their relations have been studied. These notions have been verified for a number of protocols in the areas of "classical" concurrency control (chapter 3), fault-tolerant concurrency control and recovery (chapter 4) and distributed commitment (chapter 5). As already mentioned in section 1.6, our formal study of these protocols achieved not only their mechanized verification, but also a number of additional results: a method to extend concurrency control protocols with new actions and information (chapter 3), an

efficient integration of a few related concurrency control and recovery protocols into a single system (chapter 4), the discovery of an error in a previously published distributed commitment protocol, including its correction (chapter 5). A more detailed description of the achievements of this work can be found in section 1.6.

To specify our protocols, we use a combination of state machines and global assertions on system behaviours. According to this methodology, less interesting parts of a protocol are replaced by global assertions, expressing the properties that these parts are supposed to achieve. By doing this, we avoid the need to consider the implementation of these less interesting parts. E.g., in chapter 4, we abstract from the implementation of distributed commitment because our main interest is memory management; in chapter 5, we abstract from the details of the reliable broadcast mechanism because our main interest is distributed commitment. Such an approach greatly decreases the complexity of our specifications and the verification task.

In the rest of this chapter, we give some remarks about our experience with PVS (section 6.1) and discuss possible directions for future work (section 6.2).

6.1 Reflections on the use of PVS

In our PVS specifications, we treat an arbitrary number of sites, different types of memory (stable and volatile), transaction and memory failures, advanced communication mechanisms such as reliable broadcast, continuous time and timeouts. At the same time, our specifications are not extremely complicated, and our verification efforts can be described as reasonable (a few months were needed to develop the PVS proof for each of the three "case studies"). PVS provided great help in developing and managing the proofs. It is clear that without PVS it would be virtually impossible to develop and check formal proofs of this size (consisting of a few hundred lemmas).

The table below shows an approximation of the statistical information about our use of PVS for each of the verifications presented in the three main chapters. The columns describe the following characteristics of our proofs:

- The number of PVS theories developed by the author. It includes both theories with specifications and theories with lemmas (only theories with specifications are included in this thesis).
- The total number of PVS lemmas and theorems. Type-correctness conditions (TCCs) are not included here, because they are usually proved very easily or even automatically.
- The number of lines in the proof. According to our estimations, the number of actual proof commands (which represent interactions with the PVS proof checker) is approximately half of this due to a large number of lines consumed by brackets.
- The amount of time needed to check the proofs. We used PVS release 2.3 running on a 333 Mhz Spark 10 workstation with 128 MB internal memory.

Chapter	PVS theories	Lemmas	Proof lines	Run time
Chapter 3	19	155	13000	11 minutes
Chapter 4	14	150	12500	13 minutes
Chapter 5	19	270	12500	24 minutes

Note that the PVS specifications in chapter 4 import some theories from chapter 3. In the above table, the statistical data for these additional theories is not double-counted in the line corresponding to chapter 4.

In general, our experience with PVS has been quite positive. The specification language of PVS is very expressive, very readable and can be learned in a couple of days. This is certainly not the case for some other popular verification systems such as Isabelle and Coq [GMvdP98, Gri00]. The representation of proofs is PVS is quite intuitive, and there are powerful decision procedures that usually work fine. The proof management is also excellent and can efficiently keep track of even very large proofs.

However, we have some negative remarks as well. A problem of PVS is that there are quite a number of bugs in the system, leading to crashes or difficult to understand error messages. These problems are probably caused by the fact that decision procedures make the kernel of PVS large and complex [Gri00]. Although the developers of PVS fix many bugs in subsequent patches, the general stability of the system does not seem to be improving.

The bugs seldom influence soundness, and, although some people have managed to prove $true = false$ in PVS, the author probably has never proved a single incorrect lemma. However, the bugs often make it impossible to type-check a theory or complete a proof of a correct lemma. Some bugs can be solved by restarting the PVS system (which can take a while), but others require a few hours to find a way around them. The author estimates that he spent about a quarter of the time he worked with PVS for fighting bugs and dealing with strange behaviour of the system.

Another problem of PVS is that subsequent versions usually are not fully compatible. Some changes are usually needed in the specifications and the proofs to adapt them to the next version of PVS. This is because in a new version some definitions of the specification language become no longer valid and some decision procedures work differently. E.g., when we switched from PVS 2.2 to PVS 2.3, approximately 5 percent of our proofs stopped to work. These proofs required only rather small changes, but the necessity to perform this adaptation was still quite annoying.

6.2 Future work

The most obvious direction of future work is to apply our framework to the verification of new types of protocols. Of course, there is a huge number of protocols for different aspects of concurrency control and recovery, and the methods presented in this thesis may not be suitable for all of them. For instance, if a protocol involves only a small number of data items and transactions, it may be more efficient to apply model-checking and not theorem proving (see section 1.3). Also, parts of a protocol may only perform some coordination or control activity without actually accessing data items (e.g., electing a coordinator among a number of participants of a transaction), and it may again be preferable to use model-checking for such parts. Our formal model also has a few limitations. For instance, in this model the desired properties of a protocol are expressed by logical formulas and not by "operational" specifications. This property-oriented style may not be suitable for some protocols. We also do not define an implementation relation between two operational specifications.

Therefore, in this chapter we restrict ourselves to the discussion of protocols that are rather close to the protocols presented in the previous chapters, and that can be efficiently treated by our methods. At the same time, they are too different from the previously presented protocols to verify them using the idea of extensions from section 3.5. In the area of classical serializability theory, we suggest to study the verification of *graph-based locking* protocols, i.e. lock-based protocols that are not two-phase, but use an ordering on data items in the form of a directed graph. The main ideas of such protocols are outlined in section 6.2.1. In section 6.2.2, we discuss how to extend the classical theory to *multiversion serializability theory*, which allows to keep several *versions* of a data item in the database (produced by different transactions), and weakens the serializability requirement such that transactions do not

always have to read the latest version. We show that multiversion serializability can be rather easily formalized by slightly modifying the model used in chapter 4 (which also has two versions of data items, stable and volatile, but for a different purpose).

Another way to continue our research is to study concurrency control and recovery protocols, using models that are (completely) different from the models employed in this thesis, and to compare the results of this approach with our results. In our opinion, it seems especially interesting to study how the methods of *process algebra* can be applied to our protocols. Process algebra, with its main branches ACP (Algebra of Communicating Processes) [BW90], CCS (Calculus of Communicating Systems) [Mil80] and CSP (Communicating Sequential Processes) [Hoa85], is one of the few major formalisms that are radically different from state machines and (temporal) logics. In process algebra, a distributed protocol is usually modelled as a parallel composition of a few sequential processes that are represented by a number of recursive equations. Specification of a protocol is also given by a sequential or parallel process, and verification must establish a certain form of trace inclusion between the protocol and its specification.

Process algebra has been mainly applied to communication protocols so far. For such protocols, the specification can often be given by a rather simple sequential process (representing, for instance, the desired behaviour of a buffer with several places). However, it seems much more challenging to give a simple operational specification for a serializability property, because it involves a subtle interaction of a user and an arbitrary number of transactions and data items. In an I/O automata framework, such operational specification is rather complicated [LMWF94].

For process algebra, the use of recursive equations often leads to rather compact definitions. Moreover, for the ACP version [BW90, J.C90], efficient methods of proving that a protocol satisfies its specification have been developed in the last decade, for example, the cones and foci method [GS95]. This method is based on the specification language μCRL [GP94, mCR], is supported by the μCRL toolset, and has been applied to verify a few complicated communication protocols [BBG97, KS94, GvdP96]. It is especially encouraging that some of the proofs have been checked by the proof checkers Coq, PVS and Isabelle (see [GvdP96], [GMS97] and [GMvdP98], respectively). Therefore, it remains to be seen whether a process algebra framework can provide an elegant specification of serializability, and whether this specification can lead to an efficient verification methodology for concurrency control and recovery protocols.

6.2.1 Graph-based locking protocols

Although the Two Phase Locking protocol is simple and ensures serializability, it also has some disadvantages such as long waiting times for some transactions and deadlocks (see section 2.2.1). To address these problems, a number of protocols have been developed that are lock-based yet not two-phase. Such protocols use some additional information on how each transaction will access the database. Graph-based locking protocols are based on the simplest model, in which we have prior knowledge of the order in which the data items will be accessed.

In graph-based locking protocols, the set of data items may be viewed as a directed acyclic graph, called a *database graph*. If a data item x precedes a data item y in the database graph, then any transaction accessing x and y must access x before accessing y. This partial ordering may result from either the logical or physical organization of the data, or it may be imposed only for the purpose of concurrency control.

In this section, we consider only database graphs that are rooted trees. We present a simple protocol for such graphs from [SKS97] (originally published in [SK80]), which uses only exclusive locks. In this protocol, each transaction T can lock a data item x by performing action $Xlock(T, x)$

and must observe the following rules:

1. The first lock by T may be on any data item.
2. Subsequently, a data item x can be locked by T only if the parent of x is currently locked by T.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T cannot subsequently be relocked by T.

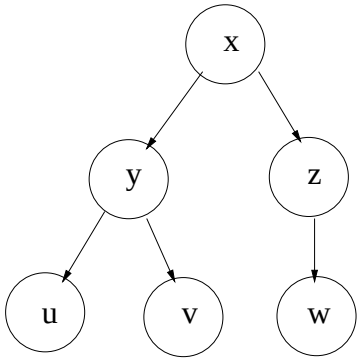

Figure 6.1: Example of a database graph.

To illustrate the protocol, consider the database graph of Figure 6.1. The following interleaved execution of transactions $T1$ and $T2$ is allowed by the protocol, in which we only show lock and unlock actions:

$Xlock(T1, x)$ $Xlock(T2, y)$ $Xlock(T1, z)$ $Xlock(T2, u)$ $Xlock(T1, w)$
$Unlock(T2, u)$ $Unlock(T2, y)$ $Xlock(T1, y)$ $Unlock(T1, x)$ $Xlock(T1, v)$
$Unlock(T1, z)$ $Unlock(T1, v)$ $Unlock(T1, y)$ $Unlock(T1, w)$

Note that this schedule does not satisfy the Two Phase Locking protocol, because $T1$ locks v after it unlocked x. It is also easy to construct schedules that satisfy the Two Phase Locking protocol yet are not allowed by this tree-based protocol; for example, $Xlock(T1, y)Xlock(T1, x)Unlock(T1, x)$ $Unlock(T1, y)$ is one of such schedules.

As stated in [SKS97], this tree-based protocol has the advantage over two-phase protocols in that unlocking may occur earlier. This may result in less waiting time, and an increase in concurrency. In addition, the protocol is deadlock-free, so no aborts of deadlocked transactions are required. However, the protocol has the disadvantage that in some cases a transaction needs to lock data items that it does not access. For example, a transaction that wants to access data items v and w in the database graph of Figure 6.1 must lock not only v and w but also data items x, y and z. This leads to increased locking overhead, and the possibility of additional waiting time and decrease in concurrency.

A few graph-based locking protocols are presented in [SK80, KS83, BS85]. The protocol in this section originates from [SK80]. In that article, a correctness proof is given that is based on graph theory and an ordering on transactions similar to our notion of conflict preserving timestamps from section 3.2. The proof does not appear extremely formal, and it would be interesting to see how difficult it is to formalize the proof in PVS (especially because we did not pay much attention to graph theory in this thesis), and whether it contains significant errors. The protocol from [SK80] has been extended in [KS83] to shared locks, and it may be worthwhile to consider the proof for the extended protocol as well.

6.2.2 Multiversion serializability theory

In this section, we informally describe the main ideas of the multiversion serializability theory based on [BHG87, SKS97] and discuss the possible ways of formalizing it.

In the serializability theory presented in chapter 3, transactions that violate the serializability order are either aborted or delayed. For instance, in the Timestamp Ordering protocol, a transaction is aborted if its read action attempts to read a value that has already been overwritten. In the Two Phase Locking protocol, a read or write action is delayed if the data item that it wants to access is currently locked in an incompatible mode. In the *multiversion serializability theory*, such aborts and delays are avoided by keeping old *versions*, i.e. old copies of each data item. As in chapter 4, versions are labelled by identifiers of transactions that produced these versions, i.e. a write of x by transaction T leads to a new versions x_T. A read of x by transaction T obtains any of the old versions of x, with one exception: T must read a version of x produced by itself if such version exists. E.g., a schedule $S1 = Write(T1, x, x_{T1})Write(T2, x, x_{T2})Read(T3, x, x_{T1})$ is allowed by the multiversion theory, whereas $S2 = Write(T1, x, x_{T1})Write(T2, x, x_{T2})Read(T1, x, x_{T2})$ is not. Note that $S1$ is not allowed by the classical serializability theory.

The correctness condition for the multiversion serializability theory is called *one-copy serailizability* [BHG87], which is informally defined as follows. We say that a schedule is *one-copy serial*, if it is serial (as defined in section 3.1), and in addition each read obtains exactly the *last version* of a data item (note that the last assumption is satisfied automatically in the model of chapter 3). E.g., the schedule $S1$ defined above is serial, but not one-copy serial, because a read by $T3$ obtains an old version x_{T1} instead of the last version x_{T2}. A schedule is said to be *one-copy serializable* if there is a one-copy serial schedule, consisting of the same actions. Unlike section 3.1, this serial schedule is not required to be produced from the original schedule by a sequence of non-conflicting swaps. For example, $S1$ is one-copy serializable, because schedule $Write(T1, x, x_{T1})Read(T3, x, x_{T1})Write(T2, x, x_{T2})$ is one-copy serial and consists of the same actions as $S1$.

The notion of one-copy serial schedules is used to eliminate illegal schedules. For example, schedule $S3 = Write(T1, x, x_{T1})Read(T2, x, x_{T1})Write(T2, y, y_{T2})Read(T1, y, y_{T2})$ consists of the same actions as serial schedule $S4 = Write(T1, x, x_{T1})Read(T1, y, y_{T2})Read(T2, x, x_{T1})$ $Write(T2, y, y_{T2})$. However, $S3$ is not one-copy serializable, because $S4$ is not one-copy serial. In fact, $S4$ is illegal, because $T1$ reads a version of y that has not yet been produced. It is easy to see that if in a schedule each read obtains the last version of a data item, then this property is preserved by each non-conflicting swap (as defined in section 3.1). This implies that each serializable schedule (as defined in section 3.1) is also one-copy serializable, so the multiversion theory indeed uses a weaker notion of serializability.

Both the Timestamp Ordering and the Two-Phase Locking protocols can be adapted for the multiversion theory. Here we only outline the adaptation of the Timestamp Ordering protocol because it is much more common. In this adaptation, each transaction T that entered the system has a unique fixed timestamp $TS(T)$, which is assigned in the same manner as in section 2.2.3. With each data item x a sequence of version is associated. Each version x_k contains a value and two timestaps $Wts(x_k)$ (write timestamp) and $Rts(x_k)$ (read timestamp). The definition of these timestamps is basically the same as in section 2.2.3: $Wts(x_k)$ denotes the timestamp of transaction that *created* x_k, and $Rts(x_k)$ denotes the largest timestamp of any transaction that successfully *read* version x_k.

The protocol proceeds as follows [SKS97]. Suppose that transaction T wants to read or write x. Let x_k denote the version of x whose write timestamp is the largest write timestamp not greater than $TS(T)$.

- If T wants to read x, such action is always allowed, and it obtains the value of version x_k. If

$Rts(x_k) < TS(T)$, then $Rts(x_k)$ is updated to $TS(T)$.

- If T wants to write x and $TS(T) < Rts(x_k)$, then T is aborted. Otherwise, write action is performed, and it creates a new version x_m such that $Wts(x_m) = Rts(x_m) = TS(T)$.

Unlike the protocol from section 2.2.3, this adaptation never aborts read actions, but it also has an obvious disadvantage: old versions of data items may consume a lot of memory.

Formalization and verification. It seems that multiversion serializability theory can easily be treated in our framework, including the formalization of all definitions in PVS. For instance, one-copy serializability can be formalized in a very straightforward way by combining some definitions from chapters 3 and 4. Indeed, one-copy serial schedules can be defined by a predicate similar to predicate $UpdateInPlace$ from section 4.1. The definition of schedules consisting of the same actions can be based on a sequence of swaps transforming one schedule into the other (as in section 3.1, only swaps don't have to be non-conflicting). The adaptation of the Timestamp Ordering protocol given above is only slightly more complicated than the original protocol and also can be easily formalized.

Finally, we can also define a complete and sound method for verification of one-copy serializability based on the method from [BHG87] and our method for classical serializability. Indeed, the method for classical serializability from section 3.2 can be easily adapted to the multiversion theory by modifying the notions of *conflict relations* and *conflict-preserving timestamps*. The definition of a conflict relation is more complicated than in the classical case, and uses the additional notion of *version orders*. We say that $\ll$ is a version order for x in a schedule S, if it is a total order on versions of x that appear in S. We say that $\ll$ is a version order for a schedule S, if it is the union of version orders for all data items. If S is a schedule and $\ll$ is a version order for S, then the conflict relation $Conflict(S, \ll)$ is defined as follows. Firstly, we include in the relation all pairs $(T1, T2)$ that belong to relation $Conflict(S)$ from section 3.2 (where different versions should be considered as different data items, e.g., actions $Write(T1, x, x_{T1})$ and $Write(T2, x, x_{T2})$ are no longer conflicting). Secondly, *version order edges* must be included, i.e. for each actions $Write(T1, x, x_{T1})$ and $Read(T3, x, x_{T2})$ in S, where $T1$, $T2$ and $T3$ are distinct, if $x_{T1} \ll x_{T2}$ then we include pair $(T1, T2)$, otherwise we include pair $(T3, T1)$. For example, if $S3 = Write(T1, x, x_{T1})Write(T2, x, x_{T2})Read(T3, x, x_{T1})Read(T4, x, x_{T2})$ and $\ll = \{(x_{T2}, x_{T1})\}$, then $Conflict(S3, \ll) = \{(T1, T3), (T2, T4), (T2, T1), (T4, T1)\}$. Here $(T1, T3)$ and $(T2, T4)$ belong to $Conflict(S3)$, whereas $(T2, T1)$ and $(T4, T1)$ are version order edges.

Using this definition of a conflict relation, a necessary and sufficient condition for one-copy serializability (corresponding to theorem 5.4 from [BHG87]) can be formulated as follows.

Theorem. A schedule S is one-copy serializable if and only if there exist a version order $\ll$ and a timestamp TS that is a conflict-preserving timestamp for S and $\ll$, i.e. for each transactions $T1$ and $T2$ the following condition holds:

$$Conflict(S, \ll)(T1, T2) \Longrightarrow TS(T1) < TS(T2)$$

For example, if for the schedule $S3$ given above we take $\ll = \{(x_{T2}, x_{T1})\}$ and $TS(T2) = 1$, $TS(T4) = 2$, $TS(T1) = 3$ and $TS(T3) = 4$, then it is easy to check that TS is conflict-preserving for S and $\ll$. The theorem above implies that $S3$ is one-copy serializable. Indeed, it is equivalent to one-copy serial schedule $Write(T2, x, x_{T2})Read(T4, x, x_{T2})Write(T1, x, x_{T1})Read(T3, x, x_{T1})$.

It would be quite interesting to formalize this theorem in PVS and try to prove it. Although we expect the proof to be considerably more difficult than the proof of the corresponding theorem for the

classical theory (i.e. theorem $OrdConfEquiv$ from section 3.2), such proof will probably greatly improve our understanding of the multiversion serializability theory.

Bibliography

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[BBG97] M.A. Bezem, R. Bol, and J.F. Groote. Formalizing process algebraic verifications in the calculus of construction. *Formal Aspects of Computing*, 9(1):1–48, 1997.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Comp., 1987.

[BLS97] A. Bestavros, K.J. Lin, and S. Son. *Real-Time Database Systems: Issues and Applications*. Kluwer, 1997.

[Bod99] M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. PhD thesis, Eindhoven University of Technology, Den Dolech 2, Eindhoven, The Netherlands, 1999.

[Bos01] D. Bosnacki. *Enhancing State Space Reduction Techniques for Model Checking*. PhD thesis, Eindhoven University of Technology, 2001.

[Bro92] M. Broy. Algebraic and functional specification of an interactive serializable database interface. *Distributed Computing*, 6(1):5–18, 1992.

[BS85] G. Buckley and A. Silberschatz. Beyond two-phase locking. *Journal of the ACM*, 32(2):314–326, 1985.

[BT93a] O. Babaoglu and S. Toueg. Non-blocking atomic commitment. In *Distributed Systems*, pages 147–168. Addison-Wesley Publishing Comp., 1993.

[BT93b] O. Babaoglu and S. Toueg. Understanding non-blocking atomic commitment. Technical Report UBLCS-93-2, University of Bologna, 31 pages, 1993.

[BvdS98] M. Bodlaender and P. van der Stok. A transaction-based temporal data model that supports prediction in real-time databases. In *Proc. of the 10th Euromicro Workshop on Real Time Systems*, pages 197–203. IEEE, 1998.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process algebra*. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[CHvdS99] D. Chkliaev, J. Hooman, and P. van der Stok. Serializability preserving extensions of concurrency control protocols. In *Proc. of the A. Ershov Third Int'l Conf. Perspectives of System Informatics*, pages 180–193. LNCS 1755, 1999.

[CHvdS00a] D. Chkliaev, J. Hooman, and P. van der Stok. Formal modelling and analysis of non-blocking atomic commitment protocols. In *Proc. of ICPADS'2000 (7th International Conference on Parallel and Distributed Systems), IEEE*, pages 151–158, 2000.

[CHvdS00b] D. Chkliaev, J. Hooman, and P. van der Stok. Mechanical verification of a non-blocking atomic commitment protocol. In *Proc. of DSVV'2000 (International Workshop on Distributed System Validation and Verification), IEEE*, pages E96–E103, 2000.

[CHvdS00c] D. Chkliaev, J. Hooman, and P. van der Stok. Mechanical verification of transaction processing systems. In *Proc. of ICFEM'2000 (3rd International Conference on Formal Engineering Methods), IEEE*, pages 89–97, 2000.

[CW96] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. Technical Report CMU-CS-96-178, 18 pages, 1996.

[Dam96] D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, 1996.

[Dat] *Database of Existing Mechanized Reasoning Systems, http://www-formal.stanford.edu/clt/ARS/systems.html*.

[DF93] R. Das and A. Fekete. Modular reasoning about open systems: A case study of distributed commit. In *Proceedings of Seventh International Workshop on Software Specification and Design*, pages 30–39, 1993.

[DGM97] M.C.A. Devillers, W.O.D. Griffioen, and O. Muller. Possibly infinite sequences: A comparative case study. In *TPHOLs'97*, pages 89–104. LNCS 1275, 1997.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[GA93] J.N. Gray and A.Reuter. *Transaction Processing Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[GHR97] R. Gupta, J. Haritsa, and K. Ramamritham. Revisiting commit processing in distributed database systems. In *Proc. of the ACM SIGMOD Int'l Conference on the Management of Data*, volume 26(2), pages 486–497, 1997.

[GMS97] J.F. Groote, F. Monin, and J. Springintveld. A computer checked algebraic verification of a distributed summing protocol. Computer Science Report 97-14, Department of Mathematics and Computer Science, Eindhoven, 1997.

[GMvdP98] J.F. Grote, F. Monin, and J. van de Pol. Checking verifications of protocols and distributed systems by computer. Technical Report 98-13, Eindhoven University of Technology, 28 pages, 1998.

[God96] P. Godefroid. *Partial Order Methods for the Verification of Concurrent Systems: An Approach to the State Space Explosion*. LNCS 1032, 1996.

[GP94] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes*, pages 26–62. Workshops in Computing, 1994.

[Gri00] D. Griffioen. *Computer Aided Verification of Protocols*. PhD thesis, University of Nijmegen, 2000.

[GS95] J.F. Groote and J. Springintveld. Focus points and convergent process operators. A proof strategy for protocol verification. Technical Report CS-R9566, CWI, Amsterdam, 1995.

[GvdP96] J.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large data packets. A case study in computer-checked verification. In *AMAST'96*, pages 536–550. LNCS 1101, 1996.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. In *Communications of the ACM*, volume 12(10), pages 576–583, 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hoa96] C.A.R. Hoare. How did software get so reliable without proof? In *Industrial benefit and advances in formal methods : international symposium*, pages 1–17. LNCS 1051, 1996.

[Hoo91] J. Hooman. *Specification and Compositional Verification of Real-Time Systems*. LNCS 558, 1991.

[Hoo97] J. Hooman. Verification of distributed real-time and fault-tolerant protocols. In *Sixth International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, pages 261–275. LNCS 1349, 1997.

[HSRT91] J. Huang, J.A. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proc of Very Large Databases*, pages 35–46, 1991.

[HSV94] L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. In *International Workshop TYPES'93*, pages 127–165. LNCS 806, 1994.

[Ip96] C.N. Ip. *State Reduction Methods for Automatic Formal Verification*. PhD thesis, Stanford University, 1996.

[J.C90] J.C.M. Baeten, editor. *Applications of process algebra*. Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press, 1990.

[JZ92] W. Janssen and J. Zwiers. Protocol design by layered decomposition: A compositional approach. In *Proc. FTRTFTS 2nd Int'l Symposium*, pages 307–326. LNCS 571, 1992.

[Kor94] H. Korver. *Protocol Verification in μCRL*. PhD thesis, University of Amsterdam, 1994.

[KS83] Z.M. Kedem and A. Silberschatz. Locking protocols: From exclusive to shared locks. *Journal of the ACM*, 30(4):787–804, 1983.

[KS92] R. Kurki-Suonio. Operational specification with joint actions: serializable databases. *Distributed Computing*, 6(1):19–38, 1992.

[KS94] H.P. Korver and J. Springintveld. A computer-checked verification of Milner's scheduler. In *International Symposium on Theoretical Aspects of Computer Software*, pages 161–178. LNCS 789, 1994.

[KST00] T. Kempster, C. Stirling, and P. Thanisch. Games-based model checking of protocols: counting doesn't count. In *Proc. of DSVV'2000 (International Workshop on Distributed System Validation and Verification), IEEE*, pages E111–E117, 2000.

[Kuo96] D. Kuo. Model and verification of a data manager based on ARIES. In *ACM Transactions on Database Systems*, volume 21(4), pages 427–479, 1996.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, 1989.

[Lam92] L. Lamport. Critique of the lake arrowhead three. *Distributed Computing*, 6(1):65–71, 1992.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[LL90] L. Lamport and N. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and semantics*, pages 1157–1199. Elsevier, 1990.

[LMWF94] N. Lynch, M. Merritt, W. Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, Inc., 1994.

[LS92] S.S. Lam and A.U. Shankar. Specifying modules to specify interfaces: a state transition system approach. *Distributed Computing*, 6(1):39–64, 1992.

[Lyn96] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.

[mCR] *μCRL: a language and tool set to study communicating processes with data, http://www.cwi.nl/ mcrl/.*

[Mil80] R. Milner. *A calculus of communicating systems*. LNCS 92, 1980.

[MP91] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: specification*. Springer Berlin, 1991.

[MP95] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer New York, 1995.

[Pap79] C.H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[Pap86] C.H. Papadimitriou. *The Theory of Concurrency Control*. Computer Science Press, 1986.

[PVS] *PVS Specification and Verification System, http://pvs.csl.sri.com/.*

[Rom99] J. Romijn. *Analysing Industrial Protocols with Formal Methods*. PhD thesis, University of Twente, 1999.

[Rus93] J. Rushby. Formal methods and the certification of critical systems. Technical Report CSL-93-7, 300 pages, 1993.

[RvH89] J. Rushby and F. von Henke. Formal verification of a fault-tolerant clock synchronization algorithm. NASA Contractor Report 4239, 1989.

[SAGG+93] J.F. Sogaard-Andersen, S.J. Garland, J.V. Gutlag, N.A. Lynch, and A. Pogosyants. Computer-assisted simulation proofs. In *Fifth Conference on Computer-Aided Verification (CAV'93)*, pages 305–319. LNCS 697, 1993.

[Sch96] A.A. Schoone. *Protocols by Invariants*. Cambridge International Series on Parallel Computation 7, Cambridge University Press, 1996.

[Sha93] N. Shankar. Verification of real-time systems using PVS. In *Fifth Conference on Computer-Aided Verification (CAV'93)*, pages 280–291. LNCS 697, 1993.

[SK80] A. Silberschatz and Z.M. Kedem. Consistency in hierarchical database systems. *Journal of the ACM*, 27(1):72–80, 1980.

[SKS97] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. The McGraw-Hill Companies, Inc., 1997.

[Spe99] D. Spelt. *Verification Support for Object Database Design*. PhD thesis, University of Twente, 1999.

[VH96] J. Vitt and J. Hooman. Assertional specification and verification using PVS of the steam boiler contol system. In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Contol*, pages 453–472. LNCS 1165, 1996.

[Vid91] K. Vidyasankar. Unified theory of database serializability. *Fundamenta Informaticae*, 14:147–183, 1991.

[vW95] J.J. van Wamel. *Verification Techniques for Elementary Data Types and Retransmission Protocols*. PhD thesis, University of Amsterdam, 1995.

Index

Summary

The thesis concerns the formal specification and mechanized verification of concurrency control and recovery protocols for distributed databases. Such protocols are needed for many modern application such as banking and are often used in safety-critical applications. Therefore it is very important to guarantee their correctness. One method to increase the confidence in the correctness of a protocol is its formal verification. In this thesis a number of important concurrency control and recovery protocols have been specified in the language of the verification system PVS. The interactive theorem prover of PVS has been used to verify their correctness.

In the first part of the thesis, the notions of conflict and view serializability have been formalized. A method to verify conflict serializability has been formulated in PVS and proved to be sound and complete with the proof checker of PVS. The method has been used to verify a few basic protocols. Next we present a systematic way to extend these protocols with new actions and control information. We show that if such an extension satisfies a few simple correctness conditions, the new protocol is serializable by construction.

In the existing literature, the protocols for concurrency control, single-site recovery and distributed recovery are often studied in isolation, making strong assumptions about each other. The problem of combining them in a formal way is largely ignored. To study the formal verification of combined protocols, we specify in the second part of the thesis a transaction processing system, integrating strict two-phase locking, undo/redo recovery and two-phase commit. In our method, the locking and undo/redo mechanism at distributed sites is defined by state machines, whereas the interaction between sites according to the two-phase commit protocol is specified by assertions. We proved with PVS that our system satisfies atomicity, durability and serializability properties.

The final part of the thesis presents the formal verification of atomic commitment protocols for distributed recovery. In particular, we consider the non-blocking protocol of Babaoglu and Toueg, combined with our own termination protocol for recovered participants. A new method to specify such protocols has been developed. In this method, timed state machines are used to specify the processes, whereas the communication mechanism between processes is defined using assertions. All safety and liveness properties, including a new improved termination property, have been proved with the interactive proof checker of PVS. We also show that the original termination protocol of Babaoglu and Toueg has an error.

Samenvatting

Dit proefschrift gaat over de formele specificatie en gemechaniseerde verificatie van concurrency control en recovery protocollen voor gedistribueerde databases. Zulke protocollen zijn nodig voor vele moderne toepassingen zoals bankieren en worden vaak gebruikt in safety-critical toepassingen. Daarom is het erg belangrijk hun correctheid te garanderen. Een methode om het vertrouwen in de correctheid van een protocol te vergroten is formele verificatie. In dit proefschrift zijn een aantal belangrijke concurrency control en recovery protocollen gespecificeerd in de taal van het verificatiesysteem PVS. De interactieve stellingenbewijzer van PVS is gebruikt om hun correctheid te verifiëren.

In het eerste gedeelte van het proefschrift, zijn de begrippen conflict en view serializability geformaliseerd. Een methode om conflict serializability te verifiëren is geformuleerd in PVS en geldig en volledig bewezen met de proof checker van PVS. De methode is gebruikt om enkele basisprotocollen te verifiëren. Vervolgens presenteren we een systematische manier om deze protocollen uit te breiden met nieuwe acties en besturingsinformatie. We laten zien dat, als zo'n uitbreiding aan een paar simpele correctheidscondities voldoet, het nieuwe protocol serializable is middels constructie.

In de bestaande literatuur, zijn de protocollen voor concurrency control, single-site recovery en gedistribueerde recovery vaak separaat bestudeerd, met sterke wederzijdse aannames. Het probleem om ze op een formele wijze te combineren wordt algemeen genegeerd. Om de formele verificatie van gecombineerde protocollen te bestuderen, specificeren we in het tweede deel van dit proefschrift een transactie verwerkings systeem, waarin strikt twee-fase locking, undo/redo recovery en twee-fase commit geïntegreerd zijn. In onze methode is het locking en undo/redo mechanisme op gedistribueerde lokaties gedefinieerd door toestandsmachines, terwijl de interactie tussen gedistribueerde processen volgens het twee-fase commit protocol wordt gespecificeerd door middel van asserties. We hebben met PVS bewezen dat ons systeem voldoet aan de eigenschappen van atomicity, durability en serializability.

Het laatste gedeelte van dit proefschrift presenteert de formele verificatie van atomic commitment protocols voor gedistribueerde recovery. In het bijzonder, bekijken we het non-blocking protocol van Babaoglu en Toueg, gecombineerd met ons eigen terminatieprotocol voor gecrashte processen die weer opgestart worden. Een nieuwe methode om zulke protocollen te specificeren is ontwikkeld. Volgens deze methode worden toestandsmachines met tijd gebruikt om de processen te specificeren, terwijl het communicatie mechanisme tussen processen wordt gedefiniëerd door middel van asserties. Alle veiligheids- en liveness eigenschappen, inclusief een nieuwe verbeterde terminatie-eigenschap, zijn bewezen met de interactieve stellingenbewijzer van PVS. We laten ook zien dat het originele terminatieprotocol van Babaoglu en Toueg een fout bevat.

Curriculum Vitae

Dmitri Chkliaev was born on May 6, 1975 in Novosibirsk, Russia. He grew up in Akademgorodok and attended the 25th Gymnasium. From 1991 to 1996, he studied at the Department of Mathematics and Mechanics of Novosibirsk State University, and obtained a Master's degree in Mathematics. From November 1996 to January 2001, he worked as a Ph.D. student at the Computing Science department of the Eindhoven University of Technology. From January 2001, he is employed as a researcher at the same department.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-1

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-2

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-3

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-4

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-5

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-6

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-7

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-8

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-9

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11