NASA Contractor Report 4239

Formal Verification of a Fault Tolerant Clock Synchronization Algorithm

John Rushby and Frieder von Henke
SRI International
Menlo Park, California

Prepared for
Langley Research Center
under Contract NAS1-17067

NASA
National Aeronautics and Space Administration
Office of Management
Scientific and Technical Information Division

1989

Abstract

We describe a formal specification and mechanically assisted verification of the Interactive Convergence Clock Synchronization Algorithm of Lamport and Melliar-Smith [11]. In the course of this work, we discovered several technical flaws in the analysis given by Lamport and Melliar-Smith, even though their presentation is unusually precise and detailed. As far as we know, these flaws (affecting the main theorem and four of its five lemmas) were not detected by the "social process" of informal peer scrutiny to which the paper has been subjected since its publication. We discuss the flaws in the published proof and give a revised presentation of the analysis that not only corrects the flaws in the original, but is also more precise and, we believe, easier to follow. This informal presentation was derived directly from our formal specification and verification. Some of our corrections to the flaws in the original require slight modifications to the assumptions underlying the algorithm and to the constraints on its parameters, and thus change the external specifications of the algorithm.

The formal analysis of the Interactive Convergence Clock Synchronization Algorithm was performed using our EHDM formal specification and verification environment. This application of EHDM provides a demonstration of some of the capabilities of the system.

Contents

1 Introduction **1**
1.1 Acknowledgments 3

2 Traditional Mathematical Presentation of the Algorithm and its Analysis **4**
2.1 Informal Overview 6
2.2 Statement of the Clock Synchronization Problem and Algorithm 10
2.3 Proof that the Algorithm maintains Synchronization 14
2.3.1 Overview of the Proof 14
2.3.2 The Proof in Detail 18
2.3.2.1 Constraints on Parameters 18
2.3.2.2 The Lemmas 19
2.3.2.3 The Correctness Theorem 26

3 Comparison with the Published Analysis by Lamport and Melliar-Smith **28**
3.1 The Definition of a Good Clock 28
3.2 Explicit Functional Dependencies 29
3.3 Approximations and Neglect of Small Quantities 30
3.3.1 A Flaw in the Main Induction 30
3.3.2 A Flaw in Lemma 4 31
3.4 The Interval in which a Clock is a "Good Clock" 32
3.4.1 Falsehood of Lemma 1 33
3.4.2 Falsehood of Lemma 2 34
3.5 Sundry Minor Flaws and Difficulties 34
3.5.1 Falsehood and Unnecessary Generality of Lemma 3 . . 34
3.5.2 Missing Requirements for Clock Synchronization Condition S2 34

3.5.3 Typographical Errors in Lemmas 2 and 4 35

4 Formal Specification and Verification in EHDM 36

4.1 Overview of EHDM 36
4.1.1 The Specification Language 37
4.1.1.1 Declarations 37
4.1.1.2 Modules 40
4.1.1.3 Proofs 41
4.1.1.4 Other Components of the EHDM System used in the Proof 45
4.2 The Formal Specification and Verification of the Algorithm . 48
4.2.1 Supporting Theories 49
4.2.1.1 Absolutes 49
4.2.1.2 Arithmetics 50
4.2.1.3 Natprops 52
4.2.1.4 Functionprops 52
4.2.1.5 Natinduction 53
4.2.1.6 Sums and Sigmaprops 54
4.2.2 Specification Modules 56
4.2.2.1 Time 56
4.2.2.2 Clocks 57
4.2.2.3 Algorithm 57
4.2.3 Proof Modules 58
4.2.3.1 Clockprops 58
4.2.3.2 Lemmas 1 to 6 59
4.2.3.3 Summations 59
4.2.3.4 Juggle 59
4.2.3.5 Main 60
4.3 Statistics and Observations 60

5 Conclusions 68

Bibliography 73

A Cross-Reference Listing 75

B LaTeX-printed Specification Listings 90

C Proof-Chain Analysis **145**
C.1 Clock Synchronization Condition S2 145
C.2 Clock Synchronization Condition S1 159

D Plain EHDM Specification Listings **162**

List of Figures

2.1 Statements of the Principal Lemmas used in The Proof . . . 16

4.1 An Example EHDM Specification Module 42
4.2 LaTeX-printed Example EHDM Specification Module 47

List of Tables

2.1 Notation, Parameters, and Concepts 15
2.2 Typical Values for the Parameters 17

4.1 Proof Summaries for EHDM Modules 61

A.1 LaTeX-Printer Translations for EHDM Identifiers 76
A.2 Cross-Reference to EHDM Identifiers 77

B.1 Page References to EHDM Specification Modules 91

D.1 Page References to raw EHDM Specification Modules 163

Chapter 1

Introduction

The Interactive Convergence Clock Synchronization Algorithm is an important and fairly difficult algorithm. It is important because the synchronization of clocks is fundamental to the fault tolerance mechanisms employed in critical process control systems such as fly-by-wire digital avionics. It is difficult because its analysis must consider the relationships among quantities (i.e., clock values) that are continually changing—and changing moreover at slightly different rates—and because it must deal with the possibility that some of the clocks may be faulty and may exhibit arbitrary behavior. Thus, although the algorithm is easy to describe and a broad understanding of why it works can be obtained fairly readily, its rigorous analysis, and the derivation of bounds on the synchronization that it can achieve, require attention to a mass of detail and very careful explication of assumptions.

Lamport and Melliar-Smith's paper [11] is a landmark in the field. They not only introduced the Interactive Convergence Clock Synchronization Algorithm, but two other algorithms as well, and they also developed formalizations of the assumptions and desired properties that made it possible to give a precise statement and proof for the correctness of clock synchronization algorithms. Nonetheless, the proof given by Lamport and Melliar-Smith is hard to internalize: there is much detailed argument, some involving approximate arithmetic and neglect of insignificant terms, and it is not easy to convince oneself that all the details mesh correctly. It is precisely in performing conceptually simple, but highly detailed arguments (i.e., *calculations*) that the human mind seems most fallible, and machines most effective. Consequently, the Interactive Convergence Clock Synchronization Algorithm seems an excellent candidate for mechanical verification. This report describes a mechanized proof of the correctness of the algorithm using the EHDM formal specification and verification environment.

As we performed the formal specification and verification of the Interactive Convergence Clock Synchronization Algorithm, we discovered that the presentation given by Lamport and Melliar-Smith was flawed in several details. One of the principal sources of error and difficulty was the use by Lamport and Melliar-Smith of approximations—i.e., approximate equality ($\approx$) and inequalities ($\lesssim$ and $\gtrsim$)—in order to "simplify the calculations." We eventually found that elimination of the approximations not only removed one class of errors, but actually simplified the analysis and presentation. We also found and corrected several other technical flaws in the published proof of Lamport and Melliar-Smith. A discussion of these flaws is given in Chapter 3. Some of our corrections require slight modifications to the assumptions underlying the algorithm, and to the constraints on its parameters, and thus change the external specifications of the algorithm. Our formal specification and verification of the algorithm is described in Chapter 4; the detailed listings are to be found in the Appendices.

We discuss the lessons learned from this exercise, and our view of the role and utility of formal specification and verification in Chapter 5. To summarize those conclusions: we now believe the Interactive Convergence Clock Synchronization Algorithm to be correct, not because our theorem prover says it is, but because the experience of arguing with the theorem prover has forced us to clarify our assumptions and proofs to the point where we think we really *understand* the algorithm and its analysis. As a result, we can present an argument for the correctness of the algorithm, in the style of a traditional mathematical presentation, that we believe is truly compelling. This presentation is given in Chapter 2 and follows very closely the presentation given in Sections 2.1, 3, and 4 of the original paper [11, pages 53–66]. However, the details of the proof were extracted directly from our formal verification.

It is this traditional mathematical presentation of our revised proof of correctness for the Interactive Convergence Clock Synchronization Algorithm that we consider the main contribution of this work; we hope that anyone contemplating using the algorithm will study our presentation and will convince *themselves* of the correctness of the algorithm and of the appropriateness of the assumptions (and of the ability of their implementation to satisfy those assumptions). We stress that our presentation merely dots the i's and crosses some important t's in the original; the substance of all the arguments is due to Lamport and Melliar-Smith. Those already familiar with the original presentation should probably read Chapter 3 before Chapter 2. (Indeed, they may then want to skip Chapter 2 altogether.)

1.1 Acknowledgments

This work was performed for the National Aeronautics and Space Administration under contract NAS1 17067 (Task 4). The guidance and advice provided by our technical monitor, Ricky Butler of NASA Langley Research Center, was extremely valuable. We owe an obvious debt to Leslie Lamport and Michael Melliar-Smith, who not only invented the algorithm studied here, but also developed the formalization and analysis that is the basis for our mechanically-assisted verification. Leslie Lamport also provided helpful comments on an earlier version of this report.

Chapter 2

Traditional Mathematical Presentation of the Algorithm and its Analysis

Many distributed systems depend upon a common notion of time that is shared by all components. Usually, each component contains a reasonably accurate clock and these clocks are initially synchronized to some common value. Because the clocks may not all run at precisely the same rate, they will gradually drift apart and it will be necessary to resynchronize them periodically. In a fault-tolerant system, this resynchronization must be robust even if some clocks are faulty: the presence of faulty clocks should not prevent those components with good clocks from synchronizing correctly.

The design, and especially the analysis, of fault-tolerant clock synchronization algorithms is a surprisingly difficult endeavor, especially if one admits the possibility of "two-faced" clocks and other so-called Byzantine faults.

Consider a system with three components: A, B, and C; A and C have good clocks, but B's clock is faulty. A's clock indicates 2.00 pm, C's 2.01 pm, and B's clock indicates 1:58 pm to A but 2.03 pm to C. A sees that C's clock is ahead of its own, and that B's is behind by a somewhat greater amount; it would be natural therefore for A to set its own clock back a little. This situation is reversed, however, when considered from C's perspective. C sees that A's clock is a little behind its own and that B's is ahead by a rather greater amount; it will be natural for C to set its own clock *forward* a little. Thus the faulty clock B has the effect of driving the good clocks

A and *C* further apart. The behavior of *B*'s clock that produces this effect may seem actively malicious and therefore implausible. This is not so, however. A failed clock may plausibly act as a random number generator (noisy diodes are indeed used as hardware random number generators) and could thereby distribute very different values to different components in response to inquiries received very close together. Of course, one can postulate a design in which a single clock value is latched and then distributed to all other components—but then one must provide compelling evidence for the correctness of the latching mechanism and the impossibility of cummunication errors, and for the correctness of a clock synchronization algorithm built on these assumptions.

Accurate clock synchronization is one of the fundamental requirements for fault-tolerant real-time control systems, such as flight-critical digital avionics. These systems use replicated processors in order to tolerate hardware faults; several processors perform each computation and the results are subjected to majority voting. It is vital to this process that the replicated processors keep in step with each other so that voting is performed on computations belonging to the same "frame." Since synchronization of processors' clocks is essential for the fault-tolerance provided by this approach, it is clear that the clock synchronization process must itself be exceptionally fault-tolerant. In particular, it should make only very robust assumptions about the behavior of faulty processors' clocks.

The strongest clock synchronization algorithms make no assumptions whatever about the behavior of faulty clocks. Lamport and Melliar-Smith [11] describe three such fault-tolerant clock synchronization algorithms. These algorithms work in the presence of any kind of fault—including malicious two-faced clocks such as that described above. Of course, there must not be *too many* faulty clocks. The first algorithm presented by Lamport and Melliar-Smith, the *Interactive Convergence Algorithm*, can tolerate up to m faults amongst $3m + 1$ clocks. Thus, 4 clocks are required to guarantee the ability to withstand a single fault. Dolev, Halpern and Strong have shown that $3m + 1$ clocks are *required* to allow synchronization in the presence of m faults unless digital signatures are used [8]. Thus, the Interactive Convergence algorithm requires the minimum possible number of clocks for its class of algorithms.

The Interactive Convergence Clock Synchronization Algorithm is quite easy to describe in broad outline: periodically, each processor reads the differences between its clock and those of all other processors, replaces those differences that are "too large" by zero, computes the average of the resulting values, and adjusts its clock by that amount. For descriptions of other clock synchronization algorithms, presented in a consistent notation, see the surveys by Butler [4] (which includes hardware techniques) and Schneider [15]. A new class of probabilistic clock synchronization algorithms that have extremely good performance (in terms of how close the clocks can be synchronized) has recently been introduced by Cristian [6], but so far the algorithms in this class are not tolerant of Byzantine failures.

In the next section we give an informal overview of the analysis of the Interactive Convergence Clock Synchronization Algorithm. This should support the reader's intuition during the more formal analysis in the section that follows. Although "formal" in the sense of traditional mathematical presentations, this level of analysis is not truly formal (in the sense of being based on an explicit set of axioms and rules of inference)—that level of presentation is described in Chapter 4 and its supporting Appendices.

2.1 Informal Overview

We assume a number of components (generally called "processors") each having its own clock. Nonfaulty clocks all run at approximately the correct rate and are assumed to be approximately synchronized initially. Due to the slight differences in their running rates, the clocks will gradually drift apart and must be resynchronized periodically. We are concerned with the problem of performing this resynchronization; we are not concerned with the problem of maintaining the clocks in synchrony with some external "objective" time (see Lamport [12] for a discussion of this problem), nor are we concerned with the problem of synchronizing the clocks initially, although the closeness with which the initial synchronization is performed will limit how closely the clocks can be brought together in subsequent resynchronizations.[1]

The goal of periodic resynchronizations is to ensure that all nonfaulty clocks have approximately the same value at any time. A secondary goal is to accomplish this without requiring excessively large adjustments to the value of any clock during the synchronization process. Formalizing these two goals and the assumptions identified earlier is one of the major steps in the verification of the Interactive Convergence Clock Synchronization Algorithm. For future convenience, we label and explicitly identify them

[1]The initial synchronization establishes a bound that cannot be bettered in the worst-case; in practice subsequent resynchronizations may improve on the initial synchronization.

here (using the same names as [11]), and give them the following informal characterizations:

Requirements

S1: At any time, the values of all the nonfaulty processors' clocks must be approximately equal. (The maximum skew between any two good clocks is denoted by δ.[2])

S2: There should be a small bound (denoted Σ) on the amount by which a nonfaulty processor's clock is changed during each resynchronization. (When taken with A1 below, this requirement rules out trivial solutions that merely set the clocks to some fixed value.)

Assumptions

A0: All clocks are initially synchronized to approximately the same value. (The maximum initial skew is denoted δ_0.)

A1: All nonfaulty processors' clocks run at approximately the correct rate. (The maximum drift is a parameter denoted by ρ.)

Schneider [15] shows that all Byzantine clock synchronization algorithms can be viewed as different refinements of a single paradigm: periodically, the processors decide that it is time to resynchronize their clocks, each processor reads the clocks of the other processors, forms a "fault tolerant average" of their values, and sets its own clock to that value. There are three main elements to this paradigm:

1. Each processor must be able to tell when it is time to resynchronize its clock with those of other processors,
2. Each processor must have some way of reading the clocks of other processors,
3. There must be a *convergence function* which each processor uses to form the "fault tolerant average" of clock values.

In the Interactive Convergence Clock Synchronization Algorithm, each processor performs a constant round of activity, executing a series of tasks

[2] A summary of the notation and definitions used is given in Table 2.1 on Page 15.

over and over again. Each iteration of this series of tasks consumes an interval of time called a *period*. All periods are supposed to be of the same duration, denoted by R. The final task in each period, occupying an interval of time denoted by S, is the clock synchronization task. Each processor uses its own clock to schedule the tasks performed during each period. Thus, each processor relies on its own clock to trigger the clock synchronization task; because the nonfaulty clocks were resynchronized during the previous synchronization task and cannot have drifted too far apart since then, all processors with nonfaulty clocks will enter their clock synchronization tasks at approximately the same time.

During its clock synchronization task, each processor reads the clock of every other processor. Of course, clock values are constantly changing and go "stale" if a long (or indeterminate) amount of time goes by between them being read and being used. For this reason, it is much more useful for each processor to record the *difference* between its clock and that of other processors. The closeness of the synchronization that can be accomplished is strongly influenced by how accurately these clock differences can be read. This gives rise to the third assumption required by the Interactive Convergence Clock Synchronization Algorithm:

Assumption

A2: A nonfaulty processor can read the difference between its own clock and that of another nonfaulty processor with at most a small error. (The upper bound on this error is a parameter denoted by ϵ).

The remaining element that is needed to characterize the Interactive Convergence Clock Synchronization Algorithm is the definition of its convergence function. As suggested above, each processor should set its clock to a "fault tolerant average" of the clock values from all the processors. The obvious "average" value to use is the arithmetic mean, but this will not have the desired fault tolerance property if faulty processors inject wildly erroneous values into the process. A simple remedy is for each processor to use its own clock value in place of those values that differ by "too much" from its own value. This function, called the "egocentric mean," is the convergence function used in the Interactive Convergence Clock Synchronization Algorithm. The parameter that determines when clock differences are "too large" is denoted Δ.

To gain an idea of why this works, consider two nonfaulty processors p and q. For simplicity, assume that these processors perform their synchronization calculations simultaneously and instantaneously. If r is also a nonfaulty processor, then the estimates that p and q form of r's clock value can differ by at most 2ϵ. If r is a faulty processor, however, p and q could form estimates of its clock value that differ by as much as $2\Delta + \delta$. (Since r could indicate a value as large as Δ different from each of p and q without being disregarded, and these processors could themselves have clocks that are δ apart.) Assuming there are n processors, of which m are faulty, the egocentric means formed by p and q can therefore differ from each other by as much as

$$\frac{2(n-m)\epsilon + m(\delta + 2\Delta)}{n}.$$

Thus, provided

$$\delta \geq 2\epsilon + \frac{2m\Delta}{n-m}, \tag{2.1}$$

this procedure will maintain the clocks of p and q within δ of each other, as required.

Since a nonfaulty processor's clock can differ from another's by as much as δ, and reading its value can introduce a further error of ϵ, it is clear that we must require

$$\Delta \geq \delta + \epsilon,$$

since otherwise perfectly good clock values could be disregarded. This gives

$$\Delta - \epsilon \geq \delta$$

which, when taken with (2.1), yields

$$3\epsilon \leq \frac{n-3m}{n-m}\Delta. \tag{2.2}$$

Because all the variables involved are strictly positive (except m, which is merely nonnegative), (2.2) implies

$$n > 3m,$$

showing that four clocks are required to tolerate a single failure. (Notice that seven clocks are required to withstand two *simultaneous* failures. However, if each failure can be detected and the system reconfigured before another failure occurs, then five clocks can withstand two failures.)

Lamport and Melliar-Smith raise a couple of fine points that should be considered in implementation and application of the Interactive Convergence

Clock Synchronization Algorithm. The correction that occurs at each synchronization causes a discontinuity in clock values. If a correction is positive (because the clock has been running slow), then some units of clock time will vanish in the discontinuity as the correction is applied. Any task scheduled to start in the vanished interval might not occur at all. Conversely, a negative correction (for a fast clock), can cause units of clock time to repeat, possibly causing a task to be executed a second time. One solution to these difficulties is to follow each clock synchronization with a "do nothing" task of duration at least Σ. An alternative, that has other attractive properties, is to avoid the discontinuity altogether and spread the application of the correction evenly over the whole period [11, pages 54–55].

2.2 Statement of the Clock Synchronization Problem and Algorithm

The informal argument presented above did not account for the fact that the clocks may drift further apart in the period between synchronizations, nor did it allow for the facts that the algorithm takes time to perform, and that different processors will start it at slightly different times. Taking care of these details, and being precise about the assumptions employed, is the task of the more detailed argument presented in this section.

The first step is to formalize what is meant by a clock, and what it means for a clock to run at approximately the correct rate.

Physically, a clock is a counter that is incremented periodically by a crystal or line-frequency oscillator. By a suitable linear transformation, the counter value is converted to a representation of conventional "time" (e.g., the number of seconds that have elapsed since January 1st, 1960, Coordinated Universal Time). This internal estimation of time may be expected to drift somewhat from the external, standard record of time maintained by international bodies. In order to distinguish these two notions of time, we will describe the internal estimate of time that may be read from a processor's clock as *clock time*, and the external notion of time (that may not be directly observable) as *real time*. Following Lamport and Melliar-Smith, we use lowercase letters to denote quantities that represent real time, and upper case for quantities that represent clock time. Thus, "second" denotes the unit of real time, while "SECOND" denotes the unit of clock time. Within this convention, Roman letters are used to denote "large" values (on the order of tens of milliseconds), while Greek letters are used to denote "small" values (on the order of tens of microseconds).

We are interested in process control applications where events are triggered by the passage of clock time—e.g., "start the furnace at 9 AM and stop it at 5 PM," or "run the clock synchronization task every 5 SECONDS." Our notion of synchronization is that activities scheduled for the same clock time in different processors should actually occur very close together in real time.[3] Thus, we define a clock c to be a mapping from clock time to real time: $c(T)$ denotes the real time at which clock c reads T. Two clocks c and c' are said to be ***synchronized*** to within real time δ at clock time T if they reach the value T within δ seconds of each other—i.e., if $|c(T) - c'(T)| < \delta$. The real time quantity $|c(T) - c'(T)|$ is called the ***skew*** between c and c' at clock time T. Another measure of the divergence between these two clocks is the adjustment that one of them should make in order to reduce the skew to zero. The clock time quantity Φ such that $c(T + \Phi) = c'(T)$ is called c's ***adjustment*** to c' (at time T).

A clock is a "good clock" if it runs at a rate very close to the passage of real time. Lamport and Melliar-Smith define this formally in terms of the derivative of the clock function. However, since we will be using a mechanical verification system, and do not want to have to axiomatize a fragment of the differential calculus, we use a slightly different formulation taken from Butler [4].

Definition 1: *A clock c is a* good clock *during the clock time interval $[T_0, T_N]$ if*

$$\left|\frac{c(T_1) - c(T_2)}{T_1 - T_2} - 1\right| < \frac{\rho}{2}.$$

whenever T_1 and T_2 ($T_1 \neq T_2$) are clock times in $[T_0, T_N]$.

Clocks are resynchronized every R SECONDS. We assume some starting time T^0, define $T^{(i)} = T^0 + iR$ ($i \geq 0$), and let $R^{(i)}$ denote the interval $[T^{(i)}, T^{(i+1)}]$, which we call the i'th ***period***. The actual synchronization task is executed during the final S SECONDS of each period: all reading and transmitting of clock values occurs within the interval $[T^{(i+1)} - S, T^{(i+1)}]$, which we call the i'th ***synchronizing period*** and denote by $S^{(i)}$.

[3]For other classes of applications, the reverse notion may be more appropriate—e.g., if a single event is to be given (clock time) timestamps by different processors, then we may want the different timestamps (all triggered at the same real time) to be very close together. Lamport and Melliar-Smith [11, page 61] indicate how to convert between this notion of synchronization and the one used here.

We consider a set of n processors, where processor p has clock c_p. Clocks are adjusted by adding a "correction" to their values; the correction used by processor p during the i'th period is denoted $C_p^{(i)}$, so that the real time corresponding to clock time T on processor p during period i is $c_p(T+C_p^{(i)})$. We denote this quantity by $c_p^{(i)}(T)$ and we call $c_p^{(i)}$ the *logical* clock for processor p during the i'th period. We call $T+C_p^{(i)}$ the *adjusted* value of T for processor p in period i and denote it by $A_p^{(i)}(T)$ (so that $c_p^{(i)}(T) = c_p(A_p^{(i)}(T))$). For simplicity, we assume that the initial correction $C_p^{(0)} = 0$.

The *skew* between the clocks of processors p and q at time T in $R^{(i)}$ is given by

$$|c_p^{(i)}(T) - c_q^{(i)}(T)|.$$

The goal of the Interactive Convergence Clock Synchronization Algorithm is to bound this quantity for good clocks. We assume that all the clocks are synchronized within δ_0 of each other at the "starting time" $T^{(0)}$:

A0: For all processors p and q, $|c_p^{(0)}(T^{(0)}) - c_q^{(0)}(T^{(0)})| < \delta_0$.

The process control applications that are of interest to us typically perform a schedule of many separate tasks during each period. Our goal is to ensure that tasks which are scheduled to occur on different processors at the same clock time during a particular period actually occur very close to each other in real time. To achieve this, processor p should perform a task scheduled for time T in the i'th period at the instant its clock actually reads $A_p^{(i)}(T)$.[4] An obvious consequence is that the i'th period for processor p runs from when its *adjusted* clock reads $T^{(i)}$ until it reads $T^{(i+1)}$. That is, it is the clock time interval $[A_p^{(i)}(T^{(i)}), A_p^{(i)}(T^{(i+1)})]$. Therefore, if a processor's clock is to work long enough to complete the i'th period, it must be a good clock throughout the interval $[A_p^{(0)}(T^{(0)}), A_p^{(i)}(T^{(i+1)})]$. This motivates the following definition of what it means for a processor to be nonfaulty:

A1: We say that a processor is *nonfaulty* through period i if its clock is a good clock in the clock time interval $[A_p^{(0)}(T^{(0)}), A_p^{(i)}(T^{(i+1)})]$.

[4]To see this, consider a processor whose clock gains one SECOND every hour and whose periods are of one HOUR duration. A task to be performed 5 MINUTES into period 3 should be started when the *adjusted* time reads 3 hours and 5 minutes from the initial time. The correction during period 3 will be -3 SECONDS, so that the task will be started when the clock actually reads 3 hours, 5 minutes and 3 seconds from the initial time. It can be seen that this is indeed the desired behavior.

There is another assumption about nonfaulty processors, which is not formalized and is not considered further during the analysis: this is the assumption that nonfaulty processors perform the algorithm correctly.

Now we can state formally the goals that the Interactive Convergence Clock Synchronization Algorithm is to satisfy.

Clock Synchronization Conditions: For all processors p and q, if all but at most m processors (out of n) are nonfaulty through period i, then

S1: If p and q are nonfaulty through period i, then for all T in $R^{(i)}$

$$|c_p^{(i)}(T) - c_q^{(i)}(T)| < \delta.$$

S2: If processor p is nonfaulty through period i, then

$$|C_p^{(i+1)} - C_p^{(i)}| < \Sigma.$$

We now formalize Assumption A2 concerning the reading of clocks. The idea is that sometime during the i'th synchronizing period, processor p should obtain a value that indicates the difference between its own clock and that of another processor q. To synchronize exactly with q at some time T' in $S^{(i)}$, p would need to know the ideal adjustment $\Phi_{qp}^{(i)}$ that it should add to its own value so that $c_p^{(i)}(T' + \Phi_{qp}^{(i)}) = c_q^{(i)}(T')$. In practice, p cannot obtain this value exactly, instead, it obtains an approximation $\Delta_{qp}^{(i)}$ that is subject to a small error ϵ. The formal statement is given below.

A2: If conditions S1 and S2 hold for the i'th period, and processor p is nonfaulty through period i, then for each other processor q, p obtains a value $\Delta_{qp}^{(i)}$ during the synchronization period $S^{(i)}$. If q is also nonfaulty through period i, then

$$|\Delta_{qp}^{(i)}| \leq S$$

and

$$|c_p^{(i)}(T' + \Delta_{qp}^{(i)}) - c_q^{(i)}(T')| < \epsilon$$

for some time T' in $S^{(i)}$.

If $p = q$, we take $\Delta_{qp}^{(i)} = 0$ so that A2 holds in this case also. Notice that A2 requires S1 and S2 to hold in the period concerned. This is because the method by which processors read the differences between their clocks may require them to cooperate—which may in turn depend upon their clocks already being adequately synchronized.

Finally, we can give a formal description of the Interactive Convergence Clock Synchronization Algorithm (in the following also referred to as "the Algorithm" for short).

Algorithm CNV: For all processors p:

$$C_p^{(i+1)} = C_p^{(i)} + \Delta_p^{(i)},$$

where

$$\begin{aligned} \Delta_p^{(i)} &= \left(\frac{1}{n}\right)\sum_{r=1}^{n} \bar{\Delta}_{rp}^{(i)}, \quad \text{and} \\ \bar{\Delta}_{rp}^{(i)} &= \textbf{if } |\Delta_{rp}^{(i)}| < \Delta \textbf{ then } \Delta_{rp}^{(i)} \textbf{ else } 0. \end{aligned}$$

A summary of the notation and definitions introduced so far is given in Table 2.1 on Page 15. Some typical values for the parameters, based on an experimental validation using the SIFT computer [5], are given in Table 2.2 on Page 17.

2.3 Proof that the Algorithm maintains Synchronization

We now need to prove that the Interactive Convergence Clock Synchronization Algorithm maintains the clock synchronization conditions S1 and S2. Condition S2 is easy; the difficult part of the proof is to show that the Algorithm maintains Condition S1. The proof is an induction on i—we show that if the clocks are synchronized through period i, and if sufficient processors remain nonfaulty through period $i+1$, then the nonfaulty processors will remain synchronized through that next period. The actual proof is a mass of details, so it will be helpful to sketch the basic approach first. For reference, the statements of the main Lemmas are collected in Figure 2.1.

2.3.1 Overview of the Proof

We are interested in the skew between two nonfaulty processors during the $i+1$'st period—that is, in the quantity

$$|c_p^{(i+1)}(T) - c_q^{(i+1)}(T)|$$

Symbol	Concept
n	number of clocks
m	number of faulty clocks
R	clock time between synchronizations
S	clock time to perform synchronization algorithm
$T^{(i)}$	clock time at start of i'th period $(= T^{(0)} + iR)$
$R^{(i)}$	i'th period $(= [T^{(i)}, T^{(i+1)}])$
$S^{(i)}$	i'th synchronizing interval $(= [T^{(i+1)} - S, T^{(i+1)}])$
$C_p^{(i)}$	cumulative correction for p's clock in i'th period
$A_p^{(i)}(T)$	adjusted value of T for p's clock in i'th period $(= T + C_p^{(i)})$
$c_p(T)$	real time when p's clock reads T
$c_p^{(i)}(T)$	real time in i'th period, when p's clock reads T $(= c_p(Ap^{(i)}(T)))$
δ	maximum real time skew between any two good clocks
δ_0	maximum initial real time skew between any two clocks
ϵ	maximum real time clock read error
ρ	maximum clock drift rate
$\Delta_{qp}^{(i)}$	clock time difference between q and p seen by p in i'th period
Δ	cut off for $\Delta_{qp}^{(i)}$
$\bar{\Delta}_{qp}^{(i)}$	**if** $\lvert\Delta_{qp}^{(i)}\rvert < \Delta$ **then** $\Delta_{qp}^{(i)}$ **else** 0
$\Delta_p^{(i)}$	clock time correction made by p in i'th period (mean of $\bar{\Delta}^{i}_{qp}$'s)
Σ	maximum correction permitted

Table 2.1: Notation, Parameters, and Concepts

Lemma 1: *If the clock synchronization conditions S1 and S2 hold for i, and processors p and q are nonfaulty through period $i+1$, then*

$$|\Delta_{qp}^{(i)}| < \Delta.$$

Lemma 2: *If processor p is nonfaulty through period $i+1$, and T and Π are such that $A_p^{(i)}(T)$ and $A_p^{(i)}(T+\Pi)$ are both in the interval $[A_p^{(0)}(T^{(0)}), A_p^{(i+1)}(T^{(i+2)})]$, then*

$$|c_p^{(i)}(T+\Pi) - [c_p^{(i)}(T) + \Pi]| < \frac{\rho}{2}|\Pi|.$$

Lemma 3: *If the clock synchronization conditions S1 and S2 hold for i, processors p and q are nonfaulty through period $i+1$, and $T \in S^{(i)}$, then*

$$|c_p^{(i)}(T + \Delta_{qp}^{(i)}) - c_q^{(i)}(T)| < \epsilon + \rho S.$$

Lemma 4: *If the clock synchronization conditions S1 and S2 hold for i, processors p, q, and r are nonfaulty through period $i+1$, and $T \in S^{(i)}$, then*

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)}]| < 2(\epsilon + \rho S) + \rho\Delta.$$

Lemma 5: *If the clock synchronization condition S1 holds for i, processors p and q are nonfaulty through period $i+1$, and $T \in S^{(i)}$, then*

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)}]| < \delta + 2\Delta.$$

Figure 2.1: Statements of the Principal Lemmas used in The Proof

Parameter	Value
n	6
R	104.8 msec.
S	3.2 msec
δ_0	132 μsec. (typically, 10 μsec. is achieved)
ϵ	66.1 μsec. (typically, better than 15 μsec. is achieved)
ρ	15×10^{-6}
Δ	340 μsec.
Σ	340 μsec.
δ	134 μsec. $(m = 0)$, 271 μsec. $(m = 1)$

Table 2.2: Typical Values for the Parameters

where $T \in R^{(i+1)}$. By the Algorithm,

$$|c_p^{(i+1)}(T) - c_q^{(i+1)}(T)| = |c_p^{(i)}(T + \Delta_p^{(i)}) - c_q^{(i)}(T + \Delta_q^{(i)})|, \qquad (2.3)$$

and since good clocks run at approximately the correct rate, $c_p^{(i)}(T + \Delta_p^{(i)})$ and $c_q^{(i)}(T+\Delta_q^{(i)})$ are close to $c_p^{(i)}(T)+\Delta_p^{(i)}$ and to $c_q^{(i)}(T)+\Delta_q^{(i)}$, respectively. From this it follows that the right hand side of (2.3) can be approximated by

$$|c_p^{(i)}(T) + \Delta_p^{(i)} - [c_q^{(i)}(T) + \Delta_q^{(i)}]|.$$

A major step in the proof, identified as Lemma 2, is concerned with bounding the error introduced by this approximation. Then, since $\Delta_p^{(i)}$ and $\Delta_q^{(i)}$ are the averages of $\bar{\Delta}_{rp}^{(i)}$ and $\bar{\Delta}_{rq}^{(i)}$, it is natural to consider the individual components

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)}]|. \qquad (2.4)$$

There are two cases to consider. The first, in which only p and q are assumed nonfaulty, is the focus of Lemma 5, while the second, in which r is also assumed nonfaulty, is considered in Lemma 4. The first case is quite easy—the Algorithm ensures that $\bar{\Delta}_{rp}^{(i)}$ and $\bar{\Delta}_{rq}^{(i)}$ can be no larger than Δ, while $c_p^{(i)}(T)$ and $c_q^{(i)}(T)$ can differ by no more than δ (by the inductive hypothesis). For the second case, Lemma 1 provides the result $|\Delta_{rp}^{(i)}| < \Delta$, so that the Algorithm will establish $\bar{\Delta}_{rp}^{(i)} = \Delta_{rp}^{(i)}$ and $\bar{\Delta}_{rq}^{(i)} = \Delta_{rq}^{(i)}$. The quantity (2.4) is then rewritten as

$$|c_p^{(i)}(T) + \Delta_{rp}^{(i)} - c_r^{(i)}(T) - [c_q^{(i)}(T) + \Delta_{rq}^{(i)} - c_r^{(i)}(T)]|.$$

Regarding this as the absolute difference of two similar expressions, we are led to consider values of the form

$$|c_p^{(i)}(T) + \Delta_{rp}^{(i)} - c_r^{(i)}(T)|$$

which, using Lemma 2, can be approximated by

$$|c_p^{(i)}(T + \Delta_{rp}^{(i)}) - c_r^{(i)}(T)|.$$

Lemma 3 is concerned with quantities of this form.

2.3.2 The Proof in Detail

We now prove that the Interactive Convergence Clock Synchronization Algorithm maintains the clock synchronization conditions S1 and S2. The proof closely follows that of Lamport and Melliar-Smith [11, pages 64–66] (though we do separate the two synchronization conditions and prove them individually as Theorems 1 and 2, respectively). In particular, our Lemmas 1–5 correspond exactly to (corrected versions of) theirs. However, since we use Lemma 2 in the proof of Lemma 1, we rearrange the order of presentation accordingly. We also introduce a Lemma 6 and a Sublemma A that is used in its proof and also in the base case of the inductive proof of condition S1. Lamport and Melliar-Smith subsumed both of these in the proof of their main theorem. In addition, we distinguish several special cases for Lemma 2, which we identify as Lemmas 2a–2d. (Lemma 2c is the one that corresponds most closely to Lemma 2 in [11].) The reasons for these additional lemmas are: first, we describe the proof in greater detail than did Lamport and Melliar-Smith; secondly, the statements of some of our lemmas are more restrictive than those of Lamport and Melliar-Smith (that is why we need several variants of Lemma 2—the single Lemma 2 stated by Lamport and Melliar-Smith is false); thirdly, this presentation of the proof exactly follows the structure of the formal verification described in Chapter 4 and presented in detail in the Appendices.

In the remainder of this section we state and prove the lemmas identified above, followed by the main theorems. First, however, we state some constraints on parameters that are employed in several of the proofs.

2.3.2.1 Constraints on Parameters

Our proofs are contingent on the parameters to the Algorithm ($n, m, R, S, \Sigma, \Delta, \epsilon, \delta, \delta_0$ and ρ) satisfying certain constraints. We could mention these constraints explicitly in the statements of the lemmas and of the theorems, but that would be tedious and would clutter those statements needlessly. Accordingly we list and name here the six constraints that the parameters are required to satisfy. Satisfaction of these constraints is assumed throughout the proof.

The first two constraints can be modified (but not eliminated) if necessary by suitably adjusting some of the proofs; we chose these particular constraints for simplicity and because we felt that there would be no difficulty satisfying them in any likely implementation. The other four constraints are fundamental to the operation and analysis of the Algorithm.

C1: $R \geq 3S$

C2: $S \geq \Sigma$

C3: $\Sigma \geq \Delta$

C4: $\Delta \geq \delta + \epsilon + \frac{\rho}{2} S$

C5: $\delta \geq \delta_0 + \rho R$

C6: $\delta \geq 2(\epsilon + \rho S) + \dfrac{2m\Delta}{n-m} + \dfrac{n\rho R}{n-m} + \dfrac{n\rho\Sigma}{n-m} + \rho\Delta$

The reader may wonder why we do not include the celebrated constraint $3m < n$. The reason is simply that this is a derived constraint, not a fundamental one. It is easy to see that C4 and C6 can be satisfied simultaneously only if indeed $3m < n$, but it is also quite possible for values of other parameters to render C4 or C6 unsatisfiable even if $3m < n$.

2.3.2.2 The Lemmas

Lemma 2: *If processor p is nonfaulty through period $i+1$, and T and Π are such that $A_p^{(i)}(T)$ and $A_p^{(i)}(T+\Pi)$ are both in the interval $[A_p^{(0)}(T^{(0)}), A_p^{(i+1)}(T^{(i+2)})]$, then*

$$|c_p^{(i)}(T+\Pi) - [c_p^{(i)}(T) + \Pi]| < \frac{\rho}{2}\,|\Pi|.$$

Proof: Since p is nonfaulty through period $i+1$, we know by A1 that c_p is a good clock in the interval $[A_p^{(0)}(T^{(0)}), A_p^{(i+1)}(T^{(i+2)})]$. Then, by the definition of a good clock, we have

$$\left| \frac{c_p(A_p^{(i)}(T+\Pi)) - c_p(A_p^{(i)}(T))}{\Pi} - 1 \right| < \frac{\rho}{2},$$

from which the result follows by the identities $c_p^{(i)}(T) = c_p(A_p^{(i)}(T))$, and $c_p^{(i)}(T+\Pi) = c_p(A_p^{(i)}(T+\Pi))$.
□

We are going to need some specializations of Lemma 2. The first will be used to bound expressions of the form

$$|c_p^{(i)}(T+\Phi+\Pi) - [c_p^{(i)}(T+\Phi)+\Pi]|$$

where $T \in S^{(i)}$. Application of Lemma 2 in this case requires us to establish that $A_p^{(i)}(T+\Phi)$ and $A_p^{(i)}(T+\Phi+\Pi)$ are both in the interval $[A_p^{(0)}(T^{(0)}), A_p^{(i+1)}(T^{(i+2)})]$.

Recall that $C_p^{(0)} = 0$, so that $A_p^{(0)}(T) = T$. Thus, in order to satisfy the lower bound $A_p^{(0)}(T^{(0)}) \leq A_p^{(i)}(T+\Phi)$ in the case $i = 0$ and $T = T^{(0)}+R-S$, it is clear that we should require $|\Phi| \leq R - S$. To prove that this condition suffices for the case of general i and T is surprisingly tedious and requires an induction on i.

We have just established the base case; for the inductive step, we assume that $T \in S^{(i)}$ and $|\Phi| \leq R - S$ are sufficient to establish that $A_p^{(0)}(T^{(0)}) \leq A_p^{(i)}(T+\Phi)$ and we note that if $T' \in S^{(i+1)}$, then $T' = T + R$ for $T \in S^{(i)}$. Thus

$$\begin{aligned}
A_p^{(i+1)}(T'+\Phi) &= A_p^{(i+1)}(T+\Phi+R) \\
&= A_p^{(i)}(T+\Phi+R+C_p^{(i+1)}-C_p^{(i)}) \\
&= A_p^{(i)}(T+\Phi)+R+C_p^{(i+1)}-C_p^{(i)} \\
&\geq A_p^{(0)}(T^{(0)})+R+C_p^{(i+1)}-C_p^{(i)}
\end{aligned}$$

where the last line follows from the inductive hypothesis. In order to complete the inductive step, we need to establish that

$$R + C_p^{(i+1)} - C_p^{(i)} \geq 0.$$

This is an easy consequence of S2, C1 (which is used to derive $S < R$), and C2.

To satisfy the upper bound $A_p^{(i)}(T+\Phi) \leq A_p^{(i+1)}(T^{(i+2)})$ in the limiting case $T = T^{(i+1)}$, we need to establish

$$T^{(i+1)} + \Phi + C_p^{(i)} \leq T^{(i+2)} + C_p^{(i+1)}.$$

Now $T^{(i+2)} = T^{(i+1)} + R$ and S2 provides $|C_p^{(i+1)} - C_p^{(i)}| < \Sigma$ so what we need is

$$\Phi \leq R - \Sigma.$$

It is clear that this can be achieved if $|\Phi| \leq R - S$ (as before), and $|\Sigma| \leq S$. The latter constraint is ensured by C2.

We have just sketched the proof of

Lemma 2a: *If processor p is nonfaulty through period $i+1$, $T \in S^{(i)}$, $|\Phi + \Pi| \leq R - S$, and $|\Phi| \leq R - S$, then*

$$|c_p^{(i)}(T + \Phi + \Pi) - [c_p^{(i)}(T + \Phi) + \Pi]| < \frac{\rho}{2}\,|\Pi|.$$

□

We will also require a variant of this result where the only bounds available on Φ and Π are $|\Phi| \leq S$ and $|\Pi| \leq S$. It is easy to see that Lemma 2a can be applied, provided $3S \leq R$—which is the Constraint C1. This yields

Lemma 2b: *If processor p is nonfaulty through period $i+1$, $T \in S^{(i)}$, $|\Phi| \leq S$, and $|\Pi| \leq S$, then*

$$|c_p^{(i)}(T + \Phi + \Pi) - [c_p^{(i)}(T + \Phi) + \Pi]| < \frac{\rho}{2}\,|\Pi|.$$

□

The special case $\Phi = 0$ provides

Lemma 2c: *If processor p is nonfaulty through period $i+1$, $T \in S^{(i)}$, and $|\Pi| \leq S$, then*

$$|c_p^{(i)}(T + \Pi) - [c_p^{(i)}(T) + \Pi]| < \frac{\rho}{2}\,|\Pi|.$$

□

The final specialization of Lemma 2 is Lemma 2d. Like that of Lemma 2a, its proof requires a surprisingly tedious argument (including an induction) to establish that the constraints on Π are sufficient to satisfy the antecedents to Lemma 2.

Lemma 2d: *If processor p is nonfaulty through period i and $0 \leq \Pi \leq R$, then*

$$|c_p^{(i)}(T^{(i)} + \Pi) - [c_p^{(i)}(T^{(i)}) + \Pi]| < \frac{\rho}{2}\,\Pi.$$

□

Lemma 1: *If the clock synchronization conditions S1 and S2 hold for i, and processors p and q are nonfaulty through period $i+1$, then*

$$|\Delta_{qp}^{(i)}| < \Delta.$$

Proof: By A2, we have

$$|\Delta_{qp}^{(i)}| \le S \tag{2.5}$$

and

$$|c_p^{(i)}(T' + \Delta_{qp}^{(i)}) - c_q^{(i)}(T')| < \epsilon$$

for some time T' in $S^{(i)}$. Using the arithmetic identity

$$x = (u - v) + (v - w) - (u - [w + x])$$

we obtain

$$\begin{aligned} |\Delta_{qp}^{(i)}| = | & \; c_p^{(i)}(T' + \Delta_{qp}^{(i)}) - c_q^{(i)}(T') \\ & + c_q^{(i)}(T') - c_p^{(i)}(T') \\ & - (c_p^{(i)}(T' + \Delta_{qp}^{(i)}) - [c_p^{(i)}(T') + \Delta_{qp}^{(i)}])|. \end{aligned}$$

Hence

$$\begin{aligned} |\Delta_{qp}^{(i)}| \le & \; |c_p^{(i)}(T' + \Delta_{qp}^{(i)}) - c_q^{(i)}(T')| \\ & + |c_q^{(i)}(T') - c_p^{(i)}(T')| \\ & + |c_p^{(i)}(T' + \Delta_{qp}^{(i)}) - [c_p^{(i)}(T') + \Delta_{qp}^{(i)}]|. \end{aligned}$$

The first term in the right hand side is the left hand side of the instance of A2 with which we began. Applying S1 and Lemma 2c to the second and third terms, respectively, we obtain

$$|\Delta_{qp}^{(i)}| < \epsilon + \delta + \frac{\rho}{2}\Delta_{qp}^{(i)}$$

from which the conclusion follows by (2.5) (which was also needed to justify application of Lemma 2c) and C4.

□

Lemma 3: *If the clock synchronization conditions S1 and S2 hold for i, processors p and q are nonfaulty through period $i+1$, and $T \in S^{(i)}$, then*

$$|c_p^{(i)}(T + \Delta_{qp}^{(i)}) - c_q^{(i)}(T)| < \epsilon + \rho S.$$

Proof: By A2, we have

$$|\Delta_{qp}^{(i)}| \le S \tag{2.6}$$

and

$$|c_p^{(i)}(T' + \Delta_{qp}^{(i)}) - c_q^{(i)}(T')| < \epsilon$$

for some time T' in $S^{(i)}$. Let $\Pi = T - T'$, so that $T = T' + \Pi$. Using the latter, plus the arithmetic identity

$$x - y = (x - [u + v]) + (u - w) - (y - [w + v]),$$

we obtain:

$$\begin{aligned} &|c_p^{(i)}(T + \Delta_{qp}^{(i)}) - c_q^{(i)}(T)| = \\ &\quad |\ c_p^{(i)}(T' + \Delta_{qp}^{(i)} + \Pi) - [c_p^{(i)}(T' + \Delta_{qp}^{(i)}) + \Pi] \\ &\quad + c_p^{(i)}(T' + \Delta_{qp}^{(i)}) - c_q^{(i)}(T') \\ &\quad - (c_q^{(i)}(T' + \Pi) - [c_q^{(i)}(T') + \Pi])|. \end{aligned}$$

Hence

$$\begin{aligned} &|c_p^{(i)}(T + \Delta_{qp}^{(i)}) - c_q^{(i)}(T)| \le \\ &\qquad |c_p^{(i)}(T' + \Delta_{qp}^{(i)} + \Pi) - [c_p^{(i)}(T' + \Delta_{qp}^{(i)}) + \Pi]| \\ &\quad + \ |c_p^{(i)}(T' + \Delta_{qp}^{(i)}) - c_q^{(i)}(T')| \\ &\quad + \ |c_q^{(i)}(T' + \Pi) - [c_q^{(i)}(T') + \Pi]|. \end{aligned}$$

Applying Lemma 2b to the first term on the right hand side (this is justified by (2.6) and the observation that $|\Pi| \le S$ since T and T' are both in $S^{(i)}$), recognizing the second term as the left hand side of the instance of A2 with which we began, and applying Lemma 2c to the third term, we obtain

$$|c_p^{(i)}(T + \Delta_{qp}^{(i)}) - c_q^{(i)}(T)| < \frac{\rho}{2}\,|\Pi| + \epsilon + \frac{\rho}{2}\,|\Pi|.$$

The result then follows from $|\Pi| \le S$.

□

Lemma 4: *If the clock synchronization conditions S1 and S2 hold for i, processors p, q, and r are nonfaulty through period $i + 1$, and $T \in S^{(i)}$, then*

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)}]| < 2(\epsilon + \rho S) + \rho\Delta.$$

Proof: By Lemma 1, we know that $|\Delta_{rp}^{(i)}| < \Delta$ and $|\Delta_{rq}^{(i)}| < \Delta$. Hence, by the Algorithm, $\bar{\Delta}_{rp}^{(i)} = \Delta_{rp}^{(i)}$ and $\bar{\Delta}_{rq}^{(i)} = \Delta_{rq}^{(i)}$ and so

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)}]| = |c_p^{(i)}(T) + \Delta_{rp}^{(i)} - [c_q^{(i)}(T) + \Delta_{rq}^{(i)}]|.$$

Using the arithmetic identity

$$x - y = (u - y) - (v - x) + (v - w) - (u - w)$$

we obtain

$$\begin{aligned}
&|c_p^{(i)}(T) + \Delta_{rp}^{(i)} - [c_q^{(i)}(T) + \Delta_{rq}^{(i)}]| = \\
&| \quad c_q^{(i)}(T + \Delta_{rq}^{(i)}) - [c_q^{(i)}(T) + \Delta_{rq}^{(i)}] \\
&\quad - (c_p^{(i)}(T + \Delta_{rp}^{(i)}) - [c_p^{(i)}(T) + \Delta_{rp}^{(i)}]) \\
&\quad + c_p^{(i)}(T + \Delta_{rp}^{(i)}) - c_r^{(i)}(T) \\
&\quad - (c_q^{(i)}(T + \Delta_{rq}^{(i)}) - c_r^{(i)}(T))|
\end{aligned}$$

and so

$$\begin{aligned}
&|c_p^{(i)}(T) + \Delta_{rp}^{(i)} - [c_q^{(i)}(T) + \Delta_{rq}^{(i)}]| \leq \\
&\qquad |c_q^{(i)}(T + \Delta_{rq}^{(i)}) - c_q^{(i)}(T) + \Delta_{rq}^{(i)}| \\
&\quad + \quad |c_p^{(i)}(T + \Delta_{rp}^{(i)}) - c_p^{(i)}(T) + \Delta_{rp}^{(i)}| \\
&\quad + \quad |c_p^{(i)}(T + \Delta_{rp}^{(i)}) - c_r^{(i)}(T)| \\
&\quad + \quad |c_q^{(i)}(T + \Delta_{rq}^{(i)}) - c_r^{(i)}(T)|.
\end{aligned}$$

The result follows on applying Lemma 2d to the first two terms in the right hand side (using C2 and C3 to provide $\Delta \leq S$) and Lemma 3 to the remaining two.

□

Lemma 5: *If the clock synchronization condition S1 holds for i, processors p and q are nonfaulty through period $i+1$, and $T \in S^{(i)}$, then*

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)}]| < \delta + 2\Delta.$$

Proof: Using the arithmetic identity

$$(a + x) - (b + y) = (a - b) + (x - y),$$

we obtain

$$\begin{aligned}
|c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)}]| &= |c_p^{(i)}(T) - c_q^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - \bar{\Delta}_{rq}^{(i)}| \\
&\leq |c_p^{(i)}(T) - c_q^{(i)}(T)| + |\bar{\Delta}_{rp}^{(i)}| + |\bar{\Delta}_{rq}^{(i)}|.
\end{aligned}$$

The result follows on applying S1 to the first term on the right hand side, and observing that the Algorithm ensures that the remaining two terms are no larger than Δ.

□

Sublemma A: *If processors p and q are nonfaulty through period i, and $T \in R^{(i)}$, then*

$$|c_p^{(i)}(T) - c_q^{(i)}(T)| < |c_p^{(i)}(T^{(i)}) - c_q^{(i)}(T^{(i)})| + \rho R.$$

Proof: Letting $\Pi = T - T^{(i)}$ (so that $T = T^{(i)} + \Pi$ and $0 \leq \Pi \leq R$), and using the arithmetic identity

$$x - y = (x - [u + v]) + (u - w) - (y - [w + v])$$

we have

$$\begin{aligned} &|c_p^{(i)}(T) - c_q^{(i)}(T)| = \\ &\quad | \; c_p^{(i)}(T^{(i)} + \Pi) - [c_p^{(i)}(T^{(i)}) + \Pi] \\ &\qquad + c_p^{(i)}(T^{(i)}) - c_q^{(i)}(T^{(i)}) \\ &\qquad - (c_q^{(i)}(T^{(i)} + \Pi) - [c_q^{(i)}(T^{(i)}) + \Pi])| \end{aligned}$$

and hence

$$\begin{aligned} &|c_p^{(i)}(T) - c_q^{(i)}(T)| \leq \\ &\qquad |c_p^{(i)}(T^{(i)} + \Pi) - [c_p^{(i)}(T^{(i)}) + \Pi]| \\ &\quad + \; |c_p^{(i)}(T^{(i)}) - c_q^{(i)}(T^{(i)})| \\ &\quad + \; |c_q^{(i)}(T^{(i)} + \Pi) - [c_q^{(i)}(T^{(i)}) + \Pi]|. \end{aligned}$$

The result then follows on applying Lemma 2c to the first and third terms on the right hand side.

□

Lemma 6: *If processors p and q are nonfaulty through period $i + 1$, and $T \in R^{(i+1)}$, then*

$$|c_p^{(i+1)}(T) - c_q^{(i+1)}(T)| < |c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - [c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)}]| + \rho(R + \Sigma).$$

Proof: Using Sublemma A (for the case $i + 1$ rather than i), we obtain

$$|c_p^{(i+1)}(T) - c_q^{(i+1)}(T)| < |c_p^{(i+1)}(T^{(i+1)}) - c_q^{(i+1)}(T^{(i+1)})| + \rho R.$$

By the Algorithm,

$$|c_p^{(i+1)}(T^{(i+1)}) - c_q^{(i+1)}(T^{(i+1)})| = |c_p^{(i)}(T^{(i+1)} + \Delta_p^{(i)}) - c_q^{(i)}(T^{(i+1)} + \Delta_q^{(i)})|.$$

Using the arithmetic identity

$$x - y = (x - [u + v]) - (y - [w + z]) + (u + v - [w + z])$$

we obtain

$$\begin{aligned} &|c_p^{(i)}(T^{(i+1)} + \Delta_p^{(i)}) - c_q^{(i)}(T^{(i+1)} + \Delta_q^{(i)})| = \\ &\quad | \; c_p^{(i)}(T^{(i+1)} + \Delta_p^{(i)}) - [c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)}] \\ &\qquad - (c_q^{(i)}(T^{(i+1)} + \Delta_q^{(i)}) - [c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)}]) \\ &\qquad + c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - [c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)}]| \end{aligned}$$

and hence

$$
\begin{aligned}
&|c_p^{(i)}(T^{(i+1)} + \Delta_p^{(i)}) - c_q^{(i)}(T^{(i+1)} + \Delta_q^{(i)})| \leq \\
&\qquad |c_p^{(i)}(T^{(i+1)} + \Delta_p^{(i)}) - [c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)}]| \\
&\quad + \ |c_q^{(i)}(T^{(i+1)} + \Delta_q^{(i)}) - [c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)}]| \\
&\quad + \ |c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - [c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)}]|
\end{aligned}
$$

Applying Lemma 2c to the first two terms on the right hand side (which is justified because the Algorithm provides $\Delta_p^{(i)} = C_p^{(i+1)} - C_p^{(i)}$, S2 then gives $|\Delta_p^{(i)}| < \Sigma$, and C2 gives $\Sigma \leq S$), we obtain

$$
\begin{aligned}
&|c_p^{(i)}(T^{(i+1)} + \Delta_p^{(i)}) - c_q^{(i)}(T^{(i+1)} + \Delta_q^{(i)})| < \\
&\qquad |c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - [c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)}]| + \rho\Sigma.
\end{aligned}
$$

and the result follows.

□

2.3.2.3 The Correctness Theorem

We divide the correctness theorem into two, and prove separately that the Algorithm maintains S1 and S2.

Theorem 1: *For all processors p and q, if all but at most m processors are nonfaulty through period i, then*

S1: *If p and q are nonfaulty through period i, then for all T in* $R^{(i)}$

$$|c_p^{(i)}(T) - c_q^{(i)}(T)| < \delta.$$

Proof: We use induction on i. The base case $i = 0$ follows from Sublemma A, Assumption A0, and Constraint C5. For the inductive step, we assume the theorem true for i, assume its hypotheses true for $i + 1$, and consider $|c_p^{(i+1)}(T) - c_q^{(i+1)}(T)|$. Lemma 6 then gives

$$|c_p^{(i+1)}(T) - c_q^{(i+1)}(T)| < |c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - [c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)}]| + \rho(R+\Sigma).$$

By the Algorithm, the right hand side equals

$$
\begin{aligned}
&\left| \left(\frac{1}{n}\right) \sum_{r=1}^{n} (c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)}]) \right| + \rho(R + \Sigma) \\
\leq \ & \left(\frac{1}{n}\right) \sum_{r=1}^{n} |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)}]| + \rho(R + \Sigma) \\
\leq \ & \left(\frac{1}{n}\right) [(n - m)(2[\epsilon + \rho S] + \rho\Delta) + m(\delta + 2\Delta)] + \rho(R + \Sigma)
\end{aligned}
$$

where the first term is obtained by applying Lemma 4 to the $n-m$ nonfaulty processors, and the second is obtained by applying Lemma 5 to the m faulty ones. The result then follows from the Constraint C6.

□

Theorem 2: *For all processors p, if all but at most m processors are non-faulty through period i, and processor p is nonfaulty through period i, then*

S2: $|C_p^{(i+1)} - C_p^{(i)}| < \Sigma$.

Proof: The Algorithm defines

$$C_p^{(i+1)} = C_p^{(i)} + \Delta_p^{(i)}$$

and $\Delta_p^{(i)}$ is the average of n terms, each less than Δ. The result follows.

□

Chapter 3

Comparison with the Published Analysis by Lamport and Melliar-Smith

In this chapter we describe the differences between our analysis and that of Lamport and Melliar-Smith, and we describe and discuss the flaws in their presentation.

Our proof of the correctness of the Interactive Convergence Clock Synchronization Algorithm, which was presented in the previous chapter, follows the original proof of Lamport and Melliar-Smith [11] very closely; our only changes are technical ones. Some of these were motivated by the needs of truly formal specification and verification; others were motivated by the need to correct flaws in the original. We begin with changes in the first class, then describe the flaws we discovered in the published proof.

3.1 The Definition of a Good Clock

Lamport and Melliar-Smith define the notion of a good clock relative to a *real time* interval as follows:

> *A clock c is a* good clock *during the real time interval $[t_1, t_2]$ if it is a monotonic, differentiable function on $[T_1, T_2]$, where $T_i = c^{-1}(t_i), i = 1, 2$, and for all T in $[T_1, T_2]$:*
>
> $$\left|\frac{dc}{dT}(T) - 1\right| < \frac{\rho}{2}.$$

This definition obviously presents a considerable challenge for a completely formal specification—it would require axiomatizing a fragment of the differential calculus. Accordingly, we follow Butler [4] and use the Mean-Value Theorem to provide a more tractable definition:

$$\left|\frac{c(T_1) - c(T_2)}{T_1 - T_2} - 1\right| < \frac{\rho}{2}.$$

This formulation avoids the use of derivatives, but still requires use of the inverse clock function. This can be avoided by defining the notion of a good clock relative to a *clock time* interval:

> *A clock c is a* good clock *during the clock time interval* $[T_0, T_N]$ *if*
>
> $$\left|\frac{c(T_1) - c(T_2)}{T_1 - T_2} - 1\right| < \frac{\rho}{2}.$$
>
> *whenever* T_1 *and* T_2 *are clock times in* $[T_0, T_N]$.

The formulation we employ for the notion of a good clock is this last one, except that we rewrite the constraint as

$$|c(T_1) - c(T_2) - (T_1 - T_2)| < \frac{\rho}{2}\,(T_1 - T_2)$$

in order to avoid the use of division and the obligation to ensure $T_1 \neq T_2$.

Notice that although we no longer *explicitly* require a good clock to be monotonic, it follows implicitly as a corollary to our definition that, since ρ is small, the clock function c is strict monotonic increasing (and therefore has an inverse function). This fact is proved as Theorem `monotonicity` in Module `clocks`.

3.2 Explicit Functional Dependencies

We made the functional dependency on i, the synchronization period, explicit in the three subscripted Δ quantities that appear in the Algorithm: where Lamport and Melliar-Smith use Δ_p, Δ_{qp} and $\bar{\Delta}_{qp}$, we use $\Delta_p^{(i)}, \Delta_{qp}^{(i)}$ and $\bar{\Delta}_{qp}^{(i)}$. Thus, $\Delta_{qp}^{(i)}$ is the difference between q's clock and p's observed by p *during the i'th period.* This change is a technical correction necessitated by our use of a strict formalism. An alternative in the case of Δ_{qp} would have been to include it in the scope of the existential quantification in A2 (Skolemization would then have provided the functional dependence on i), but that would have needlessly complicated the technical details of the argument.

Throughout the rest of this Chapter, we use the notation of Lamport and Melliar-Smith (i.e., no superscripts on the Δ functions) whenever we are discussing their proof.

3.3 Approximations and Neglect of Small Quantities

In order to "simplify the calculations" Lamport and Melliar-Smith make approximations based on the assumption that $n\rho \ll 1$. They neglect quantities of order $n\rho\epsilon$ and $n\rho^2$ [11, Section 3.4] and use the notation $x \approx y$ to indicate approximate equality and $x \lesssim y$ to indicate approximate inequality. ($x \lesssim y$ means $x < y'$ for some $y' \approx y$.)

When we first attempted to formalize the proof of Lamport and Melliar-Smith, we followed their example and used approximations. However, we soon discovered that this required use of some unjustifiable axioms; referring to the published proof, we found the corresponding steps to be incorrect there also. One of these steps is in the main induction (invalidating the whole proof), another is in Lemma 4. These are described below.

3.3.1 A Flaw in the Main Induction

The goal of the main induction is to establish the clock synchronization condition S1. This is stated [11, page 63] as

$$|c_p^{(i)}(T) - c_q^{(i)}(T)| < \delta$$

while the inductive step [11, page 66] establishes

$$|c_p^{(i+1)}(T') - c_q^{(i+1)}(T')| \lesssim \delta.$$

Thus, the inductive step establishes the desired result only under the unacceptable hypothesis that $x \lesssim y \supset x < y$. Of course, this immediate difficulty can be remedied by restating S1 as

$$|c_p^{(i)}(T) - c_q^{(i)}(T)| \lesssim \delta$$

but one would then have to reexamine the whole proof in order to be sure that the inductive step and all its lemmas remain true under this weaker premise. □

3.3.2 A Flaw in Lemma 4

Lamport and Melliar-Smith's version of Lemma 1 [11, page 64] establishes, under suitable hypotheses, that $|\Delta_{qp}| \lesssim \delta + \epsilon$. However, their proof of Lemma 4 [11, page 65] requires $|\Delta_{qp}| < \delta + \epsilon$, which is not substantiated by these premises. □

The two examples cited above are definite flaws—the proofs are incorrect as stated. In repairing these flaws we faced a choice: we could either continue to work with the approximations—attempting to get them right—or we could reexamine the whole use of approximations and investigate whether the proof could be carried through with exact inequalities. We chose the latter course. Our motivation was largely aesthetic—we found the use of approximations, and especially the potential appearance of approximate bounds in the statement of the main theorem, to be very unsatisfying. The use of approximate relations also cluttered the mechanical verification—unlike exact arithmetic relations, which are built into our specification language and theorem prover, the approximate relations had to be explicitly axiomatized and, more tediously, cited wherever they were needed. We had also come to doubt Lamport and Melliar-Smith's belief that the use of approximations simplified the *un*mechanized calculations—on the contrary, we found that the need to assure ourselves of the correctness of the approximations was a major complicating factor in understanding their published proof.

Accordingly, we revised the published proof, adding additional terms where necessary so that exact equalities and inequalities could be used. This proved to be quite straightforward and, to us at least, the resulting proof (presented in the previous chapter) is no more complicated than that published by Lamport and Melliar-Smith, and the use of exact bounds is more satisfying. The revisions necessitated by the use of exact inequalities are few and are listed below. Notice that in a couple of cases, the changes are simplifications.

Constraint C5 is changed from

$$\delta \gtrsim \delta_0 + \rho R$$

to

$$\delta \geq \delta_0 + \rho R.$$

Constraint C4 is changed from

$$\Delta \approx \delta + \epsilon$$

to

$$\Delta \geq \delta + \epsilon + \frac{\rho}{2} S.$$

Constraint C6 is formulated as follows by Butler et al. [5]:

$$\delta \geq 2(\epsilon + \rho S) + \frac{2m\Delta}{n-m} + \frac{n\rho R}{n-m}.$$

Lamport and Melliar-Smith use $\Delta \approx \delta + \epsilon$ to eliminate Δ and state the bound as

$$\delta \gtrsim n'(2\epsilon + \rho\,(R + 2S')),$$

where

$$n' = \frac{n}{n-3m}, \quad \text{and}$$
$$S' = \frac{n-m}{n} S$$

We prefer Butler's form and state the revised constraint as

$$\delta \geq 2(\epsilon + \rho S) + \frac{2m\Delta}{n-m} + \frac{n\rho R}{n-m} + \frac{n\rho\Sigma}{n-m} + \rho\Delta.$$

Lemma 1: The conclusion is changed from

$$|\Delta_{qp}| \lesssim \delta + \epsilon$$

to

$$|\Delta_{qp}^{(i)}| < \Delta$$

Lemma 4: The conclusion is changed from

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}]| \lesssim 2(\epsilon + \rho S)$$

to

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)}]| < 2(\epsilon + \rho S) + \rho\Delta.$$

3.4 The Interval in which a Clock is a "Good Clock"

Several lemmas use Definition 1 (the notion of a good clock) and Assumption A1 (a nonfaulty processor has a good clock) to establish bounds on certain quantities. In order to apply these definitions, we must establish that the times concerned fall in the interval during which the processor is hypothesized to be nonfaulty. The statements and proofs of Lemmas 1 and 2 [11, page 64] do not do this with sufficient care and both are false as stated.

3.4.1 Falsehood of Lemma 1

Lamport and Melliar-Smith's proof of Lemma 1 readily establishes

$$|c_p^{(i)}(T_0) - c_p^{(i)}(T_0 + \Delta_{qp})| < \delta + \epsilon$$

where $T_0 \in S^{(i)}$. The next step is to use the fact that p is nonfaulty up to $T^{(i+1)}$ to allow use of Definition 1. In order to be able to do this, it is necessary to show that

$$T_0 + \Delta_{qp} \leq T^{(i+1)}.$$

This constraint is not true in general—T_0 could be as large as $T^{(i+1)}$ and $\Delta_{qp} \geq 0$. However, Lemma 1 is only used when p is known to be nonfaulty up to $T^{(i+2)}$ so a plausible repair would change the statement of the Lemma to require that p be nonfaulty up to $T^{(i+2)}$. Then we would merely need to show that

$$T_0 + \Delta_{qp} \leq T^{(i+2)}. \tag{3.1}$$

Since $T_0 \leq T^{(i+1)}$ and $T^{(i+2)} = T^{(i+1)} + R$ and Δ_{qp} is small, this seems straightforward. However, although Δ_{qp} is assumed small, and the purpose of this very Lemma is to show it is less than Δ, there is no *a priori* bound on its value and therefore no basis to establish (3.1).[1] Hence, this putative proof of even the repaired version of Lemma 1 is flawed. In our proof, we introduce

$$\Delta_{qp}^{(i)} \leq S$$

as an explicit conjunct in Assumption A2. This is sufficient to substantiate our use of Definition 1.

Notice that satisfaction of this strengthened statement for Assumption A2 must be justified for any realization of the Algorithm.

---

[1]It might seem that we could establish that Δ_{qp} must be very small by using the facts the p and q were synchronized during the previous period and cannot have drifted very far since then. This argument, however, merely shows that a suitably small Δ_{qp} must exist—it does not guarantee that this will be the value that is actually obtained. It is possible that a very large value will be returned and that the constraint

$$|c_p^{(i)}(T' + \Delta_{qp}) - c_q^{(i)}(T')| < \epsilon$$

will be satisfied adventitiously because the large value for Δ_{qp} takes p's clock beyond the interval in which it is a good clock—so that $c_p^{(i)}(T' + \Delta_{qp})$ may have any value whatever.

3.4.2 Falsehood of Lemma 2

There is a similar problem in the proof of Lemma 2. In order to substantiate the use of Assumption A1, it is necessary to ensure that

$$A_p^{(i)}(T + \Pi) \leq A_p^{(i+1)}(T^{(i+2)})$$

where $T \in S^{(i)}$ and $|\Pi| < R$. Expanding definitions, this requires

$$T^{(i+1)} - \Phi + \Pi + C_p^{(i)} \leq T^{(i+1)} + R + C_p^{(i+1)}$$

where $0 \leq \Phi \leq S$. For the case where $\Phi = 0, \Pi \geq 0$, and using S2, this reduces to

$$\Pi \leq R - \Sigma$$

which is *not* ensured by the condition $|\Pi| < R$. Similar difficulty arises in satisfying the lower bound to the interval required for application of A1.

In our proof we introduce several variations on Lemma 2, each with tighter bounds on Π and/or T, and we also introduce the new constraints C1 ($3S \leq R$) and C2 ($\Sigma \leq S$) in order to overcome these difficulties. These particular constraints were chosen for simplicity, and because we felt that there would be no difficulty satisfying them in any likely implementation. Alternative constraints are feasible, and would require minor modifications to the proof.

3.5 Sundry Minor Flaws and Difficulties

3.5.1 Falsehood and Unnecessary Generality of Lemma 3

As stated, the Lemma is false because the bounds on Π are insufficiently tight to substantiate use of Assumption A1 (the argument is exactly the same as that for Lemma 2). However, Π is instantiated with 0 the only time that the Lemma is used (in Lemma 4). In our proof, we discarded the parameter Π, thereby correcting and simplifying the statement and proof of the Lemma.

3.5.2 Missing Requirements for Clock Synchronization Condition S2

The proofs of Lemmas 1 and 3 use Assumption A2, which requires that S2 should hold. Since Lemma 4 uses Lemmas 1 and 3, its statement should also require that S2 hold. The statements of all three Lemmas omit this condition.

As stated, Lemma 2 also requires that only S1 hold. When other necessary corrections to the statement and proof of the Lemma are made, it becomes necessary to require that S2 hold as well (in order to bound the extent to which the interval $[T^{(i+1)}, T^{(i+2)}]$ can "shrink" when the correction $C_p^{(i+1)}$ is applied).

3.5.3 Typographical Errors in Lemmas 2 and 4

The conclusion to the first part of Lemma 2 states that a certain quantity is strictly less than $\left(\frac{\rho}{2}\right)\Pi$. This should be $\left(\frac{\rho}{2}\right)|\Pi|$.

The conclusion to Lemma 4 is stated as

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp} - [c_q^{(i)}(T) - \bar{\Delta}_{rq}]| < 2(\epsilon + \rho S).$$

It should read

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}]| < 2(\epsilon + \rho S).$$

These seem to be no more than typographical errors.

Chapter 4

Formal Specification and Verification in EHDM

In this chapter we describe the formal specification of the Interactive Convergence Clock Synchronization Algorithm and its mechanical verification using the EHDM formal specification and verification environment. This entails encoding the Algorithm and its supporting definitions, assumptions, lemmas, and theorems in the specification language of EHDM, and then proving those lemmas and theorems with the help of the EHDM theorem prover.

We begin with an overview of those features of EHDM and its specification language that are necessary for an understanding of this particular application, then we describe our application of the system to the Interactive Convergence Clock Synchronization Algorithm.

4.1 Overview of EHDM

The EHDM Specification and Verification System is an interactive system for the composition and analysis of formal specifications and abstract programs written in the EHDM specification language. Its development by the Computer Science Laboratory of SRI International is sponsored by the National Computer Security Center.

A general overview of EHDM is provided in [18], where further references may also be found. EHDM is written in Common Lisp and implementations are available for Symbolics and Sun workstations. The specification and verification described here was performed on a Sun workstation using EHDM Version 4.1.4.

Our specification and verification of the Interactive Convergence Clock Synchronization Algorithm uses only some of the capabilities of EHDM. Specifically, it uses unparameterized modules, the functional component of the specification language, the ground prover, and the proof chain analyzer.[1] In this section we will describe only those parts of EHDM that are needed to understand our specifications and proofs for the Interactive Convergence Clock Synchronization Algorithm. Readers who wish to know more about EHDM should consult the references cited earlier.

4.1.1 The Specification Language

The fragment of the EHDM specification language used here is a strongly typed version of the First-Order Predicate Calculus, enriched with elements of other logics—specifically Higher-Order Logic and the Lambda Calculus. The two volumes by Manna and Waldinger [13, 14] provide an introduction to some of these topics that is especially suitable for computer scientists; Andrews [3] gives a more detailed treatment, including a good discussion of Higher-Order Logic.

4.1.1.1 Declarations

The EHDM specification language allows the declaration of five different sorts of entities: types, variables, constants, formulas, and proofs. There are six built-in types in EHDM (that is, types which for which the system provides an interpretation). The five of interest here are the rational numbers (indicated by the identifier `number`), the integers (indicated by the identifiers `integer` or `int`), the natural numbers (indicated by the identifiers `naturalnumber` or `nat`), the booleans (indicated by the identifiers `boolean` or `bool`), and the function types (which are described shortly). In addition, the user may introduce uninterpreted types, type synonyms, and subtypes. Here, we use only the built-in types, plus type synonyms. The declaration

[1] The capabilities not used here include parameterized modules and assuming clauses, mapping modules, the procedural component of the specification language, the instantiator for the theorem prover, the Hoare-Sentence prover, the Ada Translator, and the multilevel security analyzer. We plan to construct a procedural description of the Interactive Convergence Clock Synchronization Algorithm at some time in the future; this will enable us to demonstrate the procedural component of the specification language, the Hoare-Sentence Prover, and possibly the Ada Translator.

```
clocktime: TYPE IS number
```

introduces clocktime[2] as a synonym for the natural numbers (equivalently, we can think of the natural numbers as supplying the interpretation for the type `clocktime`).

Variables are introduced by declarations of the form

```
T1, T2: VAR clocktime
```

while uninterpreted constants are introduced by declarations of the form

```
T_ZERO: clocktime
```

Constants of a built-in type can be given an interpretation using a literal value of that type, for example:

```
T_ZERO: clocktime = 0
```

Function types are written as follows:

```
X: TYPE IS function[processor, period, clocktime -> realtime]
```

where the type-identifiers preceding the `->` indicate the domain of the function type, and that following indicates the range.

EHDM is a higher-order language, so that function types may have other function types in their domain or range, for example

```
foo: TYPE IS function[nat, nat, function[nat -> number] -> number]
```

Functions are simply constants of a function type:

```
correction: function[processor, period -> clocktime]
```

There is no special notation for predicates; a predicate is simply a function with range `bool`:

```
goodclock: function[processor, clocktime, clocktime -> bool]
```

It is also perfectly feasible to have variables of a function type:

---

[2]EHDM identifiers consist of a letter, followed by a sequence of letters, digits, and the underscore character. Identifiers are case sensitive: `t1` and `T2` are different identifiers. The keywords of EHDM are *not* case sensitive, however: `type`, `TYPE`, and even `tYpE` all denote the same keyword. By convention we put keywords in upper case. (This is the default used by the EHDM prettyprinter.)

```
prop: VAR function[nat -> bool]
```

Literal values of a function type are denoted using lambda-notation, and may be used to give an interpretation to a function constant. The following specification fragment gives an example.[3]

```
p: VAR processor
i: VAR period
T: VAR clocktime

adjusted: function[processor, period, clocktime -> clocktime] =
   (LAMBDA p, i, T -> clocktime: T + correction(p, i))
```

Formula declarations have the following schema:

name: KEY value

where the *name* is simply an identifier that is used to refer to the formula, *KEY* is one of the keywords FORMULA, AXIOM, LEMMA, or THEOREM,[4] and *value* is boolean-valued expression.

Expressions can be built up from the usual propositional connectives (which are written as NOT, AND, OR, IMPLIES, and IFF), universal and existential quantification, function application (written in the usual prefix notation—e.g., adjusted(p, i, T)), equality (written as =),[5] disequality (written as /=), the usual arithmetic operations (written as -, +, * and /), and the relations of arithmetic inequality (written as <, <=, >, and >=). There is also a three-place *if-then-else* operator that is written, for example, as:

```
abs_def: AXIOM abs(x) = IF x < 0 THEN -x ELSE x END IF
```

Quantified expressions are written in the following form:

[3]Notice that unlike many programming and specification languages, EHDM declarations are *not* terminated by a semi-colon.

[4]These four keywords are almost equivalent (AXIOM is actually distinguished from the other three). However, they are meant to be used in a way that indicates the specifier's intention: an AXIOM is something intended to be taken as primitive, while LEMMA and THEOREM indicate something that will be proved. We use FORMULA to indicate something that ought to be proved but is not (i.e., a "temporary" axiom). The EHDM Proof-Chain Checker is used to ensure that all non-AXIOMs are ultimately consequences only of AXIOMs and PROOFs.

[5]The symbol = denotes logical equivalence when its arguments are of type boolean—it is a synonym for IFF in this case.

```
R: clocktime
T, PI: VAR clocktime
i: VAR period
T_sup: function[period -> clocktime]
in_R_interval: function[clocktime, period -> boolean]

Rdef: AXIOM in_R_interval(T, i) =
    (EXISTS PI: 0 <= PI AND PI <= R AND T = T_sup(i) + PI)
```

Free variables in EHDM formulas are treated as if they are universally quantified at the outermost level (i.e., formulas denote their universal closure). Thus, the following is equivalent to the AXIOM of the same name given earlier:

```
abs_def: AXIOM (FORALL x: abs(x) = IF x < 0 THEN -x ELSE x END IF)
```

It is generally easier to read formulas when this outer level of quantification is omitted.

EHDM permits overloading of function names and provides subtype-to-supertype coercions. This is of some importance when dealing with arithmetic. The naturals are defined as a subtype of the integers, which in turn are defined as a subtype of the (rational) numbers. The binary arithmetic functions and relations require both their arguments to be of the same type; the function and relation symbols actually denote different functions according to the type of their arguments. If an arithmetic function or relation is supplied with arguments of different types, then a subtype to supertype coercion is applied until the types match. Thus, in the following fragment

```
n: VAR nat
i: VAR int
r: VAR number

X: FORMULA r = i + n
```

it is addition on the integers that is supplied as the interpretation of the + sign (n is coerced to integer), the result is coerced to a (rational) number, and the equality function used is that for the (rational) numbers.

4.1.1.2 Modules

Specifications in EHDM are structured into named units called *modules* in much the same way as programs written in modern programming languages are composed of similar units (e.g., packages in Ada). A module serves to group related concepts together and delimits the scope of names. An (unparameterized) EHDM module consists of three parts, any of which may be empty: an import/export part, a theory part, and a proof part.

Declarations of all the forms described above may appear in both the theory and proof parts (except that AXIOMs may not appear in a proof part). Types and constants declared in the theory part may be made visible to the theory parts of other modules by listing them in the exporting part—for example:

```
EXPORTING R, in_R_interval
```

Other modules gain access to these names by citing the name of the module in which they are declared in their USING clauses (as the import list is called in EHDM). A module A which imports a module B may re-export all the names imported from B by adding a WITH clause to its own exporting list:

```
USING A
EXPORTING p, q, r WITH A
```

This makes all the names exported by A visible to any module that imports B, without that module having to import A explicitly.

All names declared in a theory part, whether exported or not, are visible inside the proof part of any module that imports the module concerned. Conversely, *nothing* declared in a proof part is ever visible outside that proof part.

The reader should now have enough understanding of the specification language of EHDM to be able to read the simple module example, which is a simplified form of the module clocks used in the actual specification of the Interactive Convergence Clock Synchronization Algorithm. The module (which has no proof part) is shown in Figure 4.1

4.1.1.3 Proofs

EHDM proof declarations provide information that tells the EHDM theorem provers how to prove the formula concerned. There are two main theorem proving components in EHDM: the *ground prover*, and the *proof instantiator*. All the proofs described here were done with the ground prover. The following description covers both provers.

A proof declaration in EHDM has the general form

```
name: PROVE conclusion FROM premise1, premise2, premise3
```

```
example: MODULE

USING time

EXPORTING proc, clock, rho, Corr, adjusted WITH time

THEORY

  proc: TYPE IS nat

  rho: number

  rho_pos: AXIOM half(rho) >= 0

  clock: function[proc, clocktime -> realtime]

  p: VAR proc

  T, T0, T1, T2, TN: VAR clocktime

  goodclock: function[proc, clocktime, clocktime -> bool]

  gc_ax: AXIOM
    goodclock(p, T0, TN)
      = (FORALL T1, T2 :
          T0 <= T1 AND T0 <= T2 AND T1 <= TN AND T2 <= TN
            IMPLIES abs(clock(p, T1) - clock(p, T2) - (T1 - T2))
              < mult(half(rho), abs(T1 - T2)))

  Corr: function[proc, period -> clocktime]

  zero_correction: AXIOM Corr(p, 0) = 0

  i: VAR period

  adjusted: function[proc, period, clocktime -> clocktime] =
    (LAMBDA p, i, T -> clocktime : T + Corr(p, i))

END example
```

Figure 4.1: An Example EHDM Specification Module

where the `conclusion` and the `premises` (there can be any number of premises) are the names of formulas. This declaration indicates that the `conclusion` is to be proven to be a valid consequence of the `premises`—i.e., $p_1, p_2, p_3 \vdash c$ in the conventional notation of logic. By the deduction theorem, this is equivalent to $\vdash p_1, p_2, p_3 \supset c$, which is equivalent to the unsatisfiability of

$$\neg c \wedge p_1 \wedge p_2 \wedge p_3 \tag{4.1}$$

The theorem provers of EHDM are refutation-based provers, and their strategy is to attempt to show that (4.1) (i.e., the conjunction of the premises and the negated conclusion) is unsatisfiable. The first step on the way to accomplishing this goal is to reduce (4.1) to an equivalent quantifier-free form by the process of Skolemization. The details of Skolemization are somewhat tedious to describe (see [14] for a general explanation) but the important point is that the existentially quantified variables in the premises, and the universally quantified and unquantified variables in the conclusion, are replaced by constants.[6]

If the remaining variables in the quantifier-free formula resulting from Skolemization are substituted with expressions made up of constants (such expressions are called *ground terms*), then (ignoring arithmetic for the moment) the result will be a formula of the Propositional Calculus. Since Propositional Calculus is decidable, it can be readily determined whether this formula (which is called a ground instance of the original predicate calculus formula (4.1)) is unsatisfiable. If it is, then so is (4.1)—which means the original theorem has been proven. If the ground instance is not unsatisfiable, it does *not* mean that (4.1) is unsatisfiable, nor that the original theorem is false—it means only that the particular set of ground substitutions chosen did not establish the theorem. However, by the Herbrand-Skolem-Gödel theorem, we know that if the original theorem is valid, then there exists *some* set of substitutions that produces an unsatisfiable ground instance.

The ground prover of EHDM is simply a decision procedure for the combination of propositional calculus with equality over uninterpreted function symbols, plus "extended quantifier-free Presburger arithmetic[7] for both the rationals and integers" [17]. Proof declarations for the EHDM ground prover

[6]This description ignores the effects of explicit and implicit negations (the latter are introduced by implications and equivalences). More precisely, it is the *odd* variables in the premises and the *even* ones in the conclusion that are replaced by constants—and those constants may be functions in the general case.

[7]This includes unary minus, addition and subtraction, multiplication by constants, equality and disequality, together with the relations $<, \leq, \geq$, and $>$.

must indicate the substitutions to be used to produce the ground instance that is submitted to the ground prover. Substitutions are indicated as follows:

```
name {v1 <- e1, v2 <- e2, ... , vn <- en}
```

where name is a formula name appearing in a PROVE declaration as either the conclusion or a premise, the vi's are substitutable (unSkolemized) variables of the formula, and the ei's are ground terms. For example:

```
abs_proof0: PROVE abs_ax0 FROM abs_ax {a <- 0}
```

Not all substitutions involve literal constants; most refer to the Skolem or substitution instances of variables in other premises or in the conclusion. The notation for this appends an "@" sign and a qualifier to the variable concerned. Thus the substitution x <- y@c means "substitute for x whatever is substituted for y in the conclusion," and x <- y@p3 means "substitute for x whatever is substituted for y in the 3'rd premise." More complex forms, such as x <- y@c+z@p3 are perfectly acceptable. When function variables are concerned, the substitutions may involve LAMBDA terms.

The number of substitutions that must be given explicitly is greatly reduced by application of a number of default rules. If no qualifier is given (as in the substitution x <- y), then y is interpreted to mean "the instance of y in the conclusion, if there is one, otherwise the instance from this premise." If no substitution at all is given for a variable, then (for the case of a variable x) the substitution x <- x is supplied automatically (and the interpretation of the missing qualifier will be supplied by the previous rule).

This all sounds much more complicated than it really is. A typical proof (from the module time in the specification) is shown below:

```
inRS_proof: PROVE inRS FROM Sdef, Rdef {PI <- R-S+PI@p1}, SinR
```

The mechanics of doing a proof in EHDM are that the user moves the cursor to the proof declaration of interest and presses the "prove" button. (The interface to EHDM is a screen editor with mouse-sensitive pop-up menus.) In the fullness of time, the system will report either "proved" (meaning just that) or "unproved" (meaning either that the theorem is false, or that it is true, but the premises and substitutions provided are not sufficient to establish that fact). There is no direct interaction with the ground prover; all the interaction is through the specification text (though there are some proof-debugging tools). In addition to the commands for performing a single proof, there are commands for doing all the proofs in a module, or all the proofs in a module *and* all those modules that it uses.

It will be clear from our description that the ground prover of EHDM is really a proof checker: all the creative work is in the selection of the premises and of the substitutions—and this is performed by the user. EHDM contains another theorem proving component called the *instantiator* that can perform some of these tasks automatically. Specifically, the instantiator tries to supply the substitutions needed to make a proof succeed. If it finds the correct substitutions, it can write them back into the specification text so that in future the ground prover will be able to perform the proofs on its own.

The instantiator is a full first-order theorem prover: it can prove any true theorem of first-order predicate calculus. However, its effectiveness in finding suitable substitutions is considerably diminished in the presence of interpreted symbols, such as those for equality and arithmetic. (For example, it succeeds on only 4 of the 12 proofs in the module absolutes if all the explicit substitutions are deleted.) Since the specifications of the Interactive Convergence Clock Synchronization Algorithm make heavy use of arithmetic, we did not use the instantiator in this effort. The powerful arithmetic capabilities of the EHDM ground prover were crucial to our ability to perform this work.

4.1.1.4 Other Components of the EHDM System used in the Proof

Proof Chain Checker. The notion of "proof" that is established by the EHDM theorem prover is a local one: it assures us that the conclusion is indeed a valid consequence of the premises. But it does not tell us whether those premises are axioms or theorems, and if the latter, whether or not they have been proved. This larger scale analysis is performed by an EHDM tool called the "Proof Chain Checker." The Proof Chain Checker can be invoked with either a PROVE or a FORMULA declaration as its target. In the latter case, it first searches for a proof of the formula concerned; in either case it then recursively examines the status of all the premises named in the proof. Proof Chain Analyses for the clock synchronization conditions in our specification are given in Appendix C.

Prettyprinters. The written appearance of specifications has a significant impact on the ease with which they can be read, understood—and written. The concrete syntax of the EHDM specification language attempts to be close to traditional mathematical and logical notation. A rather sophisticated prettyprinter helps ensure a uniform lexical style for specifications. The specification listings in Appendix D were produced by the prettyprinter.

Even given the relatively straightforward concrete syntax of EHDM, it can still be hard to read specifications composed of long series of function applications. Thus, we developed a table-driven "LaTeX-printer" for EHDM that converts EHDM specifications into LaTeX input. This can then be processed by LaTeX to produce very readable specifications, with two-dimensional layout including sub- and superscripts and "mix-fix" function symbols. For example, a functional expression in EHDM

```
abs(c(p, i, T) - c(q, i, T))
```

can be converted to the more comprehensible notation

$$|c_p^{(i)}(T) - c_q^{(i)}(T)|.$$

When a function name is used alone (for example, in a declaration), it is printed as a template indicating argument positions. Thus, for example,

$A_{\star 1}^{(\star 2)}(\star 3)$: `function[proc, period, clocktime → clocktime]`

makes it clear that the first argument will appear as a subscript, the second as a parenthesized superscript, and the third in normal parentheses. We expect this tool to become a very useful addition to the EHDM environment, since it greatly assists the reading of specifications and should thereby contribute greatly to the peer review and evaluation of EHDM specifications. The LaTeX-printed version of the example from Figure 4.1 is shown in Figure 4.2.

We used the LaTeX-printer to convert our EHDM specifications into the exact notation used by Lamport and Melliar-Smith; the listings in LaTeX form are given in Appendix B. The translations used for the EHDM identifiers are displayed in Table A.1 of Appendix A.

Cross-Reference Tools. There are nearly 300 EHDM identifiers declared in our specification of the Interactive Convergence Clock Synchronization Algorithm. Keeping track of the declarations and uses of these identifiers could become quite burdensome, so the EHDM environment provides simple cross-reference functions to assist in this task. Two of these functions allow the user to locate and jump to the declarations and uses, respectively, of a example: **Module**

Using time

Exporting proc, $c_{\star 1}(\star 2), \rho, C_{\star 1}^{(\star 2)}, A_{\star 1}^{(\star 2)}(\star 3)$ **with** time

Theory

proc: TYPE IS nat
ρ: number

rho_pos: **Axiom** $\frac{\rho}{2} \geq 0$

$c_{\star 1}(\star 2)$: function[proc, clocktime → realtime]
p: VAR proc
T, T_0, T_1, T_2, T_N: VAR clocktime
goodclock: function[proc, clocktime, clocktime → bool]

gc_ax: **Axiom**

$$\begin{aligned}
&\text{goodclock}(p, T_0, T_N) \\
&\quad = (\forall\, T_1\, , T_2 : \\
&\qquad T_0 \leq T_1 \wedge T_0 \leq T_2 \wedge T_1 \leq T_N \wedge T_2 \leq T_N \\
&\qquad\quad \supset |c_p(T_1) - c_p(T_2) - (T_1 - T_2)| < \tfrac{\rho}{2} \times |T_1 - T_2|)
\end{aligned}$$

$C_{\star 1}^{(\star 2)}$: function[proc, period → clocktime]

zero_correction: **Axiom** $C_p^{(0)} = 0$

i: VAR period
$A_{\star 1}^{(\star 2)}(\star 3)$: function[proc, period, clocktime → clocktime] =
(λ p, i, T→ clocktime : $T + C_p^{(i)}$)

End example

Figure 4.2: LaTeX-printed Example EHDM Specification Module

given identifier; the third provides a tabular cross-reference to all declarations in a given EHDM library. (EHDM allows specification modules to be collected into "libraries" and manipulated as a group.)

The table produced by this third function of the EHDM cross-reference tool is given in Tables A.2 to A.14 in Appendix A.

4.2 The Formal Specification and Verification of the Algorithm

A formal specification generally divides into two components: one directly concerned with the problem at hand, and another in which are developed all the "supporting theories" needed in the first but peripheral to its main purpose. The supporting theories provide the "background knowledge" that we would like to be able to assume in order to get on with the main problem. With a formal specification system, the built-in "background knowledge" is generally very limited (usually it is little more than predicate calculus with equality) and the construction of explicit specifications for the supporting theories may often consume the greater part of a specification effort. It has been recognized for a long time that the development of certified libraries of generally useful supporting theories would be one of the most useful contributions to reducing the cost and increasing the reliability of formal specifications. The module library mechanism of the EHDM system provides a suitable framework for standard modules; however, the libraries have not yet been populated.

Examination of Chapter 2 will show that the background knowledge used in the specification and analysis of the Interactive Convergence Clock Synchronization Algorithm includes a significant amount of arithmetic, including inequalities, absolute values, and summations, but not much else. Since we define a good clock without recourse to differentiation, we avoid the need for real numbers and can use the rationals to represent time.

As mentioned earlier, integer and rational arithmetic are built into EHDM. Thus, the only supporting theories for arithmetic that we need to specify explicitly are those for absolute values and for summation. Because EHDM uses a higher-order logic, induction schemes are provided axiomatically, rather than being built in as rules of inference; consequently, we will also need a supporting theory to provide a suitable induction axiom.

Our specification and verification of the Interactive Convergence Clock Synchronization Algorithm is described in the three subsections following.

First we describe the EHDM modules that provide the supporting theories, then those that build up the specification of the Algorithm, and finally those that develop the proof that the Algorithm maintains synchronization. Listings of the specification modules described here are given in LaTeX-printed form in Appendix B and in raw form in Appendix D. Cross-references are provided in Appendix A.

4.2.1 Supporting Theories

Seven modules provide supporting theories for the specification.

4.2.1.1 Absolutes

Absolute values are used extensively in the specification. It would be entirely feasible to specify the absolute-value function in EHDM by the definition

```
a: VAR number
abs: function[number -> number] =
        (lambda a -> number: if a<0 then -a else a end if)
```

However, this would result in the definition being expanded everywhere it appeared—which would work, but would slow the theorem prover down considerably.[8] Thus we chose to specify the abs function by means of an explicit axiom, so that we could control when the definition is expanded.

```
a: VAR number
abs: function[number -> number]
abs_ax: AXIOM abs(a) = if a<0 then -a else a end if
```

We could have stopped there, but decided it would be preferable to build up a collection of useful proved results about the abs function. We were partly motivated by concerns for theorem proving efficiency, and partly by a desire to make our proofs as readable as possible. For example, if a proof needs the property $|x + y| \leq |x| + |y|$, it is not only more efficient to supply this to the theorem prover explicitly (rather than merely provide abs_ax), but it also makes it easier for a reader to follow the proof. This use of derived properties (rather than referring everything back to definitions) is, of course, quite normal in traditional mathematical presentations. A collection of some dozen elementary results of this kind are collected and proved in the module absolutes.

[8] For example, expanding the definition of abs will only complicate the proof of the formula a=b IMPLIES abs(a)=abs(b).

In addition, the module **absolutes** contains two axioms that state properties of the absolute value function in the presence of multiplication and division:

```
abs_times: AXIOM abs(a*b) = abs(a) * abs(b)
abs_div: AXIOM b /= 0 IMPLIES abs(a / b) = abs(a) / abs(b)
```

As explained in more detail in the following subsection, multiplication and division are largely uninterpreted in EHDM so it is necessary to introduce properties such as these either by means of explicit axioms, or as derived consequences of a more primitive axiomatization for multiplication and division. We have chosen the former course.

4.2.1.2 Arithmetics

Although we said earlier that most of the arithmetic needed was built-in to EHDM, we were not quite telling the truth. EHDM supports *linear* arithmetic—that is multiplication by constants only. Several of the formulas and constraints needed in the specification and verification of the Interactive Convergence Clock Synchronization Algorithm require use of nonlinear multiplication, and also division—e.g., terms such as $\frac{n\rho R}{n-m}$ appear in the constraint C6.

Although it has a special syntactic form (the infix /), division is uninterpreted in EHDM—the user must supply appropriate axioms just as if it were a newly introduced function. Ideally, EHDM should provide a library module containing a "standard" axiomatization for division, but this is not done at present. Accordingly, we provide some *ad hoc* axioms for division in the module **arithmetics**. These axioms and the lemmas derived from them are adequate for the present purpose, but we have made no attempt to construct a minimal or a complete set. The three axioms that we use are shown below (the axiom **abs_div** in module **absolutes** is also relevant).

```
quotient_ax: AXIOM y /= 0 IMPLIES x / y = x * (1 / y)
quotient_ax1: AXIOM x /= 0 IMPLIES x / x = 1
quotient_ax2: AXIOM z > 0 IMPLIES 1 / z > 0
```

Several additional properties of division are stated and proved from these axioms.

Multiplication by literal integer constants is treated as repeated addition by EHDM, and the ground theorem prover is able to fully decide formulas containing such constructs. Nonlinear multiplication can also appear in

EHDM specifications, but is treated as an "almost" uninterpreted function. It might be better, in fact, if it was completely uninterpreted—so that the user could supply and invoke appropriate multiplication axioms under explicit control. As it is, the ground prover of EHDM contains heuristics that enable it to prove certain results involving nonlinear multiplication, but these heuristics render the ground prover incomplete (i.e., it is no longer a decision procedure)[9] —which is unacceptable, given the proving paradigm used in EHDM.

Consequently, the ground prover contains conservative checks that abort the proof if there is any possibility that the presence of nonlinear multiplication will take it beyond its domain of completeness. The only thing to do when a proof aborts in this way is to define a new, uninterpreted multiplication function and use that instead of the built-in function when nonlinear multiplication is required. The semantics of the new multiplication function have to be provided by explicit axiomatization.[10]

Thus, in the module `arithmetics`, we define a function `mult` on the rationals and give it the semantics of multiplication by the axiom

```
mult_ax: AXIOM mult(x, y) = x * y
```

We introduce two additional axioms

```
mult1: AXIOM x >= 0 AND y >= 0 IMPLIES mult(x, y) >= 0
mult_mon: AXIOM x < y AND z > 0 IMPLIES mult(x, z) < mult(y, z)
```

since attempts to derive these results from the first cause the prover to abort and report that it is outside its domain of completeness. Several additional properties of `mult` are stated and proved from these two axioms.

The quantity $\frac{\ell}{2}$ appears frequently in the proof. We encode this in the function `half` defined by the following axiom:

```
half_ax: AXIOM half(x) = x/2
```

We also state and prove a couple of derived properties of this function.

The module `arithmetics` is completed by the statement and proof of two arithmetic identities (`rearrange` and `rearrange_alt`) that are used in a couple of other modules. Several other arithmetic identities of this form are used only once each and are stated and proved in the modules where they are required.

[9] There is no complete decision procedure for arithmetic with multiplication and there is no syntactic characterization for the fragment of nonlinear arithmetic that is decided by the EHDM ground prover.

[10] We are actively considering changes in the way EHDM handles nonlinear multiplication as part of a review of the prover strategies.

4.2.1.3 Natprops

EHDM does not define a subtraction operator on the natural numbers. The naturals are treated as a subtype of the integers in EHDM, so that the expression `n - m`, where `n` and `m` are naturals, is interpreted by coercing those values to type integer, and then applying the integer subtraction operator to yield an integer result. In our treatment of summations, we need subtraction-like operators on the naturals, and these are defined axiomatically in the module `natprops`. The predecessor function, `pred`, and a subtraction function `diff` are defined as follows:

```
pred: function[nat -> nat]
pred_ax: AXIOM n /= 0 IMPLIES pred(n) = n - 1

diff: function[nat, nat -> nat]
diff_ax: AXIOM n >= m IMPLIES diff(n, m) = n - m
```

Several derived properties of these two functions are stated and proved in the module `natprops`. In addition, we assert that the naturals are nonnegative using the following axiom:

```
natpos: AXIOM n >= 0
```

This is necessary because EHDM treats the naturals as simply a subtype of the integers that is closed under addition; no other properties of the naturals are built into the prover.

4.2.1.4 Functionprops

The module `functionprops` defines the (higher-order) axiom of function extensionality. This is required for one of the proofs in the module `sigmaprops`. We define this axiom for functions of exactly the signature we require (i.e., `nat -> number`) rather than for the more general case (i.e., `number -> number`) because the present version of the EHDM typechecker does not handle higher-order subtypes.

```
F, G: VAR function[nat -> number]
x: VAR nat

extensionality: AXIOM (FORALL x : F(x) = G(x)) IMPLIES F = G
```

4.2.1.5 Natinduction

The module natinduction provides a higher-order axiom called induction_m used for inductive proofs. The axiom states a principle of simple induction on the naturals using a predicate variable prop.

```
induction: AXIOM
  (prop(m)
      AND (FORALL i : i >= m AND prop(i) IMPLIES prop(i + 1)))
    IMPLIES (FORALL n >= m : prop(n))
```

Informally, it says that if prop is true for m, and prop(i) implies prop(i+i), for arbitrary i >= m, then prop is true for all natural numbers n >= m. Two special cases of this induction scheme are then introduced as lemmas: induction is the case m = 0 and corresponds to the standard induction scheme over the naturals; induction_1 is the case m = 1.

Module natinduction also introduces modified induction schemes called mod_induction and mod_induction1 that are stated as lemmas and proved from the basic induction_m axiom. The modified scheme mod_induction is used in the proof of Theorem_1 and is specialized for the proof of predicates of the form $A(i) \supset B(i)$. The inductive step in such cases has the form

$$(A(i) \supset B(i)) \supset (A(i+1) \supset B(i+1)).$$

This is equivalent to

$$((A(i) \supset B(i)) \wedge A(i+1)) \supset B(i+1)$$

which, when we know in addition that $A(i+1) \supset A(i)$, reduces to

$$(A(i+1) \wedge B(i)) \supset B(i+1).$$

This is the form for the inductive step that is stated in mod_induction and proved in mod_induction_proof. The lemma mod_induction1 is derived in a similar fashion.

Another induction scheme is introduced as an axiom: induction2 is used in the proof of sigma_rev in module sigmaprops and is specialized for the case when the proposition to be proved takes two arguments, and the induction is over the second. It can be derived from the standard induction scheme, with the addition of quantification over the first argument.

4.2.1.6 Sums and Sigmaprops

Choosing how primitive the axiomatic basis for a supporting theory should be is a matter of taste, conscience, and the time and funds available. Ideally, each supporting theory should be built up from a small and primitive set of self-evident, well-accepted axioms. Unfortunately, it may then require a considerable expenditure of time and effort to build the body of verified lemmas and theorems for the supporting theory that are needed to solve the actual problem at hand. The alternative is to simply assert as axioms the results that are actually needed from the supporting theory. The danger here is self-evident—it is remarkably easy to state plausible, but false axioms.

When formal specification and verification is practised more widely, we would expect that verified libraries of common supporting theories will be available. In the meantime, we are confronted with a dilemma: either build up the supporting theories from primitive axioms—and risk never getting to the original problem of interest, or else concentrate on the original problem—and risk building on sand. We pursued a variant of the second course in developing this proof of the Interactive Convergence Clock Synchronization Algorithm. In order to make progress on the main problem, we adopted expedient axioms at first, then as time has permitted, we went back to develop the supporting theories with greater care and with a view to incorporating them in libraries.

Our first verification of the Interactive Convergence Clock Synchronization Algorithm used high-level axiomatizations of the concepts of summations and means from the module `sums`. Later, we developed a module `sigmaprops` that establishes results very similar to those used in `sums` as verified consequences of very primitive definitions. Later still, we replaced all the axioms in module `sums` by equivalent lemmas that are proven from those in `sigmaprops`. When time permits, we may make a final revision to these parts of the specification in order to render them suitable for inclusion in a library.

Sums. The module `sums` introduces two higher-order functions, called `sum` ($\sum_{\star 1}^{\star 2}(\star 3)$) and `mean` ($\bigoplus_{\star 1}^{\star 2}(\star 3)$), respectively. Each takes three arguments: the first two are natural numbers, and the third is a function from the natural to the rational numbers. The intended interpretation for `sum` is that it sums the function supplied as its third argument from the value supplied as its first argument to that supplied as its second. That is, in conventional mathematical notation,

$$\texttt{sum(i, j, F)} = \sum_{r=i}^{j} F(r)$$

If $j < i$, the value of sum is intended to be zero. The actual definition of the function sum is accomplished by the axiom sum_ax in terms of the more primitive function sigma which is described in the next subsection.

The axiom mean_ax specifies the (arithmetic) mean function in terms of the sum function in the obvious way. The lemma mean_lemma simply restates the definition of mean directly in terms of the more primitive function sigma. Ten further lemmas then introduce additional properties of the sum and mean functions.

The first, split_sum, states that under suitable conditions a summation from i to j is equal to the sum of two smaller summations: one from i to k, and the other from $k+1$ to j. split_mean, the corresponding result for mean, is proved directly from split_sum.

Lemma sum_bound says that if a function is bounded by a constant x throughout the range i to j, then its summation over that range is bounded by $x \times (j - i + 1)$; the lemma mean_bound states the corresponding result for the mean function and is proved from sum_bound.

The lemmas mean_const and mean_mult simply state that the mean of a constant is that constant, and that the mean of a function multiplied by a constant is the same as the mean of the function multiplied by the constant. Mean_sum and mean_diff state that the mean of the sum or difference of two functions are equal to the sum or difference of the means. Abs_mean states that the absolute value of a mean is less than or equal to the mean of the absolute values. Finally, rearrange_sum states a simple property that is needed in module summations.

The lemmas in module sums are derived from similar results stated for the more primitive sigma function in the module sigmaprops, which is described next.

Sigmaprops. The module sigmaprops introduces a function sigma ($\sigma(\star 1, \star 2, \star 3)$) similar to sum described above. The significant difference, however, is that whereas sum(i, j, F) is intended to denote the sum of F from i to j, σ(i, n, F) is intended to denote the sum of F from i to i + n - 1 (i.e., the sum of n terms).

Sigma is defined by the recursive definition sigma_ax and seven lemmas concerning this function are then stated and proved. The names used for the lemmas are in correspondence with those used for the lemmas in sums:

for example, `split_sigma` in `sigmaprops` corresponds to `split_sum` and `split_mean` in `sums`. The proofs in `sigmaprops` mostly use induction; the induction schemes employed are from the module `natinduction`.

Some of the proofs in `sigmaprops` use a function `revsigma` which is defined like `sigma`, but with the recursion going in the opposite direction. A lemma called `sigma_rev` proves that these two functions are extensionally equal. A second function, called `bounded`, also used internally by `sigmaprops` is introduced and defined by the axiom `bounded_ax`. Since they are used only by the proofs in `sigmaprops`, it might be preferable if the declarations of `revsigma` and `bounded`, together with the axioms that define these functions, were placed in the proof part of the module, rather than the theory part. However, EHDM does not allow axiom declarations in the proof section of a module. (Additional axioms change the theory, which is supposed to be specified by the theory part.) The definitions for `revsigma` and `bounded` could be moved to the proof section only if they were declared as formulas; the proof chain checker would then report a dependency on unproved formulas. A planned extension of the language by a facility for defining auxiliary concepts will solve this dilemma.

4.2.2 Specification Modules

The specification of the Interactive Convergence Clock Synchronization Algorithm is performed in three modules described below.

4.2.2.1 Time

The module `time` is the first one that introduces concepts directly concerned with the Interactive Convergence Clock Synchronization Algorithm. It introduces `clocktime`, `realtime` and `period` as types, and establishes the rationals as the interpretation of the first two, and the naturals as the interpretation of the third. `R`, `S`, and `T_ZERO` (T^0) are introduced as constants of type `clocktime`, and then the functions `T_sup` ($T^{(\star 1)}$), `in_R_interval` ($\star 1 \in R^{(\star 2)}$), and `in_S_interval` ($\star 1 \in S^{(\star 2)}$) are introduced and defined (by the axioms `T_sup_ax`, `Rdef`, and `Sdef`) in the obvious way.

The constraint C1 (`R >= 3 * S`) is defined here, and also the axioms `posR` and `posS` which assert that `R` and `S` are both greater than zero. Several straightforward lemmas are stated and proved.

4.2.2.2 Clocks

The module `clocks` introduces proc (short for processor) as a type interpreted by the naturals, and introduces the clock, correction, adjusted-value, and logical clock functions: `clock` ($c_{\star 1}(\star 2)$), `Corr` ($C_{\star 1}^{(\star 2)}$), `adjusted` ($A_{\star 1}^{(\star 2)}(\star 3)$), and `rt` ($c_{\star 1}^{(\star 2)}(\star 3)$), respectively. The third of these is given an interpretation in terms of the second. The fourth is defined axiomatically (so that we can control its application) in terms of the first and third.

Next, the drift rate `rho` (ρ) is introduced as a constant of type rational number, together with the predicate `goodclock`. The intention is that `goodclock(p, T1, T2)` will be true when processor p is a good clock in the clock time interval `[T1, T2]`. This is specified in the axiom gc_ax. Finally, the predicate `nonfaulty` is introduced and the assumption A1 is stated. Whereas the informal statement of A1 says that if p is nonfaulty through period i, then (this implies that) p has a good clock during the corresponding interval, the formal definition uses equivalence instead of implication. This is necessary because we will later need to prove that if p is nonfaulty through period $i+1$, then it is also nonfaulty through period i.

Our definition of `goodclock` implies that a good clock is strict monotonic increasing. This fact is stated as the Theorem `monotonicity` and proved in the proof part of module `clocks`.

4.2.2.3 Algorithm

The heart of the Interactive Convergence Clock Synchronization Algorithm is defined in the module `algorithm`. We introduce `m` and `n` as constants of type `proc`, and assert that n is nonzero (axiom C0_a) and that `0 <= m < n` (axiom C0_b). The constants `eps` (ϵ), `delta0` (δ_0), `delta` (δ), and `Delta` (Δ) are introduced and the constraints C2 to C6 are stated. The constraint that `Delta` be strictly positive is also stated (as axiom C0_c).

Next, the functions `Delta1` ($\Delta_{\star 1}^{(\star 1)}$), `Delta2` ($\Delta_{\star 1,\star 2}^{(\star 3)}$), and `D2bar` ($\bar{\Delta}_{\star 1,\star 2}^{(\star 3)}$) are introduced, and the Interactive Convergence Clock Synchronization Algorithm itself is specified in the three axioms `Alg1`, `Alg2`, and `Alg3`.

The clock synchronization conditions are specified next. First, we define a function skew: `skew(p, q, T, i)` is the skew between the logical clocks of processors p and q in period i at clock time T (i.e., $|c_p^{(i)}(T) - c_q^{(i)}(T)|$). In the traditional mathematical presentation, we identified S1 with the requirement that the skew between nonfaulty processors should always be less than δ. However, we also need to consider the condition under which this bound should hold—namely that there should be at most m faulty processors. We regard this condition as the antecedent to S1 and identify it with the predicate `S1A`; the bound on the skew between the clocks of nonfaulty processors we consider the consequent of S1 and identify it with the predicate `S1C`. The axiom `S1Cdef` states the bound on the acceptable skew between nonfaulty processors p and q in period i, while the axiom `S1Adef` states the requirement that there should be at least $m - n$ processors nonfaulty through that period. The specification of this last requirement:

```
(FORALL r: (m+1 <= r AND r <= n) IMPLIES nonfaulty(r, i))
```

assumes that it is those processors numbered $m + 1 \ldots n$ that are the nonfaulty ones. Clearly there is no loss of generality in this.

The clock synchronization condition S2, which is identified with the predicate `S2`, is defined in the axiom `S2_ax`.

Finally, the two theorems which assert, respectively, S1A $\supset$ S1C and S2 are defined. The proof of the latter is simple and is performed directly in the proof part of the module `algorithm`.

4.2.3 Proof Modules

The proof of `Theorem_2` (the Interactive Convergence Clock Synchronization Algorithm maintains the clock synchronization condition S2) is provided directly in the module `algorithm`. The proof of `Theorem_1` (the Algorithm maintains clock synchronization condition S1) spans 10 modules that are described below.

4.2.3.1 Clockprops

The module `clockprops` is chiefly concerned with establishing some bounds on $A_p^{(i)}(T + \Pi)$ that are needed to establish Lemma 2. These bounds are stated as the lemmas `upper_bound`, `lower_bound`, and `lower_bound2`. A subsidiary lemma called `adj_always_pos` is also stated; it is used in the proof of `lower_bound`, which in turn is used to establish `lower_bound2`. The proof of `adj_always_pos` itself requires an induction. The proof of `upper_bound`, on the other hand, is straightforward.

The two lemmas `nonfx` and `S1A_lemma` complete the module `clockprops`. The first states that if a module is nonfaulty through period $i + 1$, then it is certainly nonfaulty through period i. This is established as a consequence of A1 and the definition of a good clock (`gc_ax`). `S1A_lemma` states the corresponding result for S1A, and is proved directly from `nonfx`.

4.2.3.2 Lemmas 1 to 6

These follow exactly the structure and naming described in Chapter 2. Indeed, the description in that chapter was derived directly from the formal specifications and proofs in these six modules.

Each lemma is stated and proved in a module with the appropriate name. The result called Sublemma A is to be found as a subsidiary lemma `sublemma_A` in the module `lemma6`.

4.2.3.3 Summations

The module `summations` is concerned with establishing the inductive step needed in the proof of `Theorem_1`. This result is stated as the lemma called `culmination`, and is proved from a series of intermediate lemmas named `l1` through `l5`.

The lemma `l1` connects the main term in the conclusion of Lemma 6 with the averaging step performed by the Algorithm (specified in `Alg2`). Lemma `l2` splits the summation implicitly involved in `l1` into two smaller summations—one over the faulty processors and one over the nonfaulty ones. Lemma `l3` uses Lemma 5 to obtain a bound on the sum of the errors introduced by the faulty processors; a subsidiary lemma called `bound_faulty` is used in the process.

Lemma `l4` uses Lemma 4 to obtain a bound on the sum of the errors introduced by the nonfaulty processors; a subsidiary lemma called `bound_nonfaulty` is used in the process. The proof of this lemma uses `Theorem_1`; we discuss this below (on Page 60).

Lemma `l5` simply combines lemmas `l2`, `l3` and `l4`; the `culmination` lemma is proved by combining `l5` with Lemma 6.

4.2.3.4 Juggle

The module `juggle` proves the lemma `rearrange_delta`. This result is a straightforward algebraic manipulation and is quite simple to do by hand. Its proof in EHDM, however, is rather tedious. The source of the difficulty is the appearance of nonlinear multiplication. As explained earlier, the EHDM ground prover is incomplete in the presence of nonlinear arithmetic. Consequently, the module `juggle` contains several lemmas that essentially switch between the interpreted multiplication symbol and the uninterpreted `mult` function in order to establish some simple arithmetic identities. The main proof is then accomplished in 6 steps using intermediate lemmas named step1 through step5.

4.2.3.5 Main

The module main provides the proof of Theorem_1. It uses the induction scheme mod_induction from the module natinduction, with the main work for the inductive step provided by the culmination lemma from module summations. The rather grotesque arithmetic manipulation required to complete the proof is provided by the lemma rearrange_delta from the module juggle.

As noted above, the inductive proof of Theorem_1 depends on the lemma culmination from the module summations. The proof of culmination depends on the lemma bound_nonfaulty, whose own proof depends on Theorem_1. Thus, there is a potential circularity in our proof of the theorem—which is indeed detected by the EHDM proof chain checker. In fact, this circularity is apparent, rather than real, as it occurs in the context of an inductive proof, in which the theorem is used for i in the part of the proof that extends it to i + 1. We are working towards constructing a proof description that reflects this induction step more straightforwardly.

4.3 Statistics and Observations

The specification and verification described here was performed using EHDM Version 4.1.4 running on a Sun workstation. EHDM is written in Common Lisp; the current version for Sun workstations uses the Lucid 2.1 Common Lisp implementation. The particular workstation used for this exercise was a Sun 3/75 with 8 Mbytes of real memory and 56.5 Mbytes of swap space on a lightly loaded Sun 3/160 file server with Fujitsu Eagle and Super-Eagle disk drives and slow Xylogics controllers.

The specifications described here occupy 20 modules, comprising about 1,550 (nonblank) lines of EHDM. There are 166 proofs in the full specification and it takes about an hour to prove them all (a little under 18 seconds each, on average). It is hard to obtain accurate timing for individual proofs, since the occurrence of garbage collection introduces tremendous variability—however, the worst case seems to be about a minute and a half.

The proofs in each module are summarized in the table below, which reproduces part of the output from the EHDM "proveall" command.

Module absolutes:	12 proofs
Module algorithm:	5 proofs
Module arithmetics:	25 proofs
Module clockprops:	12 proofs
Module clocks:	2 proofs
Module functionprops:	no proofs
Module juggle:	14 proofs
Module lemma1:	1 proof
Module lemma2:	5 proofs
Module lemma3:	1 proof
Module lemma4:	6 proofs
Module lemma5:	3 proofs
Module lemma6:	4 proofs
Module main:	3 proofs
Module natinduction:	5 proofs
Module natprops:	7 proofs
Module sigmaprops:	28 proofs
Module summations:	9 proofs
Module sums:	19 proofs
Module time:	6 proofs

Table 4.1: Proof Summaries for EHDM Modules

Of course, the raw statistics of CPU time and numbers of proofs and lines of specification text are among the most superficial measures one can provide for a formal specification and verification. More interesting are the questions of how much human effort was required, whether the benefits of the exercise could have been obtained more cheaply by other techniques, and whether the particular specification and verification techniques and tools used were a help or a hindrance to the effort.

Unfortunately, we did not accurately record the human effort expended on this exercise, so the following account relies on memory. Our first attempt to perform the verification occupied a week, with both of us devoting about three-quarters of our time to the effort. One of us broke the published proof of Lamport and Melliar-Smith down into elementary steps, while the other encoded these in EHDM and persuaded the theorem prover to accept the proofs. At this point we had caught the typographical errors in Lemmas 2 and 4, and had proofs of Lemmas 1, 3, 4, and 5—but Lemma 2 was essentially taken as an axiom. Approximate equality and inequalities were used freely at this stage, although several of the formulas needed were mentally flagged as suspicious.

It was when we attempted to establish Lemma 2 as a consequence of a more primitive axiomatization of the properties of good clocks that we first came to suspect that the published proof was flawed. Once we had satisfied ourselves that this was indeed so, we became more critical of other aspects of the published proof and checked all the formulas (treated as axioms at this stage) needed to support the use of approximations. This led us to fully recognize the flawed character of the proofs for Lemma 4 and the main Theorem.

Until this point we had merely been attempting to mechanize the published proof, and had not really internalized that proof, nor tried independently to re-create it. As a result of discovering flaws in the published proof, our interest in the verification exercise increased considerably and we sought not only to eliminate the use of approximations, but to simplify and systematize the proof as well. The elimination of approximations was accomplished quite easily, and simplification of the proofs of Lemmas 1, 3, 4 and 5 was achieved by more systematic use of the arithmetic "rearrangement" identities (e.g., $x = (u - v) + (v - w) - (u - [w + x])$ used in Lemma 1). All this work was done by hand, and only cast into EHDM and mechanically verified towards the end.

Our restructuring and better understanding of the proofs reduced the EHDM proof declarations for Lemmas 3 and 4 to between a half and a third of their previous lengths (elimination of the unnecessary Π from Lemma 3 also contributed to the simplification of its proof). It was during this stage of the mechanical verification, that we recognized the need for several variants on Lemma 2, and for modifications to Assumption A2. This stage of the effort (including the manual reformulation of the proof, as well as its mechanization) consumed about three man-weeks.

Next we mechanized the proof of the main theorem, developing the modules `lemma6`, `summations`, and `main`. The formulas in module `sums` were developed while doing the proofs in module `summations` and were used as axioms at this stage—which consumed about two-man weeks.

Finally, we began to put the whole verification together and to prepare this document. We developed the module `sigmaprops` and used it to prove the previously unproved formulas in module `sums`. We discovered several minor flaws in the statements of those formulas while performing their proofs. As we began to describe and document our specifications and proofs, we filled in missing fragments (e.g., the module `juggle`, which took a man-day to create), and continually revised the modules of the supporting theories in order to simplify and systematize the axiomatic basis on which the whole verification depends. This process proceeded in parallel with the preparation of this report—both activities together consumed about two man-months.

We have described the chronology of this effort in some detail to illustrate the following points:

- The mechanical verification was interleaved with pencil and paper mathematics, and each activity stimulated the other. We expand on this below, but the essential point is that formal specification and verification assists rather than replaces human thought and scrutiny.

- A substantial portion of the time devoted to the mechanical verification was expended on the supporting theories. As formal verification becomes more widely practiced, we would expect libraries of such theories to become established, so that later efforts can concentrate their efforts on the problem of real interest.[11] If we neglect the effort spent on the supporting theories, then the time required to perform the mechanical verification was of a similar order to that required to prepare an adequately detailed "journal-level" description and proof for human consumption (i.e., the first 3 Chapters of this report).

---

[11] EHDM provides linguistic and system support (in the form of module parameterization, and a mechanism for managing module libraries, respectively) that are explicitly intended for the support of reusable specifications.

- "High-level" axioms are almost always wrong! The main benefit of mechanical verification is the extreme rigor of the scrutiny to which proofs are subjected. This benefit is subverted if axioms are introduced casually. It was not until we attempted to build our proofs on the most basic definition of a good clock, and seriously scrutinized the lemmas required of the approximation operators, that we began to discover the flaws in the published proof. Similarly, our first-cut axiomatizations of the summation operators were flawed (typically at boundary cases). Others who have undertaken formal specification and verification exercises have privately reported similar experiences.

  Our current verification depends on 47 axioms. Of these, 29 (6 in module `time`, 6 in `clocks` and 17 in `algorithm`) define the concepts, constraints, and algorithm of direct interest. The other 18 introduce supporting concepts (e.g., summation) or properties of arithmetic beyond those built into the system (i.e., some of the properties of division and multiplication). We spent a great deal of effort reducing the number and simplifying the content of these 18 supporting axioms and we believe that they correspond to conventional interpretations of the concepts concerned. Similarly, we believe that the 29 axioms underlying our development of the Interactive Convergence Clock Synchronization Algorithm are a simple and near-minimal foundation on which to construct the definition and analysis of this algorithm.

  It is always necessary to scrutinize axioms with great care, and we believe that this can best be accomplished if the axioms are as simple and as few as feasible. Our experience suggests that it can be very time-consuming to pare away at the axiomatic foundation of a proof, but that it is very worthwhile to do so.

It is difficult to answer the question whether the flaws we found in the published analysis of the Interactive Convergence Clock Synchronization Algorithm could have been discovered more easily by other methods. Once the flaws are known, they are easy to describe and their presence in the published proof is almost painfully obvious. Nonetheless, as far as we know, these flaws were not discovered previously. The reputation of the journal in which the paper was published, and of its authors, may have caused some to assume that the proof "must be right" without further scrutiny, and may have stilled any doubts in the minds of those who examined the proof in sufficient detail to become concerned by some of its details. Some who scrutinized the proof with great care decided that it would be easier to develop their own analysis than to persuade themselves of the veracity of the original.[12]

The root difficulty, we believe, lies in the fact that the proof in [11], though neither mathematically deep nor intrinsically interesting, is astonishingly intricate in its details. The analysis of many algorithms, computer programs, and similar artifacts shares this characteristic—and renders the standard "mathematical demonstration" (which forms the basis for the consensus model of classical mathematics) unreliable in these contexts.

The only reliable method for conducting such highly intricate analyses is, we believe, a strictly formal one—one in which the "symbols do the work" just as they do in arithmetic and other detailed calculations. Formal calculations can introduce their own class of errors, but their formal character means that they can be checked easily (if tediously) by others. Once the decision to use a strict formalism has been taken, the additional cost of subjecting the calculations to *mechanical* checking is not great—providing the formal system and notation used by the machine does not differ too much from that used by the hand and brain.

We found that EHDM served us very well from this perspective. Because EHDM uses a standard logic (predicate calculus) with all the usual quantifiers and connectives, transliterating from the notation of Lamport and Melliar-Smith into the specification language of EHDM was straightforward. Automation of the reverse translation (by the LaTeX-printer) enabled us to do most of our work and thinking using compact and familiar notation and thereby contributed greatly to our productivity. The higher-order capabilities of EHDM allowed us to define the summation and averaging operators very straightforwardly and also enabled us to tailor induction schemes appropriately.

The arithmetic decision procedures of EHDM were of immense value in the formal verification. We doubt that verification environments lacking such decision procedures could accomplish the work described here without unreasonable effort. Most of the really tedious theorem proving that we undertook arose at the boundary of the arithmetic decision procedures (i.e., in dealing with division and non-linear multiplication). There is no perfect solution to these difficulties (the theories concerned are undecidable), but a better integration of decision procedures, incomplete heuristics, and manual guidance is both possible and desirable—and will be pursued in further developments of EHDM. We found the basic theorem-proving paradigm of

[12]Fred Schneider has told us that this was one of the motivations behind [15].

EHDM straightforward and adequate for its purpose (though others, especially novices, might not agree). The correspondence between the information in an EHDM "prove" declaration and that required for a journal-level proof description is quite close. Naturally, increased automation of details (for example, use of term rewriting to mechanize equational theories, and automatic discovery of substitution instances)[13] would be welcome, but we did not find theorem proving to be a bottleneck. (Discovering the correct theorems to prove was the bottleneck.)

The module structure supported by the EHDM specification language and its support environment simplified the task of managing and comprehending a formal development that eventually became quite large, and enabled us to keep track of undischarged proof obligations. The latter service was particularly valuable, due to the way in which our formal specification and verification were developed. Our approach was very much top-down: we introduced lemmas whenever it was convenient to do so, and worried about proving them later. We may have carried this approach a little too far in the early stages (i.e., we did not examine the content of our lemmas with sufficient care), but we did not know at that period whether our attempt to mechanically verify the algorithm would be successful[14] and we were anxious to explore the more obviously difficult parts first.

Overall, we did not find the formal specification and mechanical verification of the Interactive Convergence Clock Synchronization Algorithm particularly demanding. The main difficulty was the sheer intricacy of the argument, and we found the discipline of formal specification and verification to be a help, rather than a hindrance, in finally mastering this complexity.

We found that EHDM served us reasonably well; we do not know whether other specification and verification environments would have fared as well or better. Understanding the practical benefits and limitations of different approaches to formal specification and mechanical theorem proving is necessary for sensible further development of verification environments. Consequently, we invite the developers and users of other verification systems to repeat the experiment described here. We suggest that the Interactive Convergence Clock Synchronization Algorithm is a paradigmatic example of a problem where formal verification can show its value and a verification system can demonstrate its capabilities: it is a "real" rather than an artifi-

[13]The *instantiator* of EHDM accomplishes both of these tasks very effectively for proofs in pure predicate calculus, but is much less useful when arithmetic is employed extensively.

[14]The algorithm (or rather an implementation of it) had been asserted to be "probably beyond the ability of any current mechanical verifier" [2, page 9].

cial problem, its verification is large enough to be challenging without being overwhelming, it requires a couple of fairly interesting supporting theories, and its proofs are quite intricate and varied.

Chapter 5

Conclusions

"The virtue of a logical proof is not that it compels belief but that it suggests doubts." [10, page 48]

Verification does not prove programs "correct"; it merely establishes consistency between one description of a system and another. The extent to which such consistency can be equated with correctness depends on the extent to which one of the descriptions accurately states all the properties required of the system, on the extent to which the other accurately and completely describes its actual behavior, and on the extent to which the demonstration of consistency between these two descriptions is performed without error.

In practice, all three of these limitations on "correctness" pose significant challenges. The behavior of the actual system will depend on physical processes that may not admit completely accurate descriptions, or that may be subject to random effects, while the properties required of the system may not be fully understood, let alone fully recorded in its specification. And demonstration of consistency between the two descriptions of the system will be subject to the errors attendant upon any human enterprise. *Formal* specification and verification attempts to control and delimit some of the difficulties associated with verification; the use of formal *specifications* can at least provide precise and unambiguous descriptions of the intended behavior of the system—the questions remain whether these descriptions correctly capture what is really required, or what the behavior of the system really is, but at least the doubt about what the descriptions themselves *mean* is removed. Formal *verification* attempts to put the demonstration of consistency between two system descriptions onto a more reliable basis by making it a mathematical—indeed, calculational—activity that can be checked by a mechanical theorem prover. Of course, the validity of this approach depends on the extent to which the semantics of the specification language are correctly implemented by its support environment, and on the correctness of the mechanical theorem prover. These represent significant challenges, but they are at least more sharply posed than the problems with which we began.

Formal verification is no more than a formalization of one of the components in the widely practiced software quality assurance process called Verification and Validation (V&V). Validation (testing), the other component to this process, is not made redundant or unnecessary by formalizing the verification component. Indeed, formal verification can help clarify the assumptions that should be validated by explicit testing.

The opening paragraphs of the introductory document to EHDM [1] make our own attitude clear:

> "Writing formal specifications and performing verifications that really mean something is a serious engineering endeavor. Formal specification and verification are often recommended for systems that perform functions critical to human safety or national security, but it must be understood that formal analysis alone cannot provide assurance that a system is fit for such a critical function. Certifying a system as "safe" or "secure" is a responsibility that calls for the highest technical experience, skill, and judgment—and the consideration of multiple forms of evidence. Other important forms of analysis and evidence that should be considered for critical systems are systematic testing, quantitative reliability measurement, software safety analysis, and risk assessment. Also, it should be understood that the purpose of formal verification is not to provide unequivocal evidence that some aspects of a system design and implementation are "correct," but to help *you* the user convince yourself of that fact; the verification system does not act as an oracle, but as an implacable skeptic that insists on you explaining and justifying every step of your reasoning—thereby helping you to reach a deeper and more complete understanding of your system."

The opponents to formal verification [7, 9] ignore caveats such as those expressed above (which are similar to those expressed by all serious proponents of formal verification) and perform a straw man attack in which verification is set up as an unequivocal demonstration of correctness, and in which intelligent human participation is minimized in favor of an omniscient mechanical verifier. For example, De Millo, Lipton and Perlis [7] claim that:

> "The scenario envisaged by the proponents of verification goes something like this: the programmer inserts his 300-line input/output package into the verifier. Several hours later, he returns. There is his 20,000-line verification and the message 'VERIFIED'."

This is parody. In a paper published several years earlier [19], von Henke and Luckham indicated the true nature of the scenario envisioned by the proponents of verification when they wrote:

> "The goal of practical usefulness does not imply that the verification of a program must be made independent of creative effort on the part of the programmer ...such a requirement is utterly unrealistic."

The thrust of De Millo, Lipton and Perlis' argument is that formal verification moves responsibility away from the "social process" that involves human scrutiny, towards a mechanical process with little human participation. In reality, a verification system assists the human *user* to develop a convincing argument for the correctness of his program by acting as an implacably skeptical colleague who demands that all assumptions be stated and all claims justified. The requirement to explicate and formalize what would otherwise be unexamined assumptions is especially valuable. Shankar [16], for example, observes:

> "The utility of proof-checkers is in clarifying proofs rather than in validating assertions. The commonly held view of proof-checkers is that they do more of the latter than the former. In fact, very little of the time spent with a proof-checker is actually spent proving theorems. Much of it goes into finding counterexamples, correcting mistakes, and refining arguments, definitions, or statements of theorems. A useful automatic proof-checker plays the role of a devil's advocate for this purpose."

This perspective on mechanical theorem proving is very similar to that developed by Lakatos [10] for the role of proof (not just mechanical theorem proving) in mathematics. Crudely, this view is that successful completion is among the least interesting and useful outcomes of a proof attempt; the real benefit comes from failed proof attempts, since these challenge us to revise our hypotheses, sharpen our statements, and achieve a deeper understanding of our problem.

Our own experience with the verification of the Interactive Convergence Clock Synchronization Algorithm supports this view. Most of our time was spent in trying to prove theorems and lemmas that turned out to be false, in coming to understand why they were false, and in revising their statements, or those of supporting lemmas and assumptions. The difficulties we encountered were consequences of genuine technical flaws in the previously published analysis of the Algorithm [11], and we consider the main benefit of this exercise to be the identification and correction of those flaws. The corrections led us to eliminate the use of approximations, thereby allowing precise statements of the constraints on the values of the parameters to the Algorithm, and led us to modify one of the assumptions (A2) underlying the Algorithm, thereby changing its external specification slightly. Our corrections to the statements and proofs of some of the lemmas led us to a more uniform method for doing those proofs. When reflected back into a traditional mathematical presentation (given in Chapter 2), we consider the result to be an analysis that is not only more precise, but simpler and easier to follow than the original.

Thus, we believe that a significant benefit from our *formal* verification is an improved *informal* argument for the correctness of the Interactive Convergence Clock Synchronization Algorithm. We hope that anyone contemplating using the Algorithm will study our presentation and will convince *themselves* of the correctness of the Algorithm and of the appropriateness of the assumptions (and of the ability of their implementation to satisfy those assumptions).

Our formal verification does not usurp the "social process" in which De Millo, Lipton and Perlis place their faith, but should serve to shift its focus from details to fundamentals. We note that the "social process" apparently failed to discover the flaws that we have noted in the main theorem concerning the Interactive Convergence Clock Synchronization Algorithm, and in four of its five lemmas. This is not surprising: the standards of rigor and formality in the normal "mathematical demonstration" are simply inadequate to the intricacy and detail required for the analysis of many algorithms and programs. Mechanically checked verification provides valuable supplementary scrutiny and evidence in these cases.

The extent to which our verification provides a formal *guarantee* of the correctness of the Interactive Convergence Clock Synchronization Algorithm is compromised by the fact that the representation of the problem is somewhat abstracted from reality. The aspect of the representation of the clock synchronization problem that causes us most concern is the basic definition of a clock. Real clocks increment in discrete "ticks" whose magnitude may be quite large compared with some of the other parameters in the system. Using the rationals as the interpretation of clock time is therefore unrealistic, as is the requirement that a good clock should be a strict monotonic function. Schneider [15] presents a model which treats these aspects more realistically; formalizing this approach provides an interesting challenge for the future.

A further challenge will be to formalize and verify an *implementation* of the Interactive Convergence Clock Synchronization Algorithm—so far, we have simply verified properties of the algorithm itself. Our current work is addressing these challenges; we expect to report our results in early 1990.

Bibliography

[1] *Introduction to* EHDM. Computer Science Laboratory, SRI International, Menlo Park, CA 94025, September 28, 1988.

[2] NASA Conference Publication 2377. *Peer Review of a Formal Verification/Design Proof Methodology*, July 1983.

[3] Peter B. Andrews. *An Introduction to Logic and Type Theory: To Truth through Proof.* Academic press, 1986.

[4] Ricky W. Butler. *A Survey of Provably Correct Fault-Tolerant Clock Synchronization Techniques.* Technical Report TM-100553, NASA Langley Research Center, February 1988.

[5] Ricky W. Butler, Daniel L. Palumbo, and Sally C. Johnson. Application of a clock synchronization validation methodology to the SIFT computer system. In *Digest of Papers, FTCS 15*, pages 194–199, IEEE Computer Society, Ann Arbor, MI., June 1985.

[6] Flaviu Cristian. *Probabilistic Clock Synchronization.* Technical Report RJ 6432, IBM Almaden Research Center, San Jose, CA., September 1988.

[7] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[8] D. Dolev, J.Y. Halpern, and H.R. Strong. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of 16th Annual ACM Symposium on Theory of Computing*, pages 504–511, Washington, D.C., April 1984.

[9] James H. Fetzer. Program verification: the very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

[10] Imre Lakatos. *Proofs and Refutations.* Cambridge University Press, Cambridge, England, 1976.

[11] L. Lamport and P.M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[12] Leslie Lamport. *Synchronizing Time Servers.* Technical Report 18, DEC Systems Research Center, Palo Alto, CA., June 1987.

[13] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming.* Volume 1, Addison-Wesley, 1985.

[14] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming.* Volume 2, Addison-Wesley, 1988.

[15] Fred B. Schneider. *Understanding Protocols for Byzantine Clock Synchronization.* Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, August 1987.

[16] N. Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the ACM*, 35(3):475–522, July 1988.

[17] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[18] F.W. von Henke, J.S. Crow, R. Lee, J.M. Rushby, and R.A. Whitehurst. The EHDM verification environment: an overview. In *Proceedings 11th National Computer Security Conference*, pages 147–155, NBS/NCSC, Baltimore, MD., October 1988.

[19] F.W. von Henke and D.C. Luckham. A methodology for verifying programs. In *Proceedings, International Conference on Reliable Software*, pages 156–164, IEEE Computer Society, Los Angeles, CA., April 1975.

Appendix A

Cross-Reference Listing

This Appendix provides two cross-reference tables to assist in reading and navigating the EHDM specifications that follow. The first provides the translations used between EHDM identifiers and the symbols used in the traditional mathematical presentation and in the LaTeX-printed version of the specifications. The second table provides a cross-reference listing to the identifiers declared in the EHDM specification.

Identifier	Translation
abs	$\|\star 1\|$
adjusted	$A_{\star 1}^{(\star 2)}(\star 3)$
clock	$c_{\star 1}(\star 2)$
Corr	$C_{\star 1}^{(\star 2)}$
D2bar	$\bar{\Delta}_{\star 1,\star 2}^{(\star 3)}$
Delta	Δ
delta	δ
delta0	δ_0
Delta1	$\Delta_{\star 1}^{(\star 2)}$
Delta2	$\Delta_{\star 1,\star 2}^{(\star 3)}$
eps	ϵ
Gamma	Γ
half	$\frac{\star 1}{2}$
in_R_interval	$\star 1 \in R^{(\star 2)}$
in_S_interval	$\star 1 \in S^{(\star 2)}$
mean	$\bigoplus_{\star 1}^{\star 2}(\star 3)$
mult	$\star 1 \times \star 2$
PHI	Φ
PI	Π
rho	ρ
rt	$c_{\star 1}^{(\star 2)}(\star 3)$
Sigma	Σ
sigma	$\sigma(\star 1, \star 2, \star 3)$
sum	$\sum_{\star 1}^{\star 2}(\star 3)$
T0	T_0
T1	T_1
t1	t_1
T2	T_2
t2	t_2
TN	T_N
T_sup	$T^{(\star 1)}$
T_ZERO	T^0

Table A.1: LaTeX-Printer Translations for EHDM Identifiers

Identifier	Type of Declaration	Module where Declared
A0	axiom	algorithm
A1	axiom	clocks
A2	axiom	algorithm
A2_aux	axiom	algorithm
abs	function	absolutes
absolutes	module	absolutes
abs_ax	axiom	absolutes
abs_ax0	lemma	absolutes
abs_ax1	lemma	absolutes
abs_ax2	lemma	absolutes
abs_ax2b	lemma	absolutes
abs_ax2c	lemma	absolutes
abs_ax3	lemma	absolutes
abs_ax4	lemma	absolutes
abs_ax5	lemma	absolutes
abs_ax6	lemma	absolutes
abs_ax7	lemma	absolutes
abs_ax8	lemma	absolutes
abs_div	axiom	absolutes
abs_div2	lemma	arithmetics
abs_div2_proof	prove	arithmetics
abs_mean	lemma	sums
abs_mean_proof	prove	sums
abs_proof0	prove	absolutes
abs_proof1	prove	absolutes
abs_proof2	prove	absolutes
abs_proof2b	prove	absolutes
abs_proof2c	prove	absolutes
abs_proof3	prove	absolutes
abs_proof4	prove	absolutes
abs_proof5	prove	absolutes
abs_proof6	prove	absolutes
abs_proof7	prove	absolutes
abs_proof8	prove	absolutes
abs_sum	lemma	sums
abs_sum_proof	prove	sums
abs_times	axiom	absolutes

Table A.2: Cross-Reference to EHDM Identifiers

Identifier	Type of Declaration	Module where Declared
adjusted	function	clocks
adj_always_pos	lemma	clockprops
adj_pos_proof	prove	clockprops
Alg1	axiom	algorithm
Alg2	axiom	algorithm
Alg3	axiom	algorithm
algorithm	module	algorithm
alt_sb_step_proof	prove	sigmaprops
alt_sigma_bound_step	lemma	sigmaprops
arithmetics	module	arithmetics
basis	lemma	clockprops
basis	lemma	main
basis_proof	prove	clockprops
basis_proof	prove	main
bounded	function	sigmaprops
bounded_ax	axiom	sigmaprops
bounded_lemma	lemma	sigmaprops
bounded_proof	prove	sigmaprops
bounds	lemma	clockprops
bounds_proof	prove	clockprops
bound_faulty	lemma	summations
bound_faulty_proof	prove	summations
bound_nonfaulty	lemma	summations
bound_nonfaulty_proof	prove	summations
C0_a	axiom	algorithm
C0_b	axiom	algorithm
C0_c	axiom	algorithm
C1	axiom	time
C2	axiom	algorithm
C2and3	lemma	algorithm
C2and3_proof	prove	algorithm
C3	axiom	algorithm
C4	axiom	algorithm
C5	axiom	algorithm
C6	axiom	algorithm

Table A.3: Cross-Reference to EHDM Identifiers (Continued)

Identifier	Type of Declaration	Module where Declared
cancellation	lemma	arithmetics
cancellation_mult	lemma	arithmetics
cancellation_mult_proof	prove	arithmetics
cancellation_proof	prove	arithmetics
cancel_mult	lemma	juggle
cancel_mult_proof	prove	juggle
clock	function	clocks
clockdef	axiom	clocks
clockprops	module	clockprops
clocks	module	clocks
clocktime	type	time
clock_proof	prove	algorithm
clock_prop	lemma	algorithm
Corr	function	clocks
Cross	reference	of
culmination	lemma	summations
culm_proof	prove	summations
D2bar	function	algorithm
D2bar_prop	lemma	algorithm
D2bar_prop_proof	prove	algorithm
Delta	const	algorithm
delta	const	algorithm
delta0	const	algorithm
Delta1	function	algorithm
Delta2	function	algorithm
diff	function	natprops
diff1	lemma	natprops
diff1_proof	prove	natprops
diff_ax	axiom	natprops
diff_diff	lemma	natprops
diff_diff_proof	prove	natprops
diff_ineq	lemma	natprops
diff_ineq_proof	prove	natprops
diff_plus	lemma	natprops
diff_plus_proof	prove	natprops
diff_zero	lemma	natprops
diff_zero_proof	prove	natprops

Table A.4: Cross-Reference to EHDM Identifiers (Continued)

Identifier	Type of Declaration	Module where Declared
diminish	lemma	clocks
diminish_proof	prove	clocks
distrib4_div	lemma	juggle
distrib4_div_proof	prove	juggle
distrib6	lemma	juggle
distrib6_div	lemma	juggle
distrib6_div_proof	prove	juggle
distrib6_mult	lemma	juggle
distrib6_mult_proof	prove	juggle
distrib6_proof	prove	juggle
div_distr	lemma	arithmetics
div_distr_proof	prove	arithmetics
div_mon	lemma	arithmetics
div_mon2	lemma	arithmetics
div_mon2_proof	prove	arithmetics
div_mon_proof	prove	arithmetics
div_mult	lemma	arithmetics
div_mult2	lemma	arithmetics
div_mult2_proof	prove	arithmetics
div_mult_proof	prove	arithmetics
div_prod	lemma	arithmetics
div_prod2	lemma	arithmetics
div_prod2_proof	prove	arithmetics
div_prod_proof	prove	arithmetics
div_times	lemma	arithmetics
div_times_proof	prove	arithmetics
eps	const	algorithm
extensionality	axiom	functionprops
final	prove	juggle
functionprops	module	functionprops
gc_ax	axiom	clocks
gc_proof	prove	clockprops
gc_prop	lemma	clockprops
goodclock	function	clocks

Table A.5: Cross-Reference to EHDM Identifiers (Continued)

Identifier	Type of Declaration	Module where Declared
half	function	arithmetics
half2	lemma	arithmetics
half2_proof	prove	arithmetics
half3	lemma	arithmetics
half3_proof	prove	arithmetics
half_ax	axiom	arithmetics
i2R	lemma	clockprops
i2R_proof	prove	clockprops
Identifier	Type	Module
induction	lemma	natinduction
induction1	lemma	natinduction
induction1_proof	prove	natinduction
induction2	axiom	natinduction
induction_m	axiom	natinduction
induction_proof	prove	natinduction
inductive_step	lemma	clockprops
ind_proof	prove	clockprops
ind_proof	prove	main
ind_step	lemma	main
inRS	lemma	time
inRS_proof	prove	time
in_R_interval	function	time
in_S_interval	function	time
in_S_lemma	lemma	time
in_S_proof	prove	time
juggle	module	juggle
l1	lemma	summations
l1_proof	prove	summations
l2	lemma	summations
l2_proof	prove	summations
l3	lemma	summations
l3_proof	prove	summations
l4	lemma	summations
l4_proof	prove	summations
l5	lemma	summations
l5_proof	prove	summations

Table A.6: Cross-Reference to EHDM Identifiers (Continued)

Identifier	Type of Declaration	Module where Declared
lemma1	module	lemma1
lemma1def	lemma	lemma1
lemma1_proof	prove	lemma1
lemma2	module	lemma2
lemma2a	lemma	lemma2
lemma2a_proof	prove	lemma2
lemma2b	lemma	lemma2
lemma2b_proof	prove	lemma2
lemma2c	lemma	lemma2
lemma2c_proof	prove	lemma2
lemma2d	lemma	lemma2
lemma2def	lemma	lemma2
lemma2d_proof	prove	lemma2
lemma2x	lemma	lemma4
lemma2x_proof	prove	lemma4
lemma2_proof	prove	lemma2
lemma3	module	lemma3
lemma3def	lemma	lemma3
lemma3_proof	prove	lemma3
lemma4	module	lemma4
lemma4def	lemma	lemma4
lemma4_proof	prove	lemma4
lemma5	module	lemma5
lemma5def	lemma	lemma5
lemma5proof	prove	lemma5
lemma6	module	lemma6
lemma6def	lemma	lemma6
lemma6_proof	prove	lemma6
lower_bound	lemma	clockprops
lower_bound2	lemma	clockprops
lower_bound2_proof	prove	clockprops
lower_bound_proof	prove	clockprops
m	const	algorithm
main	module	main

Table A.7: Cross-Reference to EHDM Identifiers (Continued)

Identifier	Type of Declaration	Module where Declared
mean	function	sums
mean_ax	axiom	sums
mean_bound	lemma	sums
mean_bound_proof	prove	sums
mean_const	lemma	sums
mean_const_proof	prove	sums
mean_diff	lemma	sums
mean_diff_proof	prove	sums
mean_lemma	lemma	sums
mean_lemma_proof	prove	sums
mean_mult	lemma	sums
mean_mult_proof	prove	sums
mean_sum	lemma	sums
mean_sum_proof	prove	sums
mod_induction	lemma	natinduction
mod_induction1	lemma	natinduction
mod_induction1_proof	prove	natinduction
mod_induction_m	lemma	natinduction
mod_induction_proof	prove	natinduction
mod_m_proof	prove	natinduction
mod_sigma_mult	lemma	sigmaprops
mod_sigma_mult_proof	prove	sigmaprops
monoproof	prove	clocks
monotonicity	theorem	clocks
mult	function	arithmetics
mult0	lemma	arithmetics
mult0_proof	prove	arithmetics
mult1	axiom	arithmetics
mult2	lemma	arithmetics
mult2_proof	prove	arithmetics
mult3	lemma	arithmetics
mult3_proof	prove	arithmetics
mult4	lemma	arithmetics
mult4_proof	prove	arithmetics

Table A.8: Cross-Reference to EHDM Identifiers (Continued)

Identifier	Type of Declaration	Module where Declared
mult_ax	axiom	arithmetics
mult_div	lemma	arithmetics
mult_div_proof	prove	arithmetics
mult_ineq1	lemma	juggle
mult_ineq1_proof	prove	juggle
mult_ineq2	lemma	juggle
mult_ineq2_proof	prove	juggle
mult_mon	axiom	arithmetics
mult_mon2	lemma	arithmetics
mult_mon2_proof	prove	arithmetics
n	const	algorithm
natinduction	module	natinduction
natpos	axiom	natprops
natprops	module	natprops
nonfaulty	function	clocks
nonfx	lemma	clockprops
nonfx_proof	prove	clockprops
period	type	time
posR	axiom	time
posS	axiom	time
pos_abs	lemma	absolutes
pos_abs_proof	prove	absolutes
pred	function	natprops
pred_ax	axiom	natprops
pred_diff	lemma	natprops
pred_diff_proof	prove	natprops
pred_lemma	lemma	natprops
pred_lemma_proof	prove	natprops
proc	type	clocks
quotient_ax	axiom	arithmetics
quotient_ax1	axiom	arithmetics
quotient_ax2	axiom	arithmetics
quotient_mult	lemma	arithmetics
quotient_mult_proof	prove	arithmetics

Table A.9: Cross-Reference to EHDM Identifiers (Continued)

Identifier	Type of Declaration	Module where Declared
R	const	time
Rdef	axiom	time
realtime	type	time
rearrange	lemma	arithmetics
rearrange1	lemma	arithmetics
rearrange1	lemma	lemma4
rearrange1	lemma	lemma5
rearrange1_proof	prove	arithmetics
rearrange1_proof	prove	lemma4
rearrange1_proof	prove	lemma5
rearrange2	lemma	arithmetics
rearrange2	lemma	lemma4
rearrange2	lemma	lemma5
rearrange2_proof	prove	arithmetics
rearrange2_proof	prove	lemma4
rearrange2_proof	prove	lemma5
rearrange3	lemma	lemma4
rearrange3_proof	prove	lemma4
rearrange_alt	lemma	arithmetics
rearrange_alt_proof	prove	arithmetics
rearrange_delta	lemma	juggle
rearrange_proof	prove	arithmetics
rearrange_sub	lemma	sums
rearrange_sub_proof	prove	sums
rearrange_sum	lemma	sums
rearrange_sum_proof	prove	sums
reciprocal	lemma	juggle
reciprocal_proof	prove	juggle
revsigma	function	sigmaprops
revsigma_ax	axiom	sigmaprops
rho	const	clocks
rho_pos	axiom	clocks
rho_small	axiom	clocks
rt	function	clocks
S	const	time

Table A.10: Cross-Reference to EHDM Identifiers (Continued)

Identifier	Type of Declaration	Module where Declared
S1A	function	algorithm
S1Adef	axiom	algorithm
S1A_lemma	lemma	clockprops
S1A_lemma_proof	prove	clockprops
s1b_proof	prove	sigmaprops
S1C	function	algorithm
S1Cdef	axiom	algorithm
S1C_lemma	lemma	algorithm
S1C_lemma_proof	prove	algorithm
s1s_proof	prove	sigmaprops
S2	function	algorithm
S2_ax	axiom	algorithm
S2_pqr	lemma	summations
S2_pqr_proof	prove	summations
sa_basis_proof	prove	sigmaprops
sa_proof	prove	sigmaprops
sa_step_proof	prove	sigmaprops
sb	lemma	sigmaprops
sb_basis_proof	prove	sigmaprops
sb_proof	prove	sigmaprops
sb_step_proof	prove	sigmaprops
sc_basis_proof	prove	sigmaprops
sc_proof	prove	sigmaprops
sc_step_proof	prove	sigmaprops
Sdef	axiom	time
Sigma	const	algorithm
sigma	function	sigmaprops
sigma1	lemma	sigmaprops
sigma1_basis	lemma	sigmaprops
sigma1_proof	prove	sigmaprops
sigma1_step	lemma	sigmaprops
sigmaprops	module	sigmaprops
sigma_abs	lemma	sigmaprops
sigma_abs_basis	lemma	sigmaprops
sigma_abs_step	lemma	sigmaprops
sigma_ax	axiom	sigmaprops

Table A.11: Cross-Reference to EHDM Identifiers (Continued)

Identifier	Type of Declaration	Module where Declared
sigma_bound	lemma	sigmaprops
sigma_bound2	lemma	sums
sigma_bound2_proof	prove	sums
sigma_bound_basis	lemma	sigmaprops
sigma_bound_proof	prove	sigmaprops
sigma_bound_step	lemma	sigmaprops
sigma_const	lemma	sigmaprops
sigma_const_basis	lemma	sigmaprops
sigma_const_step	lemma	sigmaprops
sigma_mult	lemma	sigmaprops
sigma_mult_basis	lemma	sigmaprops
sigma_mult_step	lemma	sigmaprops
sigma_rev	lemma	sigmaprops
sigma_rev_basis	lemma	sigmaprops
sigma_rev_proof	prove	sigmaprops
sigma_rev_step	lemma	sigmaprops
sigma_sum	lemma	sigmaprops
sigma_sum_basis	lemma	sigmaprops
sigma_sum_step	lemma	sigmaprops
SinR	lemma	time
SinR_proof	prove	time
skew	function	algorithm
small_shift	lemma	clockprops
small_shift_proof	prove	clockprops
sm_basis_proof	prove	sigmaprops
sm_proof	prove	sigmaprops
sm_step_proof	prove	sigmaprops
split_basis_proof	prove	sigmaprops
split_mean	lemma	sums
split_mean_proof	prove	sums
split_proof	prove	sigmaprops
split_sigma	lemma	sigmaprops
split_sigma_basis	lemma	sigmaprops
split_sigma_step	lemma	sigmaprops
split_step_proof	prove	sigmaprops
split_sum	lemma	sums
split_sum_proof	prove	sums

Table A.12: Cross-Reference to EHDM Identifiers (Continued)

Identifier	Type of Declaration	Module where Declared
srb_proof	prove	sigmaprops
srp_proof	prove	sigmaprops
ss_basis_proof	prove	sigmaprops
ss_proof	prove	sigmaprops
ss_step_proof	prove	sigmaprops
step1	lemma	juggle
step1_proof	prove	juggle
step2	lemma	juggle
step2_proof	prove	juggle
step3	lemma	juggle
step3_proof	prove	juggle
step4	lemma	juggle
step4_proof	prove	juggle
step5	lemma	juggle
step5_proof	prove	juggle
sub1_proof	prove	lemma6
sub2_proof	prove	lemma6
sublemma1	lemma	lemma4
sublemma1	lemma	lemma6
sublemma1_proof	prove	lemma4
sublemma2	lemma	lemma6
sublemma_A	lemma	lemma6
sub_A_proof	prove	lemma6
sum	function	sums
summations	module	summations
sums	module	sums
sum_ax	axiom	sums
sum_bound	lemma	sums
sum_bound0	lemma	sums
sum_bound0_proof	prove	sums
sum_bound1	lemma	sums
sum_bound1_proof	prove	sums
sum_bound2	lemma	sums
sum_bound2_proof	prove	sums
sum_bound_mod	lemma	sums
sum_bound_mod_proof	prove	sums
sum_bound_proof	prove	sums

Table A.13: Cross-Reference to EHDM Identifiers (Continued)

Identifier	Type of Declaration	Module where Declared
sum_mult	lemma	sums
sum_mult_proof	prove	sums
Theorem_1	theorem	algorithm
Theorem_1_proof	prove	main
Theorem_2	theorem	algorithm
Theorem_2_proof	prove	algorithm
time	module	time
times_half	lemma	arithmetics
times_half_proof	prove	arithmetics
Ti_in_R	lemma	time
Ti_in_S	lemma	time
Ti_in_S_proof	prove	time
Ti_proof	prove	time
T_next	lemma	time
T_next_proof	prove	time
T_sup	function	time
T_sup_ax	axiom	time
T_ZERO	const	time
upper_bound	lemma	clockprops
upper_bound_proof	prove	clockprops
zero_correction	axiom	clocks

Table A.14: Cross-Reference to EHDM Identifiers (concluded)

Appendix B

LaTeX-printed Specification Listings

The following specification listings were formatted and converted to mathematical notation automatically using the EHDM LaTeX-printer. The raw EHDM text is in Appendix D. All the proofs in these listings have been checked by the EHDM theorem prover using the EHDM variable settings `prmode = checking` and `prlambdafree = everywhere`.

Module	Page
Absolutes	92
Algorithm	120
Arithmetics	94
Clockprops	123
Clocks	118
Functionprops	100
Juggle	139
Lemma1	126
Lemma2	127
Lemma3	129
Lemma4	130
Lemma5	132
Lemma6	133
Main	144
Natinduction	101
Natprops	98
Sigmaprops	108
Summations	135
Sums	103
Time	116

Table B.1: Page References to EHDM Specification Modules absolutes: **Module**

Exporting $|\star 1|$

Theory

a, b, w, x, y, z: VAR number
$|\star 1|$: function[number → number]

abs_ax: **Axiom** $|a| =$ **if** $a < 0$ **then** $-a$ **else** a **end if**

abs_times: **Axiom** $|a * b| = |a| * |b|$

abs_div: **Axiom** $b \neq 0 \supset |a/b| = |a|/|b|$

abs_ax0: **Lemma** $0 = |0|$

abs_ax1: **Lemma** $0 \leq |x|$

abs_ax2: **Lemma** $|x + y| \leq |x| + |y|$

abs_ax2b: **Lemma** $|x + y + z| \leq |x| + |y| + |z|$

abs_ax2c: **Lemma** $|w + x + y + z| \leq |w| + |x| + |y| + |z|$

abs_ax3: **Lemma** $|-x| = |x|$

abs_ax4: **Lemma** $|x - y| = |y - x|$

abs_ax5: **Lemma** $0 \leq x \wedge x \leq z \wedge 0 \leq y \wedge y \leq z \supset |x - y| \leq z$

abs_ax6: **Lemma** $|x| \leq y \supset -y \leq x \wedge x \leq y$

abs_ax7: **Lemma** $|x| = ||x||$

abs_ax8: **Lemma** $|x - y| \leq |x| + |y|$

pos_abs: **Lemma** $0 \leq x \supset |x| = x$

Proof

abs_proof0: **Prove** abs_ax0 **from** abs_ax {a ← 0}

abs_proof1: **Prove** abs_ax1 **from** abs_ax {a ← x}

abs_proof2: **Prove** abs_ax2 **from**
abs_ax {a ← $x + y$}, abs_ax {a ← x}, abs_ax {a ← y}

abs_proof2b: **Prove** abs_ax2b **from**
abs_ax2 {y ← $y + z$}, abs_ax2 {x ← y, y ← z}

abs_proof2c: **Prove** abs_ax2c **from**
abs_ax2 $\{x \leftarrow w, y \leftarrow x + y + z\}$, abs_ax2b

abs_proof3: **Prove** abs_ax3 **from** abs_ax $\{a \leftarrow x\}$, abs_ax $\{a \leftarrow -x\}$

abs_proof4: **Prove** abs_ax4 **from**
abs_ax $\{a \leftarrow x - y\}$, abs_ax $\{a \leftarrow y - x\}$

abs_proof5: **Prove** abs_ax5 **from** abs_ax $\{a \leftarrow x - y\}$

abs_proof6: **Prove** abs_ax6 **from** abs_ax $\{a \leftarrow x\}$

abs_proof7: **Prove** abs_ax7 **from** abs_ax1, abs_ax $\{a \leftarrow |x|\}$

abs_proof8: **Prove** abs_ax8 **from**
abs_ax $\{a \leftarrow x - y\}$, abs_ax $\{a \leftarrow x\}$, abs_ax $\{a \leftarrow y\}$

pos_abs_proof: **Prove** pos_abs **from** abs_ax $\{a \leftarrow x\}$

End absolutes

arithmetics: **Module**

Using absolutes

Exporting $\star 1 \times \star 2, \frac{\star 1}{2}$ **with** absolutes

Theory

$a, b, c, u, v, w, x, y, z$: VAR number
$\star 1 \times \star 2$: function[number, number → number]
$\frac{\star 1}{2}$: function[number → number]
(* ———————————————— *)

quotient_ax: **Axiom** $y \neq 0 \supset x/y = x * (1/y)$

quotient_ax1: **Axiom** $x \neq 0 \supset x/x = 1$

quotient_ax2: **Axiom** $z > 0 \supset 1/z > 0$

(* ———————————————— *)

div_times: **Lemma** $y \neq 0 \supset (x/y) * z = (x * z)/y$

div_distr: **Lemma** $z \neq 0 \supset x/z + y/z = (x + y)/z$

abs_div2: **Lemma** $y > 0 \supset |x/y| = |x|/y$

div_mon: **Lemma** $x < y \wedge z > 0 \supset x/z < y/z$

div_mon2: **Lemma** $x \leq y \wedge z > 0 \supset x/z \leq y/z$

div_prod: **Lemma** $y > 0 \wedge a < x * y \supset a/y < x$

div_prod2: **Lemma** $y > 0 \wedge a \leq x * y \supset a/y \leq x$

cancellation: **Lemma** $y \neq 0 \supset (y * x)/y = x$

(* ———————————————— *)

mult_ax: **Axiom** $x \times y = x * y$

mult1: **Axiom** $x \geq 0 \wedge y \geq 0 \supset x \times y \geq 0$

mult_mon: **Axiom** $x < y \wedge z > 0 \supset x \times z < y \times z$

(* ———————————————— *)

mult_mon2: **Lemma** $x \leq y \wedge z > 0 \supset x \times z \leq y \times z$

cancellation_mult: **Lemma** $y \neq 0 \supset x \times y/y = x$

mult0: **Lemma** $y = 0 \supset x \times y = 0$

mult_div: **Lemma** $y \neq 0 \supset x/y \times y = x$

(* ———————————————— *)

half_ax: **Axiom** $\frac{x}{2} = x/2$

(* ———————————————— *)

times_half: **Lemma** $2 * \frac{x}{2} = x$

half2: **Lemma** $\frac{x}{2} + \frac{x}{2} = x$

half3: **Lemma** $2 * \frac{x}{2} \times y = x \times y$

mult2: **Lemma** $2 * (x \times y) = (2 * x) \times y$

mult3: **Lemma** $x \times y + z = x \times y + x \times z$

mult4: **Lemma** $0 \leq x \wedge y \leq z \supset x \times y \leq x \times z$

rearrange: **Lemma**
$|x - y| \leq |x - (u + v)| + |y - (w + z)| + |u + v - (w + z)|$

rearrange_alt: **Lemma** $|x - y| \leq |x - (u + v)| + |u - w| + |y - (w + v)|$

Proof

div_times_proof: **Prove** div_times **from**
quotient_ax, quotient_ax {x ← $x * z$}

div_distr_proof: **Prove** div_distr **from**
quotient_ax {y ← z},
quotient_ax {x ← y, y ← z},
quotient_ax {x ← $x + y$, y ← z}

abs_div2_proof: **Prove** abs_div2 **from**
abs_div {a ← x, b ← y}, pos_abs {x ← y}

quotient_mult: **Lemma** $y \neq 0 \supset x/y = x \times 1/y$

quotient_mult_proof: **Prove** quotient_mult **from**
quotient_ax, mult_ax {y ← $1/y$}

div_mon_proof: **Prove** div_mon **from**
mult_mon {z ← $1/z$},
quotient_mult {y ← z},
quotient_mult {x ← y, y ← z},
quotient_ax2

div_mon2_proof: **Prove** div_mon2 **from** div_mon

div_mult: **Lemma** $y > 0 \wedge a < x \times y \supset a/y < x$

div_mult_proof: **Prove** div_mult **from**
div_mon {z ← y, x ← a, y ← $x \times y$}, cancellation_mult

div_mult2: **Lemma** $y > 0 \wedge a \leq x \times y \supset a/y \leq x$

div_mult2_proof: **Prove** div_mult2 **from**
div_mon {z ← y, x ← a, y ← $x \times y$}, cancellation_mult

div_prod_proof: **Prove** div_prod **from** div_mult, mult_ax

div_prod2_proof: **Prove** div_prod2 **from** div_mult2, mult_ax

cancellation_proof: **Prove** cancellation **from**
div_times {x ← y, z ← x}, quotient_ax1 {x ← y}

mult_mon2_proof: **Prove** mult_mon2 **from** mult_mon

cancellation_mult_proof: **Prove** cancellation_mult **from**
cancellation, mult_ax

mult0_proof: **Prove** mult0 **from** mult_ax {y ← 0}

mult_div_proof: **Prove** mult_div **from**
mult_ax {x ← x/y}, div_times {z ← y}, cancellation

times_half_proof: **Prove** times_half **from**
half_ax, div_times {y ← 2, z ← 2}, cancellation {y ← 2}

half2_proof: **Prove** half2 **from** times_half

half3_proof: **Prove** half3 **from** mult2 {x ← $\frac{x}{2}$}, times_half

mult2_proof: **Prove** mult2 **from** mult_ax, mult_ax {x ← $2 * x$}

mult3_proof: **Prove** mult3 **from**
mult_ax, mult_ax {y ← z}, mult_ax {y ← $y + z$}

mult4_proof: **Prove** mult4 **from** mult3 {z ← $z - y$}, mult1 {y ← $z - y$}

rearrange1: **Lemma**
$x - y = (x - (u + v)) + (w + z - y) + (u + v - (w + z))$

rearrange1_proof: **Prove** rearrange1

rearrange2: **Lemma**
$|(x - (u + v)) + (w + z - y) + (u + v - (w + z))|$
$\le |x - (u + v)| + |y - (w + z)| + |u + v - (w + z)|$

rearrange2_proof: **Prove** rearrange2 **from**
abs_ax2b {x ← $x - (u + v)$, y ← $u + v - (w + z)$, z ← $w + z - y$},
abs_ax3 {x ← $w + z - y$}

rearrange_proof: **Prove** rearrange **from** rearrange1, rearrange2

rearrange_alt_proof: **Prove** rearrange_alt **from** rearrange {z ← v}

End arithmetics

natprops: **Module**

Exporting pred, diff

Theory

i, m, n: VAR nat
pred: function[nat $\rightarrow$ nat]

natpos: **Axiom** $n \geq 0$

pred_ax: **Axiom** $n \neq 0 \supset \text{pred}(n) = n - 1$

diff: function[nat, nat $\rightarrow$ nat]

diff_ax: **Axiom** $n \geq m \supset \text{diff}(n, m) = n - m$

pred_lemma: **Lemma** $\text{pred}(n + 1) = n$

diff_zero: **Lemma** $n > m \supset \text{diff}(n, m) > 0$

pred_diff: **Lemma** $n > m \supset \text{pred}(\text{diff}(n, m)) = \text{diff}(n, m + 1)$

diff1: **Lemma** $n \geq m \supset \text{diff}(n + 1, m + 1) = \text{diff}(n, m)$

diff_diff: **Lemma**
$n \geq m \wedge n \geq i \wedge m \geq i \supset \text{diff}(\text{diff}(n, i), \text{diff}(m, i)) = \text{diff}(n, m)$

diff_plus: **Lemma** $n \geq m \supset m + \text{diff}(n, m) = n$

diff_ineq: **Lemma** $n \geq m \wedge n \geq i \wedge m \geq i \supset \text{diff}(n, i) \geq \text{diff}(m, i)$

Proof

pred_lemma_proof: **Prove** pred_lemma **from** pred_ax {n $\leftarrow n + 1$}, natpos

diff_zero_proof: **Prove** diff_zero **from** diff_ax

pred_diff_proof: **Prove** pred_diff **from**
pred_ax {n $\leftarrow \text{diff}(n, m)$}, diff_ax, diff_ax {m $\leftarrow m + 1$}

diff1_proof: **Prove** diff1 **from**
diff_ax, diff_ax {n $\leftarrow n + 1$, m $\leftarrow m + 1$}

diff_diff_proof: **Prove** diff_diff **from**
diff_ax,
diff_ax {m $\leftarrow i$},
diff_ax {n $\leftarrow m$, m $\leftarrow i$},
diff_ax {n $\leftarrow \text{diff}(n, i)$, m $\leftarrow \text{diff}(m, i)$}

diff_plus_proof: **Prove** diff_plus **from** diff_ax

diff_ineq_proof: **Prove** diff_ineq **from**
diff_ax {m ← *i*}, diff_ax {n ← *m*, m ← *i*}

End natprops

functionprops: **Module**

Theory

F, G: VAR function[nat → number]
x: VAR nat

extensionality: **Axiom** $(\forall\, \mathrm{x}: F(x) = G(x)) \supset F = G$

End functionprops

natinduction: **Module**

Using natprops

Theory

i, i0, i1, i2, i3, j, m, n: VAR nat
prop, A, B: VAR function[nat → bool]
prop2: VAR function[nat, nat → bool]

induction_m: **Axiom**
$(\text{prop}(m) \wedge (\forall \text{i}: i \geq m \wedge \text{prop}(i) \supset \text{prop}(i+1)))$
$\supset (\forall \text{n}: n \geq m \supset \text{prop}(n))$

induction2: **Axiom**
$(\forall \text{i0}: \text{prop2}(\text{i0}, 0))$
$\wedge (\forall \text{j}: (\forall \text{i1}: \text{prop2}(\text{i1}, j)) \supset (\forall \text{i2}: \text{prop2}(\text{i2}, j+1)))$
$\supset (\forall \text{i3, n}: \text{prop2}(\text{i3}, n))$

mod_induction_m: **Lemma**
$(\forall \text{j}: j \geq m \wedge A(j+1) \supset A(j))$
$\wedge ((A(m) \supset B(m)) \wedge (\forall \text{i}: i \geq m \wedge A(i+1) \wedge B(i) \supset B(i+1)))$
$\supset (\forall \text{n}: n \geq m \wedge A(n) \supset B(n))$

induction: **Lemma**
$(\text{prop}(0) \wedge (\forall \text{i}: \text{prop}(i) \supset \text{prop}(i+1))) \supset (\forall \text{n}: \text{prop}(n))$

mod_induction: **Lemma**
$(\forall \text{j}: A(j+1) \supset A(j))$
$\wedge ((A(0) \supset B(0)) \wedge (\forall \text{i}: A(i+1) \wedge B(i) \supset B(i+1)))$
$\supset (\forall \text{n}: A(n) \supset B(n))$

induction1: **Lemma**
$(\text{prop}(1) \wedge (\forall \text{i}: i \geq 1 \wedge \text{prop}(i) \supset \text{prop}(i+1)))$
$\supset (\forall \text{n}: n \geq 1 \supset \text{prop}(n))$

mod_induction1: **Lemma**
$(\forall \mathrm{j}: j \geq 1 \wedge A(j+1) \supset A(j))$
$\wedge ((A(1) \supset B(1)) \wedge (\forall \mathrm{i}: i \geq 1 \wedge A(i+1) \wedge B(i) \supset B(i+1)))$
$\supset (\forall \mathrm{n}: n \geq 1 \wedge A(n) \supset B(n))$

Proof

mod_m_proof: **Prove** mod_induction_m {i ← i@p1, j ← i} **from**
induction_m {prop ← (λ i→ bool : $A(i) \supset B(i)$)}

induction_proof: **Prove** induction {i ← i@p1} **from**
induction_m {m ← 0}, natpos

mod_induction_proof: **Prove** mod_induction {i ← i@p1, j ← j@p1} **from**
mod_induction_m {m ← 0}, natpos

induction1_proof: **Prove** induction1 {i ← i@p1} **from**
induction_m {m ← 1}

mod_induction1_proof: **Prove** mod_induction1 {i ← i@p1, j ← j@p1}
from mod_induction_m {m ← 1}

End natinduction

sums: **Module**

Using arithmetics, natprops, sigmaprops

Exporting $\sum_{*1}^{*2}(*3), \bigoplus_{*1}^{*2}(*3)$

Theory

i, j, k, n, pp, qq, rr: VAR nat
x, y, z: VAR number
F, G: VAR function[nat → number]
$\sum_{*1}^{*2}(*3)$: function[nat, nat, function[nat → number] → number]
$\bigoplus_{*1}^{*2}(*3)$: function[nat, nat, function[nat → number] → number]

sum_ax: **Axiom**
$\sum_i^j F =$ **if** $i \leq j+1$ **then** $\sigma(i, \text{diff}(j+1, i), F)$ **else** 0 **end if**

mean_ax: **Axiom**
$\bigoplus_i^j F =$ **if** $i \leq j$ **then** $\sum_i^j F/(j+1-i)$ **else** 0 **end if**

mean_lemma: **Lemma**
$\bigoplus_i^j F =$ **if** $i \leq j$
then $\sigma(i, \text{diff}(j+1, i), F)/(j+1-i)$
else 0
end if

split_sum: **Lemma**
$i \leq j+1 \wedge i \leq k+1 \wedge k \leq j \supset \sum_i^j F = \sum_i^k F + \sum_{k+1}^j F$

split_mean: **Lemma**
$i \leq j \wedge i \leq k+1 \wedge k \leq j$
$\supset \bigoplus_i^j F = (\sum_i^k F + \sum_{k+1}^j F)/(j-i+1)$

sum_bound: **Lemma**
$i \leq j+1 \wedge (\forall \text{pp}: i \leq \text{pp} \wedge \text{pp} \leq j \supset F(\text{pp}) < x)$
$\supset \sum_i^j F \leq x * (j-i+1)$

mean_bound: **Lemma**
$i \leq j \wedge (\forall \text{pp}: i \leq \text{pp} \wedge \text{pp} \leq j \supset F(\text{pp}) < x) \supset \bigoplus_i^j F < x$

mean_const: **Lemma** $i \leq j \supset x = \bigoplus_i^j(\lambda\, \text{qq} \rightarrow \text{number} : x)$

mean_mult: **Lemma** $\bigoplus_i^j F * x = \bigoplus_i^j(\lambda\, \text{qq} \rightarrow \text{number} : F(\text{qq}) * x)$

mean_sum: **Lemma**
$\bigoplus_i^j F + \bigoplus_i^j G = \bigoplus_i^j(\lambda\, \text{qq} \rightarrow \text{number} : F(\text{qq}) + G(\text{qq}))$

mean_diff: **Lemma**
$\bigoplus_i^j F - \bigoplus_i^j G = \bigoplus_i^j(\lambda\, \text{qq} \rightarrow \text{number} : F(\text{qq}) - G(\text{qq}))$

abs_mean: **Lemma** $|\bigoplus_i^j F| \leq \bigoplus_i^j(\lambda\, \text{qq} \rightarrow \text{number} : |F(\text{qq})|)$

rearrange_sum: **Lemma**
$i \leq j \supset x + \bigoplus_i^j F - (y + \bigoplus_i^j G)$
$= \bigoplus_i^j(\lambda\, \text{qq} \rightarrow \text{number} : x + F(\text{qq}) - (y + G(\text{qq})))$

Proof

mean_lemma_proof: **Prove** mean_lemma **from** mean_ax, sum_ax

(* —————————————————————— *)

split_sum_proof: **Prove** split_sum **from**
sum_ax,
sum_ax {j ← k},
sum_ax {i ← $k + 1$},
split_sigma {n ← diff($j + 1, i$), m ← diff($k + 1, i$), i ← i},
diff_diff {n ← $j + 1$, m ← $k + 1$},
diff_plus {n ← $k + 1$, m ← i},
diff_ineq {n ← $j + 1$, m ← $k + 1$}

split_mean_proof: **Prove** split_mean **from** split_sum, mean_ax

(* —————————————————————— *)

sigma_bound2: **Lemma**
$n > 0 \wedge (\forall\, \text{k}: i \leq k \wedge k \leq i + \text{pred}(n) \supset F(k) < x)$
$\supset \sigma(i, n, F) < x \times n$

sigma_bound2_proof: **Prove** sigma_bound2 {k ← k@p1} **from**
sigma_bound, mult_ax {y ← n}

sum_bound_mod: **Lemma**
$i \leq j \wedge (\forall\, \text{pp}: i \leq \text{pp} \wedge \text{pp} \leq j \supset F(\text{pp}) < x)$
$\supset \sum_i^j F < x \times (j + 1 - i)$

sum_bound_mod_proof: **Prove** sum_bound_mod {pp ← k@p2} **from**
  sum_ax,
  sigma_bound2 {n ← diff($j+1, i$), i ← i},
  pred_diff {n ← $j+1$, m ← i},
  diff_ax {n ← $j+1$, m ← i},
  diff_ax {n ← $j+1$, m ← $i+1$}

sum_bound1: **Lemma**
  $i \leq j \wedge (\forall \text{pp}: i \leq \text{pp} \wedge \text{pp} \leq j \supset F(\text{pp}) < x)$
    $\supset \sum_i^j F < x * (j - i + 1)$

sum_bound1_proof: **Prove** sum_bound1 {pp ← pp@p1} **from**
  sum_bound_mod, mult_ax {y ← $j+1-i$}

sum_bound0: **Lemma**
  $i = j+1 \wedge (\forall \text{pp}: i \leq \text{pp} \wedge \text{pp} \leq j \supset F(\text{pp}) < x)$
    $\supset \sum_i^j F \leq x \times (j+1-i)$

sum_bound0_proof: **Prove** sum_bound0 **from**
  sum_ax {i ← $j+1$},
  diff_ax {n ← $j+1$, m ← $j+1$},
  sigma_ax {i ← $j+1$, n ← 0},
  mult0 {y ← $j+1-i$}

sum_bound2: **Lemma**
  $i \leq j+1 \wedge (\forall \text{pp}: i \leq \text{pp} \wedge \text{pp} \leq j \supset F(\text{pp}) < x)$
    $\supset \sum_i^j F \leq x \times (j+1-i)$

sum_bound2_proof: **Prove** sum_bound2 {pp ← pp@p1} **from**
  sum_bound_mod, sum_bound0

sum_bound_proof: **Prove** sum_bound {pp ← pp@p1} **from**
  sum_bound2, mult_ax {y ← $j+1-i$}

(* ———————————————————————————— *)

mean_bound_proof: **Prove** mean_bound {pp ← pp@p1} **from**
  sum_bound1, mean_ax, div_prod {a ← $\sum_i^j F$, y ← $j-i+1$}

(* ———————————————————————— *)

mean_const_proof: **Prove** mean_const **from**
  mean_lemma {F ← (λ qq→ number : x)},
  sigma_const {n ← diff($j + 1, i$), i ← i},
  diff_ax {n ← $j + 1$, m ← i},
  cancellation {y ← $j + 1 - i$}

(* ———————————————————————— *)

sum_mult: **Lemma** $\sum_i^j F * x = \sum_i^j$(λ qq→ number : F(qq) $* x$)

sum_mult_proof: **Prove** sum_mult **from**
  sum_ax,
  sum_ax {F ← (λ qq→ number : F(qq) $* x$)},
  mod_sigma_mult {i ← i, n ← diff($j + 1, i$)}

mean_mult_proof: **Prove** mean_mult **from**
  mean_ax,
  mean_ax {F ← (λ qq→ number : F(qq) $* x$)},
  sum_mult,
  div_times {x ← $\sum_i^j$ F@p3, y ← $j + 1 - i$, z ← x}

(* ———————————————————————— *)

mean_sum_proof: **Prove** mean_sum **from**
  mean_lemma {F ← (λ qq→ number : F(qq) + G(qq))},
  mean_lemma,
  mean_lemma {F ← G},
  sigma_sum {n ← diff($j + 1, i$), i ← i},
  div_distr {x ← $\sigma(i$, diff($j + 1, i$), F),
    y ← $\sigma(i$, diff($j + 1, i$), G),
    z ← $j + 1 - i$}

(* ———————————————————————— *)

mean_diff_proof: **Prove** mean_diff **from**
  mean_mult {F ← G, x ← -1},
  mean_sum {G ← (λ qq→ number : G(qq) $* -1$)}

(* ———————————————————————— *)

abs_sum: **Lemma** $|\sum_i^j F| \leq \sum_i^j(\lambda\, \mathrm{qq} \to \mathrm{number} : |F(\mathrm{qq})|)$

abs_sum_proof: **Prove** abs_sum **from**
sum_ax,
sum_ax {F ← (λ qq→ number : $|F(\mathrm{qq})|$)},
sigma_abs {n ← diff($j+1, i$), i ← i},
abs_ax0

abs_mean_proof: **Prove** abs_mean **from**
mean_ax,
mean_ax {F ← (λ qq→ number : $|F(\mathrm{qq})|$)},
abs_sum,
abs_div2 {x ← $\sum_i^j F$, y ← $j+1-i$},
div_mon2 {x ← $|\sum_i^j F|$, y ← $\sum_i^j$ F@p2, z ← $j+1-i$},
abs_ax0

(* ———————————————————————— *)

rearrange_sub: **Lemma**
$i \leq j \supset x + \bigoplus_i^j F = \bigoplus_i^j(\lambda\, \mathrm{qq} \to \mathrm{number} : x + F(\mathrm{qq}))$

rearrange_sub_proof: **Prove** rearrange_sub **from**
mean_const, mean_sum {G ← (λ qq→ number : x)}

rearrange_sum_proof: **Prove** rearrange_sum **from**
rearrange_sub,
rearrange_sub {x ← y, F ← G},
mean_diff {F ← (λ pp→ number : x + F@c(pp)),
G ← (λ pp→ number : y + G@c(pp))}

End sums

sigmaprops: **Module**

Using arithmetics, natprops, functionprops, natinduction

Exporting $\sigma(\star 1, \star 2, \star 3)$

Theory

i, i1, i2, j, k, l: VAR nat
F, G: VAR function[nat → number]
n, m, mm, nn, qq: VAR nat
x, y: VAR number
$\sigma(\star 1, \star 2, \star 3)$: function[nat, nat, function[nat → number] → number]

sigma_ax: **Axiom**
$\sigma(i, n, F) =$ **if** $n = 0$
then 0
else $F(i + \text{pred}(n)) + \sigma(i, \text{pred}(n), F)$
end if

sigma_const: **Lemma** $\sigma(i, n, (\lambda\, \text{qq} \rightarrow \text{number} : x)) = n * x$

sigma_mult: **Lemma**
$\sigma(i, n, (\lambda\, \text{qq} \rightarrow \text{number} : x * F(\text{qq}))) = x * \sigma(i, n, F)$

mod_sigma_mult: **Lemma**
$\sigma(i, n, (\lambda\, \text{qq} \rightarrow \text{number} : F(\text{qq}) * x)) = \sigma(i, n, F) * x$

sigma_sum: **Lemma**
$\sigma(i, n, F) + \sigma(i, n, G) = \sigma(i, n, (\lambda\, \text{qq} \rightarrow \text{number} : F(\text{qq}) + G(\text{qq})))$

split_sigma: **Lemma**
$n \geq m \supset \sigma(i, n, F) = \sigma(i, m, F) + \sigma(i + m, \text{diff}(n, m), F)$

sigma_abs: **Lemma** $|\sigma(i, n, F)| \leq \sigma(i, n, (\lambda\, \text{qq} \rightarrow \text{number} : |F(\text{qq})|))$

sigma_bound: **Lemma**
$n > 0 \wedge (\forall \text{k} : i \leq k \wedge k \leq i + \text{pred}(n) \supset F(k) < x)$
$\supset \sigma(i, n, F) < n * x$

bounded: function[nat, nat, function[nat → number], number → bool]

bounded_ax: **Axiom**
    $n > 0 \supset (\text{bounded}(i, n, F, x)$
        $= (\forall \text{k}: i \leq k \wedge k \leq i + \text{pred}(n) \supset F(k) < x))$

revsigma: function[nat, nat, function[nat → number] → number]

revsigma_ax: **Axiom**
    revsigma(i, n, F) = **if** $n = 0$
        then 0
        else $F(i)$ + revsigma($i + 1$, pred(n), F)
        end if

sigma_rev: **Lemma** $\sigma(i, n, F)$ = revsigma(i, n, F)

Proof

sigma_const_basis: **Lemma** $\sigma(i, 0, (\lambda\, \text{qq} \rightarrow \text{number} : x)) = 0$

sc_basis_proof: **Prove** sigma_const_basis **from**
    sigma_ax {n ← 0, F ← (λ qq→ number : x)}

sigma_const_step: **Lemma**
    $\sigma(i, n, (\lambda\, \text{qq} \rightarrow \text{number} : x)) = n * x$
        $\supset \sigma(i, n + 1, (\lambda\, \text{qq} \rightarrow \text{number} : x)) = (n + 1) * x$

sc_step_proof: **Prove** sigma_const_step **from**
    sigma_ax {n ← $n + 1$, F ← (λ qq→ number : x)}, pred_lemma

sc_proof: **Prove** sigma_const **from**
    induction {prop ← (λ nn→ bool :
        $\sigma(i, \text{nn}, (\lambda\, \text{qq} \rightarrow \text{number} : x)) = \text{nn} * x$)},
    sigma_const_basis,
    sigma_const_step {n ← i@p1}

(* ———————————————————————— *)

sigma_mult_basis: **Lemma**
    $\sigma(i, 0, (\lambda\, \text{qq} \rightarrow \text{number} : x * F(\text{qq}))) = x * \sigma(i, 0, F)$

sm_basis_proof: **Prove** sigma_mult_basis **from**
sigma_ax {n ← 0},
sigma_ax {n ← 0, F ← (λ qq→ number : $x * F(\text{qq})$)}

sigma_mult_step: **Lemma**
$\sigma(i, n, (\lambda\, \text{qq} \rightarrow \text{number} : x * F(\text{qq}))) = x * \sigma(i, n, F)$
$\supset \sigma(i, n+1, (\lambda\, \text{qq} \rightarrow \text{number} : x * F(\text{qq}))) = x * \sigma(i, n+1, F)$

sm_step_proof: **Prove** sigma_mult_step **from**
sigma_ax {n ← $n + 1$, F ← (λ qq→ number : $x * F(\text{qq})$)},
sigma_ax {n ← $n + 1$},
pred_lemma

sm_proof: **Prove** sigma_mult **from**
induction {prop ← (λ nn→ bool :
$\sigma(i, \text{nn}, (\lambda\, \text{qq} \rightarrow \text{number} : x * F(\text{qq}))) = x * \sigma(i, \text{nn}, F)$)},
sigma_mult_basis,
sigma_mult_step {n ← i@p1}

(* ———————————————————— *)

mod_sigma_mult_proof: **Prove** mod_sigma_mult **from**
sigma_mult,
extensionality {F ← (λ qq→ number : $x * F(\text{qq})$),
G ← (λ qq→ number : $F(\text{qq}) * x$)}

(* ———————————————————— *)

sigma_sum_basis: **Lemma**
$\sigma(i, 0, F) + \sigma(i, 0, G) = \sigma(i, 0, (\lambda\, \text{qq} \rightarrow \text{number} : F(\text{qq}) + G(\text{qq})))$

ss_basis_proof: **Prove** sigma_sum_basis **from**
sigma_ax {n ← 0, F ← (λ qq→ number : $F(\text{qq}) + G(\text{qq})$)},
sigma_ax {n ← 0, F ← (λ qq→ number : $G(\text{qq})$)},
sigma_ax {n ← 0}

sigma_sum_step: **Lemma**
$\sigma(i, n, F) + \sigma(i, n, G) = \sigma(i, n, (\lambda\, \text{qq} \rightarrow \text{number} : F(\text{qq}) + G(\text{qq})))$
$\supset \sigma(i, n+1, F) + \sigma(i, n+1, G)$
$= \sigma(i, n+1, (\lambda\, \text{qq} \rightarrow \text{number} : F(\text{qq}) + G(\text{qq})))$

ss_step_proof: **Prove** sigma_sum_step **from**
    sigma_ax {n ← $n+1$, F ← (λ qq→ number : $F(\text{qq}) + G(\text{qq})$)},
    sigma_ax {n ← $n+1$, F ← (λ qq→ number : $G(\text{qq})$)},
    sigma_ax {n ← $n+1$},
    pred_lemma

ss_proof: **Prove** sigma_sum **from**
    induction {prop ← (λ nn→ bool :
        $\sigma(i, \text{nn}, F) + \sigma(i, \text{nn}, G)$
            $= \sigma(i, \text{nn}, (\lambda\, \text{qq} \rightarrow \text{number} : F(\text{qq}) + G(\text{qq})))$)},
    sigma_sum_basis,
    sigma_sum_step {n ← i@p1}

(* ———————————————————————— *)

split_sigma_basis: **Lemma** $\sigma(i, n, F) = \sigma(i, 0, F) + \sigma(i, \text{diff}(n, 0), F)$

split_basis_proof: **Prove** split_sigma_basis **from**
    sigma_ax, sigma_ax {n ← 0}, diff_ax {m ← 0}, natpos

split_sigma_step: **Lemma**
    $(n \geq m \supset \sigma(i, n, F) = \sigma(i, m, F) + \sigma(i + m, \text{diff}(n, m), F))$
        $\supset (n \geq m + 1$
            $\supset \sigma(i, n, F) = \sigma(i, m+1, F) + \sigma(i + m + 1, \text{diff}(n, m+1), F))$

split_step_proof: **Prove** split_sigma_step **from**
    sigma_ax {n ← $m+1$},
    sigma_rev {i ← $i+m+1$, n ← diff($n, m+1$)},
    revsigma_ax {i ← $i+m$, n ← diff(n, m)},
    sigma_rev {i ← $i+m$, n ← diff(n, m)},
    pred_lemma {n ← m},
    pred_diff,
    diff_zero,
    natpos {n ← m}

split_proof: **Prove** split_sigma **from**
    induction {n ← m,
      prop ← (λ nn→ bool :
          $n \geq \text{nn} \supset \sigma(i, n, F) = \sigma(i, \text{nn}, F) + \sigma(i + \text{nn}, \text{diff}(n, \text{nn}), F)$)},
    split_sigma_basis,
    split_sigma_step {m ← i@p1}

(* ———————————————————— *)

sigma_abs_basis: **Lemma**
$|\sigma(i, 0, F)| \leq \sigma(i, 0, (\lambda\, \mathrm{qq} \rightarrow \mathrm{number} : |F(\mathrm{qq})|))$

sa_basis_proof: **Prove** sigma_abs_basis **from**
sigma_ax {n ← 0},
sigma_ax {n ← 0, F ← $(\lambda\, \mathrm{qq} \rightarrow \mathrm{number} : |F(\mathrm{qq})|)$},
abs_ax0

sigma_abs_step: **Lemma**
$|\sigma(i, n, F)| \leq \sigma(i, n, (\lambda\, \mathrm{qq} \rightarrow \mathrm{number} : |F(\mathrm{qq})|))$
$\supset |\sigma(i, n+1, F)| \leq \sigma(i, n+1, (\lambda\, \mathrm{qq} \rightarrow \mathrm{number} : |F(\mathrm{qq})|))$

sa_step_proof: **Prove** sigma_abs_step **from**
sigma_ax {n ← $n + 1$},
sigma_ax {n ← $n + 1$, F ← $(\lambda\, \mathrm{qq} \rightarrow \mathrm{number} : |F(\mathrm{qq})|)$},
abs_ax2 {x ← $F(i + n)$, y ← $\sigma(i, n, F)$},
natpos,
pred_lemma

sa_proof: **Prove** sigma_abs **from**
induction {prop ← $(\lambda\, \mathrm{nn} \rightarrow \mathrm{bool} :$
$|\sigma(i, \mathrm{nn}, F)| \leq \sigma(i, \mathrm{nn}, (\lambda\, \mathrm{qq} \rightarrow \mathrm{number} : |F(\mathrm{qq})|)))$},
sigma_abs_basis,
sigma_abs_step {n ← i@p1}

(* ———————————————————— *)

bounded_lemma: **Lemma**
$n > 0 \wedge \mathrm{bounded}(i, n+1, F, x) \supset \mathrm{bounded}(i, n, F, x)$

bounded_proof: **Prove** bounded_lemma **from**
bounded_ax {k ← k@p1},
bounded_ax {n ← $n + 1$, k ← k@p1},
pred_lemma,
pred_ax

sigma_bound_basis: **Lemma** $\mathrm{bounded}(i, 1, F, x) \supset \sigma(i, 1, F) < x$

sb_basis_proof: **Prove** sigma_bound_basis **from**
bounded_ax {n ← 1, k ← i},
sigma_ax {n ← 0},
sigma_ax {n ← 1},
pred_ax {n ← 1}

alt_sigma_bound_step: **Lemma**
$n > 0 \wedge \text{bounded}(i, n+1, F, x) \wedge \sigma(i, n, F) < n \times x$
$\supset \sigma(i, n+1, F) < x + n \times x$

alt_sb_step_proof: **Prove** alt_sigma_bound_step **from**
bounded_ax {n ← $n + 1$, k ← $i + n$},
sigma_ax {n ← $n + 1$},
pred_lemma,
natpos

sigma_bound_step: **Lemma**
$n > 0 \wedge \text{bounded}(i, n+1, F, x) \wedge \sigma(i, n, F) < n * x$
$\supset \sigma(i, n+1, F) < (n+1) * x$

sb_step_proof: **Prove** sigma_bound_step **from**
alt_sigma_bound_step, mult_ax {x ← n, y ← x}

sb: **Lemma** $n > 0 \wedge \text{bounded}(i, n, F, x) \supset \sigma(i, n, F) < n * x$

sb_proof: **Prove** sb **from**
mod_induction1 {A ← (λ nn→ bool : bounded(i, nn, F, x)),
B ← (λ mm→ bool : σ(i, mm, F) < mm * x)},
bounded_lemma {n ← j@p1},
sigma_bound_basis,
sigma_bound_step {n ← i@p1}

sigma_bound_proof: **Prove** sigma_bound {k ← k@p2} **from** sb, bounded_ax

(* ———————————————————— *)

sigma1: **Lemma** $\sigma(i, n+1, F) = F(i) + \sigma(i+1, n, F)$

sigma1_basis: **Lemma** $\sigma(i, 1, F) = F(i) + \sigma(i+1, 0, F)$

s1b_proof: **Prove** sigma1_basis **from**
    sigma_ax {n ← 0},
    sigma_ax {i ← $i + 1$, n ← 0},
    sigma_ax {n ← 1},
    pred_ax {n ← 1}

sigma1_step: **Lemma**
    $\sigma(i, n + 1, F) = F(i) + \sigma(i + 1, n, F)$
        $\supset \sigma(i, n + 2, F) = F(i) + \sigma(i + 1, n + 1, F)$

s1s_proof: **Prove** sigma1_step **from**
    sigma_ax {i ← $i + 1$, n ← $n + 1$},
    sigma_ax {n ← $n + 2$},
    pred_lemma,
    pred_lemma {n ← $n + 1$},
    natpos

sigma1_proof: **Prove** sigma1 **from**
    induction {prop ← (λ nn→ bool :
            $\sigma(i, \text{nn} + 1, F) = F(i) + \sigma(i + 1, \text{nn}, F)$)},
    sigma1_basis,
    sigma1_step {n ← i@p1}

(* ——————————————————— *)

sigma_rev_basis: **Lemma** $\sigma(i, 0, F) = \text{revsigma}(i, 0, F)$

srb_proof: **Prove** sigma_rev_basis **from**
    sigma_ax {n ← 0}, revsigma_ax {n ← 0}

sigma_rev_step: **Lemma**
    ($\forall$ i1: $\sigma(\text{i1}, n, F) = \text{revsigma}(\text{i1}, n, F)$)
        $\supset$ ($\forall$ i2: $\sigma(\text{i2}, n + 1, F) = \text{revsigma}(\text{i2}, n + 1, F)$)

srp_proof: **Prove** sigma_rev_step {i1 ← i2 + 1} **from**
    revsigma_ax {i ← i2, n ← $n + 1$},
    sigma1 {i ← i2},
    pred_lemma,
    natpos

sigma_rev_proof: **Prove** sigma_rev **from**
induction2 {i1 ← i1@p3,
i3 ← *i*,
prop2 ← (λ i, nn→ bool : $\sigma(i, \text{nn}, F) = \text{revsigma}(i, \text{nn}, F)$)},
sigma_rev_basis {i ← i0@p1},
sigma_rev_step {i2 ← i2@p1, n ← j@p1}

End sigmaprops

time: **Module**

Using arithmetics

Exporting clocktime, realtime, period, $R, S, T^0, T^{(*1)}, *1 \in R^{(*2)}$, $* 1 \in S^{(*2)}$ **with** arithmetics

Theory

clocktime: TYPE IS number
realtime: TYPE IS number
period: TYPE IS nat
R, S: clocktime (* Synchronizing periods *)
posR: **Axiom** $0 < R$

posS: **Axiom** $0 < S$

C1: **Axiom** $R \geq 3 * S$

SinR: **Lemma** $S < R$

i: VAR period
$T^{(*1)}$: function[period → clocktime]
T^0: clocktime

T_sup_ax: **Axiom** $T^{(i)} = T^0 + i * R$

T_next: **Lemma** $T^{(i+1)} = T^{(i)} + R$

T, T_1, T_2, Π: VAR clocktime
$*1 \in R^{(*2)}$: function[clocktime, period → boolean]

Rdef: **Axiom** $T \in R^{(i)} = (\exists \Pi : 0 \leq \Pi \wedge \Pi \leq R \wedge T = T^{(i)} + \Pi)$

Ti_in_R: **Lemma** $T^{(i)} \in R^{(i)}$

$*1 \in S^{(*2)}$: function[clocktime, period → boolean]

Sdef: **Axiom** $T \in S^{(i)} = (\exists \Pi : 0 \leq \Pi \wedge \Pi \leq S \wedge T = T^{(i)} + R - S + \Pi)$

inRS: **Lemma** $T \in S^{(i)} \supset T \in R^{(i)}$

Ti_in_S: **Lemma** $T^{(i+1)} \in S^{(i)}$

in_S_lemma: **Lemma** $T_1 \in S^{(i)} \wedge T_2 \in S^{(i)} \supset |T_1 - T_2| \leq S$

Proof

SinR_proof: **Prove** SinR **from** C1, posS, posR

Ti_proof: **Prove** Ti_in_R **from** Rdef {T ← $T^{(i)}$, Π ← 0}, abs_ax0, posR

inRS_proof: **Prove** inRS **from** Sdef, Rdef {Π ← $R - S +$ Π@p1}, SinR

T_next_proof: **Prove** T_next **from** T_sup_ax, T_sup_ax {i ← $i + 1$}

Ti_in_S_proof: **Prove** Ti_in_S **from**
Sdef {Π ← S, T ← $T^{(i+1)}$}, posS, T_next

in_S_proof: **Prove** in_S_lemma **from**
Sdef {T ← T_1},
Sdef {T ← T_2},
abs_ax5 {x ← Π@p1, y ← Π@p2, z ← S}

End time

clocks: **Module**

Using time

Exporting proc, $c_{\star 1}(\star 2), \rho, C_{\star 1}^{(\star 2)}, A_{\star 1}^{(\star 2)}(\star 3), c_{\star 1}^{(\star 2)}(\star 3)$, nonfaulty
with time

Theory

proc: TYPE IS nat
p: VAR proc
$c_{\star 1}(\star 2)$: function[proc, clocktime → realtime]
$C_{\star 1}^{(\star 2)}$: function[proc, period → clocktime]

zero_correction: **Axiom** $C_p^{(0)} = 0$

i: VAR period
T, T_0, T_1, T_2, T_N: VAR clocktime
$A_{\star 1}^{(\star 2)}(\star 3)$: function[proc, period, clocktime → clocktime] =
(λ p, i, T→ clocktime : $T + C_p^{(i)}$)
$c_{\star 1}^{(\star 2)}(\star 3)$: function[proc, period, clocktime → realtime]

clockdef: **Axiom** $c_p^{(i)}(T) = c_p(A_p^{(i)}(T))$

goodclock: function[proc, clocktime, clocktime → bool]
ρ: number

rho_pos: **Axiom** $\frac{\rho}{2} \geq 0$

rho_small: **Axiom** $\frac{\rho}{2} < 1$

gc_ax: **Axiom**
goodclock(p, T_0, T_N)
$= (\forall T_1, T_2 :$
$T_0 \leq T_1 \wedge T_0 \leq T_2 \wedge T_1 \leq T_N \wedge T_2 \leq T_N$
$\supset |c_p(T_1) - c_p(T_2) - (T_1 - T_2)| < \frac{\rho}{2} \times |T_1 - T_2|)$

monotonicity: **Theorem**
goodclock$(p, T_0, T_N) \wedge T_0 \leq T_1 \wedge T_0 \leq T_2 \wedge T_1 \leq T_N \wedge T_2 \leq T_N$
$\supset (T_1 > T_2 \supset c_p(T_1) > c_p(T_2))$

nonfaulty: function[proc, period → boolean]

A1: **Axiom** nonfaulty(p, i) = goodclock$(p, A_p^{(0)}(T^{(0)}), A_p^{(i)}(T^{(i+1)}))$

Proof

x, y: VAR number

diminish: **Lemma** $x > 0 \supset \frac{\rho}{2} \times x \leq x$

diminish_proof: **Prove** diminish **from**
    mult_mon $\{x \leftarrow \frac{\rho}{2}, y \leftarrow 1, z \leftarrow x\}$,
    rho_small,
    mult_ax $\{x \leftarrow 1, y \leftarrow x\}$

monoproof: **Prove** monotonicity **from**
    gc_ax,
    diminish $\{x \leftarrow |T_1 - T_2|\}$,
    abs_ax $\{a \leftarrow c_p(T_1) - c_p(T_2) - (T_1 - T_2)\}$,
    abs_ax $\{a \leftarrow T_1 - T_2\}$

End clocks

algorithm: **Module**

Using clocks, sums

Exporting $\Sigma, \Delta, \Delta_{*1}^{(*2)}, \Delta_{*1,*2}^{(*3)}, \bar{\Delta}_{*1,*2}^{(*3)}$, skew, S1A, S1C, S2, $\delta, \epsilon, \delta_0,$
n, m **with** clocks

Theory

T, T_0, T_1, X, Π: VAR clocktime
i: VAR period
p, q, r: VAR proc
$\Delta_{*1}^{(*2)}$: function[proc, period → clocktime]
$\Delta_{*1,*2}^{(*3)}, \bar{\Delta}_{*1,*2}^{(*3)}$: function[proc, proc, period → clocktime]
m, n: proc
$\epsilon, \delta_0, \delta$: realtime
Σ, Δ: clocktime

C0_a: **Axiom** $n > 0$

C0_b: **Axiom** $0 \leq m \wedge m < n$

C0_c: **Axiom** $\Delta > 0$

C2: **Axiom** $S \geq \Sigma$

C3: **Axiom** $\Sigma \geq \Delta$

C4: **Axiom** $\Delta \geq \delta + \epsilon + \frac{\rho}{2} \times S$

C5: **Axiom** $\delta \geq \delta_0 + \rho * R$

C6: **Axiom** δ
$$\geq 2 * (\epsilon + \rho * S) + 2 * m * \Delta/(n - m) + n * \rho * R/(n - m) + \rho * \Delta + n * \rho * \Sigma/(n - m)$$

C2and3: **Lemma** $\Delta \leq S$

Alg1: **Axiom** $C_p^{(i+1)} = C_p^{(i)} + \Delta_p^{(i)}$

Alg2: **Axiom** $\Delta_p^{(i)} = \bigoplus_1^n(\lambda\, \mathrm{r} \rightarrow \mathrm{number} : \bar{\Delta}_{rp}^{(i)})$

Alg3: **Axiom** $\bar{\Delta}_{rp}^{(i)} =$ **if** $r \neq p \wedge |\Delta_{rp}^{(i)}| < \Delta$ **then** $\Delta_{rp}^{(i)}$ **else** 0 **end if** clock_prop: **Lemma** $c_p^{(i+1)}(T) = c_p^{(i)}(T + \Delta_p^{(i)})$

D2bar_prop: **Lemma** $|\bar{\Delta}_{pq}^{(i)}| < \Delta$

skew: function[proc, proc, clocktime, period → clocktime] =
$(\lambda$ p, q, T, i→ clocktime : $|c_p^{(i)}(T) - c_q^{(i)}(T)|)$
S1A: function[period → bool]

S1Adef: **Axiom** S1A(i) = ($\forall$ r: $(m + 1 \leq r \wedge r \leq n) \supset$ nonfaulty(r, i))

S1C: function[proc, proc, period → bool]

S1Cdef: **Axiom**
S1C(p, q, i)
= (nonfaulty$(p, i) \wedge$ nonfaulty$(q, i) \wedge T \in R^{(i)} \supset$ skew$(p, q, T, i) \leq \delta)$

S1C_lemma: **Lemma** S1C$(p, q, i) \supset$ S1C(q, p, i)

S2: function[proc, period → bool]

S2_ax: **Axiom** S2$(p, i) = (|C_p^{(i+1)} - C_p^{(i)}| < \Sigma)$

A0: **Axiom** skew$(p, q, T^{(0)}, 0) < \delta_0$

A2: **Axiom** nonfaulty$(p, i) \wedge$ nonfaulty$(q, i) \wedge$ S1C$(p, q, i) \wedge$ S2(p, i)
$\supset |\Delta_{qp}^{(i)}| \leq S$
$\wedge (\exists T_0 : T_0 \in S^{(i)} \wedge |c_p^{(i)}(T_0 + \Delta_{qp}^{(i)}) - c_q^{(i)}(T_0)| < \epsilon)$

A2_aux: **Axiom** $\Delta_{pp}^{(i)} = 0$

Theorem_1: **Theorem** S1A$(i) \supset$ S1C(p, q, i)

Theorem_2: **Theorem** S2(p, i)

Proof

C2and3_proof: **Prove** C2and3 **from** C2, C3

clock_proof: **Prove** clock_prop **from**
clockdef {T ← $T + \Delta_p^{(i)}$}, clockdef {i ← $i + 1$}, Alg1

D2bar_prop_proof: **Prove** D2bar_prop **from**
Alg3 {r ← p, p ← q}, C0_c, abs_ax0

S1C_lemma_proof: **Prove** S1C_lemma **from**
    S1Cdef,
    S1Cdef {p ← q, q ← p},
    abs_ax4 {x ← $c_q^{(i)}$(T@p1), y ← $c_p^{(i)}$(T@p1)}

Theorem_2_proof: **Prove** Theorem_2 **from**
    S2_ax,
    Alg1,
    D2bar_prop {p ← pp@p7, q ← p},
    Alg2,
    C0_a,
    C0_c,
    mean_bound {i ← 1,
      j ← n,
      x ← Δ,
      F ← (λ r→ number : $|\bar{\Delta}_{rp}^{(i)}|$)},
    abs_mean {i ← 1, j ← n, F ← (λ r→ number : $\bar{\Delta}_{rp}^{(i)}$)},
    C3

End algorithm

clockprops: **Module**

Using clocks, algorithm, natinduction

Theory

$T, T_0, T_1, T_2, T_N, \Pi$: VAR clocktime
p, q: VAR proc
i: VAR period

upper_bound: **Lemma**
$$T \in S^{(i)} \wedge |\Pi| \leq R - S \supset A_p^{(i)}(T + \Pi) \leq A_p^{(i+1)}(T^{(i+2)})$$

lower_bound: **Lemma** $0 \leq \Pi \supset A_p^{(0)}(T^{(0)}) \leq A_p^{(i)}(T^{(i)} + \Pi)$

lower_bound2: **Lemma**
$$T \in S^{(i)} \wedge |\Pi| \leq R - S \supset A_p^{(0)}(T^{(0)}) \leq A_p^{(i)}(T + \Pi)$$

adj_always_pos: **Lemma** $A_p^{(i)}(T^{(i)}) \geq T^0$

nonfx: **Lemma** nonfaulty$(p, i + 1) \supset$ nonfaulty(p, i)

S1A_lemma: **Lemma** S1A$(i + 1) \supset$ S1A(i)

Proof

i2R: **Lemma** $T^{(i+2)} = T^{(i)} + 2 * R$

i2R_proof: **Prove** i2R **from** T_sup_ax $\{i \leftarrow i + 2\}$, T_sup_ax

upper_bound_proof: **Prove** upper_bound **from**
Sdef,
i2R,
abs_ax6 $\{x \leftarrow \Pi, y \leftarrow R - S\}$,
S2_ax,
Theorem_2,
abs_ax6 $\{x \leftarrow C_p^{(i+1)} - C_p^{(i)}, y \leftarrow \Sigma\}$,
C2

basis: **Lemma** $A_p^{(0)}(T^{(0)}) \geq T^0$

basis_proof: **Prove** basis **from** zero_correction, T_sup_ax {i ← 0}

small_shift: **Lemma** $C_p^{(i+1)} - C_p^{(i)} \geq -R$

small_shift_proof: **Prove** small_shift **from**
S2_ax, Theorem_2, abs_ax {a ← $C_p^{(i+1)} - C_p^{(i)}$}, C2, SinR

inductive_step: **Lemma** $A_p^{(i)}(T^{(i)}) \geq T^0 \supset A_p^{(i+1)}(T^{(i+1)}) \geq T^0$

ind_proof: **Prove** inductive_step **from** small_shift, T_next

adj_pos_proof: **Prove** adj_always_pos **from**
induction {n ← i, prop ← (λ i→ bool : $A_p^{(i)}(T^{(i)}) \geq T^0$)},
basis,
inductive_step {i ← i@p1}

lower_bound_proof: **Prove** lower_bound **from**
adj_always_pos, T_sup_ax {i ← 0}, zero_correction

lower_bound2_proof: **Prove** lower_bound2 **from**
lower_bound {Π ← $T - T^{(i)} + \Pi$@c}, Sdef, abs_ax {a ← Π}, SinR

gc_prop: **Lemma**
$\text{goodclock}(p, T_0, T_N) \wedge T_0 \leq T \wedge T \leq T_N \supset \text{goodclock}(p, T_0, T)$

gc_proof: **Prove** gc_prop **from**
gc_ax {T_1 ← T_1@p2, T_2 ← T_2@p2}, gc_ax {T_N ← T}

bounds: **Lemma**
$A_p^{(0)}(T^{(0)}) \leq A_p^{(i)}(T^{(i+1)})$
$\wedge A_p^{(i)}(T^{(i+1)}) \leq A_p^{(i+1)}(T^{(i+2)})$

bounds_proof: **Prove** bounds **from**
upper_bound {Π ← 0, T ← $T^{(i+1)}$},
lower_bound2 {Π ← 0, T ← $T^{(i+1)}$},
abs_ax0,
SinR,
Ti_in_S

nonfx_proof: **Prove** nonfx **from**
A1 {i ← $i + 1$},
A1,
gc_prop {T_0 ← $A_p^{(0)}(T^{(0)})$,
T_N ← $A_p^{(i+1)}(T^{(i+2)})$,
T ← $A_p^{(i)}(T^{(i+1)})$},
bounds

S1A_lemma_proof: **Prove** S1A_lemma **from**
S1Adef, S1Adef {i ← $i + 1$, r ← r@p1}, nonfx {p ← r@p1}

End clockprops

lemma1: **Module**

Using algorithm, lemma2

Theory

p, q: VAR proc
i: VAR period

lemma1def: **Lemma**
$S1C(p,q,i) \wedge S2(p,i) \wedge nonfaulty(p,i+1) \wedge nonfaulty(q,i+1)$
$\supset |\Delta_{qp}^{(i)}| < \Delta$

Proof

lemma1_proof: **Prove** lemma1def **from**
A2,
lemma2c $\{\Pi \leftarrow \Delta_{qp}^{(i)}, T \leftarrow T_0@p1\}$,
S1Cdef $\{T \leftarrow T_0@p1\}$,
abs_ax4 $\{x \leftarrow c_p^{(i)}(T_0@p1), y \leftarrow c_q^{(i)}(T_0@p1)\}$,
abs_ax4 $\{x \leftarrow c_p^{(i)}(T_0@p1 + \Pi@p2), y \leftarrow c_p^{(i)}(T_0@p1) + \Pi@p2\}$,
abs_ax2b $\{x \leftarrow y@p5 - x@p5, y \leftarrow y@p4 - x@p4, z \leftarrow x@p5 - y@p4\}$,
nonfx,
nonfx $\{p \leftarrow q\}$,
inRS $\{T \leftarrow T_0@p1\}$,
mult4 $\{x \leftarrow \frac{\rho}{2}, y \leftarrow |\Delta_{qp}^{(i)}|, z \leftarrow S\}$,
rho_pos,
C4

End lemma1

lemma2: **Module**

Using algorithm, clockprops

Theory

p, q, r: VAR proc
i: VAR period
T: VAR clocktime
Π, Φ: VAR realtime

lemma2def: **Lemma**

$$\text{nonfaulty}(p, i+1) \wedge A_p^{(i)}(T) \leq A_p^{(i+1)}(T^{(i+2)}) \wedge A_p^{(0)}(T^{(0)}) \leq A_p^{(i)}(T) \wedge A_p^{(i)}(T+\Pi) \leq A_p^{(i+1)}(T^{(i+2)}) \wedge A_p^{(0)}(T^{(0)}) \leq A_p^{(i)}(T+\Pi) \supset |c_p^{(i)}(T+\Pi) - (c_p^{(i)}(T)+\Pi)| < \frac{\rho}{2} \times |\Pi|$$

lemma2a: **Lemma**

$$\text{nonfaulty}(p, i+1) \wedge |\Pi + \Phi| \leq R - S \wedge |\Phi| \leq R - S \wedge T \in S^{(i)} \supset |c_p^{(i)}(T+\Phi+\Pi) - (c_p^{(i)}(T+\Phi)+\Pi)| < \frac{\rho}{2} \times |\Pi|$$

lemma2b: **Lemma**

$$\text{nonfaulty}(p, i+1) \wedge |\Phi| \leq S \wedge |\Pi| \leq S \wedge T \in S^{(i)} \supset |c_p^{(i)}(T+\Phi+\Pi) - (c_p^{(i)}(T+\Phi)+\Pi)| < \frac{\rho}{2} \times |\Pi|$$

lemma2c: **Lemma**

$$\text{nonfaulty}(p, i+1) \wedge |\Pi| \leq S \wedge T \in S^{(i)} \supset |c_p^{(i)}(T+\Pi) - (c_p^{(i)}(T)+\Pi)| < \frac{\rho}{2} \times |\Pi|$$

lemma2d: **Lemma**

$$\text{nonfaulty}(p, i) \wedge 0 \leq \Pi \wedge \Pi \leq R \supset |c_p^{(i)}(T^{(i)}+\Pi) - (c_p^{(i)}(T^{(i)})+\Pi)| < \frac{\rho}{2} \times \Pi$$

Proof

lemma2_proof: **Prove** lemma2def **from**
A1 {i ← $i+1$},
gc_ax {$T_0 \leftarrow A_p^{(0)}(T^{(0)})$,
$T_N \leftarrow A_p^{(i+1)}(T^{(i+2)})$,
$T_2 \leftarrow A_p^{(i)}(T)$,
$T_1 \leftarrow A_p^{(i)}(T+\Pi)$},
clockdef,
clockdef {T ← $T+\Pi$}

lemma2a_proof: **Prove** lemma2a **from**
lemma2def {T ← $T+\Phi$},
upper_bound {$\Pi \leftarrow \Phi+\Pi$},
lower_bound2 {$\Pi \leftarrow \Phi+\Pi$},
upper_bound {$\Pi \leftarrow \Phi$},
lower_bound2 {$\Pi \leftarrow \Phi$}

lemma2b_proof: **Prove** lemma2b **from**
lemma2a, abs_ax1 {x ← Π}, abs_ax2 {x ← Φ, y ← Π}, C1, posS, posR

lemma2c_proof: **Prove** lemma2c **from** lemma2b {$\Phi \leftarrow 0$}, abs_ax0, posS

lemma2d_proof: **Prove** lemma2d **from**
A1,
gc_ax {$T_0 \leftarrow A_p^{(0)}(T^{(0)})$,
$T_N \leftarrow A_p^{(i)}(T^{(i+1)})$,
$T_1 \leftarrow A_p^{(i)}(T^{(i)}+\Pi)$,
$T_2 \leftarrow A_p^{(i)}(T^{(i)})$},
clockdef {T ← $T^{(i)}$},
clockdef {T ← $T^{(i)}+\Pi$},
posR,
pos_abs {x ← Π},
lower_bound,
lower_bound {$\Pi \leftarrow 0$},
T_next

End lemma2

lemma3: **Module**

Using algorithm, lemma2

Theory

p, q: VAR proc
i: VAR period
T, T_0, T_1, T_2: VAR clocktime
Π: VAR realtime

lemma3def: **Lemma**
  S1C(p, q, i)
    $\wedge$ S2(p, i) $\wedge$ nonfaulty($p, i+1$) $\wedge$ nonfaulty($q, i+1$) $\wedge\ T \in S^{(i)}$
  $\supset |c_p^{(i)}(T + \Delta_{qp}^{(i)}) - c_q^{(i)}(T)| < \epsilon + \rho * S$

Proof

lemma3_proof: **Prove** lemma3def **from**
  A2,
  rearrange_alt {x ← $c_p^{(i)}(T + \Delta_{qp}^{(i)})$,
   y ← $c_q^{(i)}(T)$,
   u ← $c_p^{(i)}(T_0@\mathrm{p1} + \Delta_{qp}^{(i)})$,
   v ← $T - T_0@\mathrm{p1}$,
   w ← $c_q^{(i)}(T_0@\mathrm{p1})$},
  lemma2b {T ← T_0@p1, Φ ← $\Delta_{qp}^{(i)}$, Π ← $T - T_0$@p1},
  lemma2c {p ← q, T ← T_0@p1, Π ← $T - T_0$@p1},
  nonfx,
  nonfx {p ← q},
  mult4 {x ← $\frac{\rho}{2}$, y ← $|T - T_0@\mathrm{p1}|$, z ← S},
  rho_pos,
  half3 {x ← ρ, y ← S},
  mult_ax {x ← ρ, y ← S},
  in_S_lemma {T_1 ← T, T_2 ← T_0@p1}

End lemma3

lemma4: **Module**

Using algorithm, lemma1, lemma2, lemma3

Theory

p, q, r: VAR proc
i: VAR period
T: VAR clocktime

lemma4def: **Lemma**

$$\begin{aligned}
&\mathrm{S1C}(q,r,i)\\
&\quad\wedge \mathrm{S1C}(p,q,i)\\
&\quad\quad\wedge \mathrm{S1C}(p,r,i)\\
&\quad\quad\quad\wedge \mathrm{S2}(p,i)\\
&\quad\quad\quad\quad\wedge \mathrm{S2}(q,i)\\
&\quad\quad\quad\quad\quad\wedge \mathrm{S2}(r,i)\\
&\quad\quad\quad\quad\quad\quad\wedge \mathrm{nonfaulty}(p,i+1)\\
&\quad\quad\quad\quad\quad\quad\quad\wedge \mathrm{nonfaulty}(q,i+1) \wedge \mathrm{nonfaulty}(r,i+1) \wedge T \in S^{(i)}\\
&\supset |c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)})| < 2 * (\epsilon + \rho * S + \tfrac{\rho}{2} \times \Delta)
\end{aligned}$$

Proof

T_0, T_1, T_2: VAR clocktime
Π: VAR realtime
u, v, w, x, y, z: VAR number

rearrange1: **Lemma** $x - y = (u - y) - (v - x) + (v - w) - (u - w)$

rearrange1_proof: **Prove** rearrange1

rearrange2: **Lemma**

$$\begin{aligned}
&|(u-y) - (v-x) + (v-w) - (u-w)|\\
&\quad \le |u-y| + |v-x| + |v-w| + |u-w|
\end{aligned}$$

rearrange2_proof: **Prove** rearrange2 **from**
abs_ax2c {w ← $(u - y)$, x ← $(x - v)$, y ← $(v - w)$, z ← $(w - u)$},
abs_ax3 {x ← $(v - x)$},
abs_ax3 {x ← $(u - w)$}

rearrange3: **Lemma** $|x - y| \le |u - y| + |v - x| + |v - w| + |u - w|$

rearrange3_proof: **Prove** rearrange3 **from** rearrange1, rearrange2

sublemma1: **Lemma**

$\text{S1C}(p, r, i) \wedge \text{S2}(p, i) \wedge \text{nonfaulty}(p, i+1) \wedge \text{nonfaulty}(r, i+1)$
$\supset \bar{\Delta}_{rp}^{(i)} = \Delta_{rp}^{(i)}$

sublemma1_proof: **Prove** sublemma1 **from** lemma1def {q ← r}, Alg3, A2_aux

lemma2x: **Lemma**

$\text{S1C}(p, r, i)$
$\wedge\ \text{S2}(p, i) \wedge \text{nonfaulty}(p, i+1) \wedge \text{nonfaulty}(r, i+1) \wedge T \in S^{(i)}$
$\supset |c_p^{(i)}(T + \Delta_{rp}^{(i)}) - (c_p^{(i)}(T) + \Delta_{rp}^{(i)})| < \frac{\rho}{2} \times \Delta$

lemma2x_proof: **Prove** lemma2x **from**
lemma2c {Π ← $\Delta_{rp}^{(i)}$},
lemma1def {q ← r},
C2and3,
mult4 {x ← $\frac{\rho}{2}$, y ← $|\Delta_{rp}^{(i)}|$, z ← Δ},
rho_pos

lemma4_proof: **Prove** lemma4def **from**
rearrange3 {x ← $c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)}$,
y ← $c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)}$,
u ← $c_q^{(i)}(T + \Delta_{rq}^{(i)})$,
v ← $c_p^{(i)}(T + \Delta_{rp}^{(i)})$,
w ← $c_r^{(i)}(T)$},
sublemma1,
sublemma1 {p ← q},
lemma2x,
lemma2x {p ← q},
lemma3def {q ← r},
lemma3def {p ← q, q ← r},
S1C_lemma

End lemma4

lemma5: **Module**

Using algorithm, clockprops

Theory

p, q, r: VAR proc
T: VAR clocktime
i, j: VAR period

lemma5def: **Lemma**
$S1C(p, q, i) \wedge \text{nonfaulty}(p, i+1) \wedge \text{nonfaulty}(q, i+1) \wedge T \in S^{(i)}$
$\supset |c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)})| < \delta + 2 * \Delta$

Proof

a, b, x, y: VAR clocktime

rearrange1: **Lemma** $(a + x) - (b + y) = (a - b) + x - y$

rearrange1_proof: **Prove** rearrange1

rearrange2: **Lemma** $|(a + x) - (b + y)| \leq |a - b| + |x| + |y|$

rearrange2_proof: **Prove** rearrange2 **from**
rearrange1, abs_ax8, abs_ax2 {x ← $(a - b)$, y ← $(x - y)$}

lemma5proof: **Prove** lemma5def **from**
rearrange2 {a ← $c_p^{(i)}(T)$,
b ← $c_q^{(i)}(T)$,
x ← $\bar{\Delta}_{rp}^{(i)}$,
y ← $\bar{\Delta}_{rq}^{(i)}$},
D2bar_prop {p ← r, q ← p},
D2bar_prop {p ← r, q ← q},
inRS,
S1Cdef,
nonfx,
nonfx {p ← q}

End lemma5

lemma6: **Module**

Using algorithm, clockprops, lemma2

Theory

p, q: VAR proc
i: VAR period
T, Π: VAR clocktime

sublemma_A: **Lemma**

$$\text{nonfaulty}(p, i) \wedge \text{nonfaulty}(q, i) \wedge T \in R^{(i)} \supset \text{skew}(p, q, T, i) < \text{skew}(p, q, T^{(i)}, i) + \rho * R$$

lemma6def: **Lemma**

$$\text{nonfaulty}(p, i+1) \wedge \text{nonfaulty}(q, i+1) \wedge T \in R^{(i+1)} \supset \text{skew}(p, q, T, i+1) < |c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)})| + \rho * R + \rho * \Sigma$$

Proof

sublemma1: **Lemma** $0 \leq \Pi \wedge \Pi \leq R \supset 2 * \frac{\rho}{2} \times \Pi \leq \rho * R$

sub1_proof: **Prove** sublemma1 **from**
mult2 $\{x \leftarrow \frac{\rho}{2}, y \leftarrow R\}$,
times_half $\{x \leftarrow \rho\}$,
mult4 $\{x \leftarrow \frac{\rho}{2}, y \leftarrow \Pi, z \leftarrow R\}$,
rho_pos,
mult_ax $\{x \leftarrow \rho, y \leftarrow R\}$

sub_A_proof: **Prove** sublemma_A **from**
Rdef,
rearrange_alt $\{x \leftarrow c_p^{(i)}(T)$,
$y \leftarrow c_q^{(i)}(T)$,
$u \leftarrow c_p^{(i)}(T^{(i)})$,
$v \leftarrow \Pi@p1$,
$w \leftarrow c_q^{(i)}(T^{(i)})\}$,
lemma2d $\{\Pi \leftarrow \Pi@p1\}$,
lemma2d $\{p \leftarrow q, \Pi \leftarrow \Pi@p1\}$,
sublemma1 $\{\Pi \leftarrow \Pi@p1\}$

sublemma2: **Lemma**
$\text{skew}(p, q, T, i+1) = |c_p^{(i)}(T + \Delta_p^{(i)}) - c_q^{(i)}(T + \Delta_q^{(i)})|$

sub2_proof: **Prove** sublemma2 **from** clock_prop, clock_prop $\{p \leftarrow q\}$

lemma6_proof: **Prove** lemma6def **from**
sublemma_A $\{i \leftarrow i+1\}$,
sublemma2 $\{T \leftarrow T^{(i+1)}\}$,
rearrange $\{x \leftarrow c_p^{(i)}(T^{(i+1)} + \Delta_p^{(i)})$,
$y \leftarrow c_q^{(i)}(T^{(i+1)} + \Delta_q^{(i)})$,
$u \leftarrow c_p^{(i)}(T^{(i+1)})$,
$v \leftarrow \Delta_p^{(i)}$,
$w \leftarrow c_q^{(i)}(T^{(i+1)})$,
$z \leftarrow \Delta_q^{(i)}\}$,
lemma2c $\{T \leftarrow T^{(i+1)}, \Pi \leftarrow \Delta_p^{(i)}\}$,
lemma2c $\{T \leftarrow T^{(i+1)}, \Pi \leftarrow \Delta_q^{(i)}, p \leftarrow q\}$,
Alg1,
Alg1 $\{p \leftarrow q\}$,
S2_ax,
S2_ax $\{p \leftarrow q\}$,
Theorem_2,
Theorem_2 $\{p \leftarrow q\}$,
mult4 $\{x \leftarrow \frac{\rho}{2}, y \leftarrow |\Delta_p^{(i)}|, z \leftarrow \Sigma\}$,
mult4 $\{x \leftarrow \frac{\rho}{2}, y \leftarrow |\Delta_q^{(i)}|, z \leftarrow \Sigma\}$,
rho_pos,
Ti_in_S,
C2,
half3 $\{x \leftarrow \rho, y \leftarrow \Sigma\}$,
mult_ax $\{x \leftarrow \rho, y \leftarrow \Sigma\}$

End lemma6

summations: **Module**

Using algorithm, sums, lemma4, lemma5, lemma6

Theory

p, q, r: VAR proc
T: VAR clocktime
i: VAR period

culmination: **Lemma**

$$\begin{aligned}
&\mathrm{S1A}(i+1) \wedge \mathrm{S1C}(p,q,i) \\
&\quad \supset (\mathrm{nonfaulty}(p,i+1) \wedge \mathrm{nonfaulty}(q,i+1) \wedge T \in R^{(i+1)} \\
&\qquad \supset \mathrm{skew}(p,q,T,i+1) \\
&\qquad\quad \leq ((\delta + 2 * \Delta) * m + 2 * (\rho * S + \epsilon + \tfrac{\rho}{2} \times \Delta) * (n-m))/n \\
&\qquad\qquad + \rho * R + \rho * \Sigma)
\end{aligned}$$

Proof

l1: **Lemma**
$$\begin{aligned}
&|c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)})| \\
&\quad \leq \bigoplus_1^n(\lambda\, \mathrm{r} \rightarrow \mathrm{number}: \\
&\qquad |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)})|)
\end{aligned}$$

l2: **Lemma**
$$\begin{aligned}
&|c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)})| \\
&\quad \leq (\textstyle\sum_1^m(\lambda\, \mathrm{r} \rightarrow \mathrm{number}: \\
&\qquad\qquad |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)})|) \\
&\qquad + \textstyle\sum_{m+1}^n(\lambda\, \mathrm{r} \rightarrow \mathrm{number}: \\
&\qquad\qquad |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)})|)) \\
&\quad /n
\end{aligned}$$

l3: **Lemma**
$$\begin{aligned}
&\mathrm{S1A}(i+1) \\
&\quad \wedge \mathrm{S1C}(p,q,i) \wedge \mathrm{nonfaulty}(p,i+1) \wedge \mathrm{nonfaulty}(q,i+1) \\
&\quad \supset \textstyle\sum_1^m(\lambda\, \mathrm{r} \rightarrow \mathrm{number}: \\
&\qquad\qquad |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)})|) \\
&\qquad \leq (\delta + 2 * \Delta) * m
\end{aligned}$$

l4: **Lemma** S1A($i+1$)
$\wedge$ S1C(p,q,i) $\wedge$ nonfaulty($p,i+1$) $\wedge$ nonfaulty($q,i+1$)
$\supset \sum_{m+1}^{n}(\lambda$ r$\rightarrow$ number :
$|c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)})|)$
$\leq 2 * (\rho * S + \epsilon + \frac{\rho}{2} \times \Delta) * (n - m)$

l5: **Lemma** S1A($i+1$)
$\wedge$ S1C(p,q,i) $\wedge$ nonfaulty($p,i+1$) $\wedge$ nonfaulty($q,i+1$)
$\supset |c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)})|$
$\leq ((\delta + 2 * \Delta) * m + 2 * (\rho * S + \epsilon + \frac{\rho}{2} \times \Delta) * (n - m))/n$

l1_proof: **Prove** l1 **from**
Alg2,
Alg2 {p $\leftarrow q$},
rearrange_sum {x $\leftarrow c_p^{(i)}(T^{(i+1)})$,
y $\leftarrow c_q^{(i)}(T^{(i+1)})$,
F $\leftarrow$ (λ r$\rightarrow$ number : $\bar{\Delta}_{rp}^{(i)}$),
G $\leftarrow$ (λ r$\rightarrow$ number : $\bar{\Delta}_{rq}^{(i)}$),
i $\leftarrow$ 1,
j $\leftarrow n$},
abs_mean {i $\leftarrow$ 1,
j $\leftarrow n$,
F $\leftarrow$ (λ r$\rightarrow$ number : x@p3 + $\bar{\Delta}_{rp}^{(i)}$ − (y@p3 + $\bar{\Delta}_{rq}^{(i)}$))},
C0_a

l2_proof: **Prove** l2 **from**
l1,
split_mean {i $\leftarrow$ 1,
j $\leftarrow n$,
k $\leftarrow m$,
F $\leftarrow$ (λ r$\rightarrow$ number :
$|c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)})|$)},
C0_a,
C0_b

bound_faulty: **Lemma**
$\text{S1A}(i+1) \wedge \text{S1C}(p,q,i)$
$\wedge\, 1 \leq r \wedge r \leq m \wedge \text{nonfaulty}(p, i+1) \wedge \text{nonfaulty}(q, i+1)$
$\supset |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)})|$
$< \delta + 2 * \Delta$

bound_faulty_proof: **Prove** bound_faulty **from**
lemma5def $\{T \leftarrow T^{(i+1)}\}$, Ti_in_S

l3_proof: **Prove** l3 **from**
sum_bound {F ← (λ r→ number :
$|c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)})|$),
x ← $\delta + 2 * \Delta$,
i ← 1,
j ← m},
bound_faulty {r ← pp@p1},
C0_b

S2_pqr: **Lemma** $\text{S2}(p,i) \wedge \text{S2}(q,i) \wedge \text{S2}(r,i)$

S2_pqr_proof: **Prove** S2_pqr **from**
Theorem_2, Theorem_2 {p ← q}, Theorem_2 {p ← r}

bound_nonfaulty: **Lemma**
$\text{S1A}(i+1) \wedge \text{S1C}(p,q,i)$
$\wedge\, m+1 \leq r \wedge r \leq n \wedge \text{nonfaulty}(p, i+1) \wedge \text{nonfaulty}(q, i+1)$
$\supset |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)})|$
$< 2 * (\rho * S + \epsilon + \frac{\rho}{2} \times \Delta)$

bound_nonfaulty_proof: **Prove** bound_nonfaulty **from**
    S1Adef {i ← $i + 1$},
    S1A_lemma,
    S1Adef,
    nonfx,
    nonfx {p ← q},
    Theorem_1 {q ← r},
    Theorem_1 {p ← q, q ← r},
    S2_pqr,
    lemma4def {T ← $T^{(i+1)}$},
    Ti_in_S

l4_proof: **Prove** l4 **from**
    sum_bound {F ← (λ r→ number :
        $|c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)})|$),
      x ← $2 * (\rho * S + \epsilon + \frac{\rho}{2} \times \Delta)$,
      i ← $m + 1$,
      j ← n},
    bound_nonfaulty {r ← pp@p1},
    C0_b

l5_proof: **Prove** l5 **from**
    l2,
    l3,
    l4,
    div_mon2 {x ← $\sum_1^m$(λ r→ number :
        $|c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)})|$)
      + $\sum_{m+1}^n$(λ r→ number :
        $|c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rp}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{rq}^{(i)})|$),
      y ← $(\delta + 2 * \Delta) * m + 2 * (\rho * S + \epsilon + \frac{\rho}{2} \times \Delta) * (n - m)$,
      z ← n},
    C0_a

culm_proof: **Prove** culmination **from** lemma6def, l5, S1Adef {i ← $i + 1$}

End summations

juggle: **Module**

Using algorithm

Theory

rearrange_delta: **Lemma**

$$\delta \geq 2 * (\epsilon + \rho * S) + 2 * m * \Delta/(n-m) + n * \rho * R/(n-m) + \rho * \Delta + n * \rho * \Sigma/(n-m) \supset \delta \geq ((\delta + 2 * \Delta) * m + 2 * (\epsilon + \rho * S + \tfrac{\rho}{2} \times \Delta) * (n-m))/n + \rho * R + \rho * \Sigma$$

Proof

a, b, b1, b2, b3, b4, b5, b6, c, x, y: VAR number

distrib6: **Lemma**

(b1 + b2 + b3 + b4 + b5 + b6) $* c$
= b1 $* c$ + b2 $* c$ + b3 $* c$ + b4 $* c$ + b5 $* c$ + b6 $* c$

distrib6_proof: **Prove** distrib6

distrib6_mult: **Lemma**

(b1 + b2 + b3 + b4 + b5 + b6) $\times c$
= b1 $\times c$ + b2 $\times c$ + b3 $\times c$ + b4 $\times c$ + b5 $\times c$ + b6 $\times c$

distrib6_mult_proof: **Prove** distrib6_mult **from**
distrib6,
mult_ax {x ← b1 + b2 + b3 + b4 + b5 + b6, y ← c},
mult_ax {x ← b1, y ← c},
mult_ax {x ← b2, y ← c},
mult_ax {x ← b3, y ← c},
mult_ax {x ← b4, y ← c},
mult_ax {x ← b5, y ← c},
mult_ax {x ← b6, y ← c}

mult_ineq1: **Lemma**

$a \geq$ b1 + b2 + b3 + b4 + b5 $\wedge\ c > 0$
$\supset a \times c \geq$ b1 $\times c$ + b2 $\times c$ + b3 $\times c$ + b4 $\times c$ + b5 $\times c$

mult_ineq1_proof: **Prove** mult_ineq1 **from**
distrib6_mult {b6 ← 0},
mult_mon2 {x ← b1 + b2 + b3 + b4 + b5, y ← a, z ← c},
mult_ax {x ← 0, y ← c}

distrib6_div: **Lemma**
$c > 0 \supset (\text{b1} + \text{b2} + \text{b3} + \text{b4} + \text{b5} + \text{b6})/c$
$= \text{b1}/c + \text{b2}/c + \text{b3}/c + \text{b4}/c + \text{b5}/c + \text{b6}/c$

reciprocal: **Lemma** $y \neq 0 \supset x \times 1/y = x/y$

reciprocal_proof: **Prove** reciprocal **from**
quotient_ax, mult_ax {y ← $1/y$}

distrib6_div_proof: **Prove** distrib6_div **from**
distrib6_mult {c ← $1/c$},
reciprocal {x ← b1 + b2 + b3 + b4 + b5 + b6, y ← c},
reciprocal {x ← b1, y ← c},
reciprocal {x ← b2, y ← c},
reciprocal {x ← b3, y ← c},
reciprocal {x ← b4, y ← c},
reciprocal {x ← b5, y ← c},
reciprocal {x ← b6, y ← c}

cancel_mult: **Lemma** $c > 0 \wedge a \times c \geq b \supset a \geq b/c$

cancel_mult_proof: **Prove** cancel_mult **from**
div_mon2 {z ← c, x ← b, y ← $a \times c$},
cancellation_mult {x ← a, y ← c}

mult_ineq2: **Lemma**
$c > 0 \wedge a \times c \geq \text{b1} + \text{b2} + \text{b3} + \text{b4} + \text{b5} + \text{b6}$
$\supset a \geq \text{b1}/c + \text{b2}/c + \text{b3}/c + \text{b4}/c + \text{b5}/c + \text{b6}/c$

mult_ineq2_proof: **Prove** mult_ineq2 **from**
cancel_mult {b ← b1 + b2 + b3 + b4 + b5 + b6}, distrib6_div

distrib4_div: **Lemma**
$c > 0 \supset \text{b1}/c + \text{b2}/c + \text{b3}/c + \text{b4}/c = (\text{b1} + \text{b2} + \text{b3} + \text{b4})/c$

distrib4_div_proof: **Prove** distrib4_div **from**
distrib6_mult {b5 ← 0, b6 ← 0, c ← $1/c$},
reciprocal {x ← b1 + b2 + b3 + b4, y ← c},
reciprocal {x ← b1, y ← c},
reciprocal {x ← b2, y ← c},
reciprocal {x ← b3, y ← c},
reciprocal {x ← b4, y ← c},
mult_ax {x ← 0, y ← $1/c$}

step1: **Lemma**
$$\delta \geq 2 * (\epsilon + \rho * S) + 2 * m * \Delta/(n - m) + n * \rho * R/(n - m) + \rho * \Delta + n * \rho * \Sigma/(n - m)$$
$$\supset \delta \times n - m \geq 2 * (\epsilon + \rho * S) \times n - m + 2 * m * \Delta + n * \rho * R + \rho * \Delta \times n - m + n * \rho * \Sigma$$

step1_proof: **Prove** step1 **from**
mult_ineq1 {a ← δ,
c ← $n - m$,
b1 ← $2 * (\epsilon + \rho * S)$,
b2 ← $2 * m * \Delta/(n - m)$,
b3 ← $n * \rho * R/(n - m)$,
b4 ← $\rho * \Delta$,
b5 ← $n * \rho * \Sigma/(n - m)$},
mult_div {x ← $2 * m * \Delta$, y ← $n - m$},
mult_div {x ← $n * \rho * R$, y ← $n - m$},
mult_div {x ← $n * \rho * \Sigma$, y ← $n - m$},
C0_b

step2: **Lemma**
$$\delta \times n - m \geq 2 * (\epsilon + \rho * S) \times n - m + 2 * m * \Delta + n * \rho * R + \rho * \Delta \times n - m + n * \rho * \Sigma$$
$$\supset \delta \times n \geq \delta \times m + 2 * (\epsilon + \rho * S) \times n - m + 2 * m * \Delta + n * \rho * R + \rho * \Delta \times n - m + n * \rho * \Sigma$$

step2_proof: **Prove** step2 **from**
    mult_ax {x ← δ, y ← $n - m$},
    mult_ax {x ← δ, y ← n},
    mult_ax {x ← δ, y ← m}

step3: **Lemma**

$$\begin{aligned}
&\delta \times n \geq \delta \times m + 2 * (\epsilon + \rho * S) \times n - m + 2 * m * \Delta + n * \rho * R \\
&\qquad + \rho * \Delta \times n - m \\
&\quad + n * \rho * \Sigma \\
&\supset \delta \geq \delta \times m/n + 2 * (\epsilon + \rho * S) \times n - m/n + 2 * m * \Delta/n + \rho * R \\
&\qquad + \rho * \Delta \times n - m/n \\
&\quad + \rho * \Sigma
\end{aligned}$$

step3_proof: **Prove** step3 **from**
    mult_ineq2 {a ← δ,
      c ← n,
      b1 ← $\delta \times m$,
      b2 ← $2 * (\epsilon + \rho * S) \times n - m$,
      b3 ← $2 * m * \Delta$,
      b4 ← $n * \rho * R$,
      b5 ← $\rho * \Delta \times n - m$,
      b6 ← $n * \rho * \Sigma$},
    cancellation {x ← $\rho * R$, y ← n},
    cancellation {x ← $\rho * \Sigma$, y ← n},
    C0_a

step4: **Lemma**

$$\begin{aligned}
&\delta \geq \delta \times m/n + 2 * (\epsilon + \rho * S) \times n - m/n + 2 * m * \Delta/n + \rho * R \\
&\qquad + \rho * \Delta \times n - m/n \\
&\quad + \rho * \Sigma \\
&\supset \delta \geq (\delta \times m + 2 * (\epsilon + \rho * S) \times n - m + 2 * m * \Delta + \rho * \Delta \times n - m)/n \\
&\qquad + \rho * R \\
&\quad + \rho * \Sigma
\end{aligned}$$

step4_proof: **Prove** step4 **from**
C0_a,
distrib4_div {c ← n,
b1 ← $\delta \times m$,
b2 ← $2 * (\epsilon + \rho * S) \times n - m$,
b3 ← $2 * m * \Delta$,
b4 ← $\rho * \Delta \times n - m$}

step5: **Lemma**

$$\delta \geq (\delta \times m + 2 * (\epsilon + \rho * S) \times n - m + 2 * m * \Delta + \rho * \Delta \times n - m)/n + \rho * R + \rho * \Sigma$$
$$\supset \delta \geq ((\delta + 2 * \Delta) * m + 2 * (\epsilon + \rho * S + \tfrac{\rho}{2} \times \Delta) * (n - m))/n + \rho * R + \rho * \Sigma$$

step5_proof: **Prove** step5 **from**
mult_ax {x ← δ, y ← m},
mult_ax {x ← $\rho * \Delta$, y ← $n - m$},
mult_ax {x ← $2 * (\epsilon + \rho * S)$, y ← $n - m$},
half3 {x ← ρ, y ← Δ},
mult_ax {x ← ρ, y ← Δ}

final: **Prove** rearrange_delta **from** step1, step2, step3, step4, step5

End juggle

main: **Module**

Using natinduction, algorithm, lemma6, summations, juggle

Proof

p, q, r: VAR proc
i, j, k: VAR period
T: VAR clocktime

basis: **Lemma** S1A(0) $\supset$ S1C(p, q, 0)

basis_proof: **Prove** basis **from**
S1Adef {i $\leftarrow$ 0}, sublemma_A {i $\leftarrow$ 0}, S1Cdef {i $\leftarrow$ 0}, A0, C5

ind_step: **Lemma** S1A(i + 1) $\wedge$ S1C(p, q, i) $\supset$ S1C(p, q, i + 1)

ind_proof: **Prove** ind_step **from**
culmination, rearrange_delta, S1Cdef {i $\leftarrow$ i + 1}, C6

Theorem_1_proof: **Prove** Theorem_1 **from**
basis,
ind_step {i $\leftarrow$ i@p3},
mod_induction {n $\leftarrow$ i,
A $\leftarrow$ (λ k$\rightarrow$ bool : S1A(k)),
B $\leftarrow$ (λ k$\rightarrow$ bool : S1C(p, q, k))},
S1A_lemma {i $\leftarrow$ j@p3}

End main

Appendix C

Proof-Chain Analysis

This Appendix reproduces the output from the EHDM Proof Chain Analyzer for the two Theorems proved in the specification.

C.1 Clock Synchronization Condition S2

The proof chain for Theorem_2 in the specification is given below in full. It can be seen that the proof chain is complete.

```
Proof chain for formula Theorem_2 in module algorithm

algorithm.Theorem_2
  is the conclusion of the proof
    algorithm.Theorem_2_proof

Proof algorithm.Theorem_2_proof (which is PROVED) establishes
  algorithm.Theorem_2

Its premises are:
  algorithm.S2_ax
  algorithm.Alg1
  algorithm.D2bar_prop
  algorithm.Alg2
  algorithm.CO_a
  algorithm.CO_c
  sums.mean_bound
  sums.abs_mean
  algorithm.C3
```

```
algorithm.S2_ax
  is an axiom

algorithm.Alg1
  is an axiom

algorithm.D2bar_prop
  is the conclusion of the proof
    algorithm.D2bar_prop_proof

Proof algorithm.D2bar_prop_proof (which is PROVED) establishes
  algorithm.D2bar_prop

Its premises are:
  algorithm.Alg3
  algorithm.CO_c
  absolutes.abs_ax0

algorithm.Alg3
  is an axiom

algorithm.CO_c
  is an axiom

absolutes.abs_ax0
  is the conclusion of the proof
    absolutes.abs_proof0

Proof absolutes.abs_proof0 (which is PROVED) establishes
  absolutes.abs_ax0

Its premises are:
  absolutes.abs_ax

absolutes.abs_ax
  is an axiom

algorithm.Alg2
  is an axiom

algorithm.CO_a
  is an axiom

algorithm.CO_c
```

```
  has already been justified

sums.mean_bound
  is the conclusion of the proof
    sums.mean_bound_proof

Proof sums.mean_bound_proof (which is PROVED) establishes
  sums.mean_bound

Its premises are:
  sums.sum_bound1
  sums.mean_ax
  arithmetics.div_prod

sums.sum_bound1
  is the conclusion of the proof
    sums.sum_bound1_proof

Proof sums.sum_bound1_proof (which is PROVED) establishes
  sums.sum_bound1

Its premises are:
  sums.sum_bound_mod
  arithmetics.mult_ax

sums.sum_bound_mod
  is the conclusion of the proof
    sums.sum_bound_mod_proof

Proof sums.sum_bound_mod_proof (which is PROVED) establishes
  sums.sum_bound_mod

Its premises are:
  sums.sum_ax
  sums.sigma_bound2
  natprops.pred_diff
  natprops.diff_ax
  natprops.diff_ax

sums.sum_ax
  is an axiom

sums.sigma_bound2
  is the conclusion of the proof
```

```
    sums.sigma_bound2_proof

Proof sums.sigma_bound2_proof (which is PROVED) establishes
  sums.sigma_bound2

Its premises are:
  sigmaprops.sigma_bound
  arithmetics.mult_ax

sigmaprops.sigma_bound
  is the conclusion of the proof
    sigmaprops.sigma_bound_proof

Proof sigmaprops.sigma_bound_proof (which is PROVED) establishes
  sigmaprops.sigma_bound

Its premises are:
  sigmaprops.sb
  sigmaprops.bounded_ax

sigmaprops.sb
  is the conclusion of the proof
    sigmaprops.sb_proof

Proof sigmaprops.sb_proof (which is PROVED) establishes
  sigmaprops.sb

Its premises are:
  natinduction.mod_induction1
  sigmaprops.bounded_lemma
  sigmaprops.sigma_bound_basis
  sigmaprops.sigma_bound_step

natinduction.mod_induction1
  is the conclusion of the proof
    natinduction.mod_induction1_proof

Proof natinduc-
tion.mod_induction1_proof (which is PROVED) establishes
  natinduction.mod_induction1

Its premises are:
  natinduction.mod_induction_m
```

```
natinduction.mod_induction_m
  is the conclusion of the proof
    natinduction.mod_m_proof

Proof natinduction.mod_m_proof (which is PROVED) establishes
  natinduction.mod_induction_m

Its premises are:
  natinduction.induction_m

natinduction.induction_m
  is an axiom

sigmaprops.bounded_lemma
  is the conclusion of the proof
    sigmaprops.bounded_proof

Proof sigmaprops.bounded_proof (which is PROVED) establishes
  sigmaprops.bounded_lemma

Its premises are:
  sigmaprops.bounded_ax
  sigmaprops.bounded_ax
  natprops.pred_lemma
  natprops.pred_ax

sigmaprops.bounded_ax
  is an axiom

sigmaprops.bounded_ax
  has already been justified

natprops.pred_lemma
  is the conclusion of the proof
    natprops.pred_lemma_proof

Proof natprops.pred_lemma_proof (which is PROVED) establishes
  natprops.pred_lemma

Its premises are:
  natprops.pred_ax
  natprops.natpos

natprops.pred_ax
```

```
  is an axiom

natprops.natpos
  is an axiom

natprops.pred_ax
  has already been justified

sigmaprops.sigma_bound_basis
  is the conclusion of the proof
    sigmaprops.sb_basis_proof

Proof sigmaprops.sb_basis_proof (which is PROVED) establishes
  sigmaprops.sigma_bound_basis

Its premises are:
  sigmaprops.bounded_ax
  sigmaprops.sigma_ax
  sigmaprops.sigma_ax
  natprops.pred_ax

sigmaprops.bounded_ax
  has already been justified

sigmaprops.sigma_ax
  is an axiom

sigmaprops.sigma_ax
  has already been justified

natprops.pred_ax
  has already been justified

sigmaprops.sigma_bound_step
  is the conclusion of the proof
    sigmaprops.sb_step_proof

Proof sigmaprops.sb_step_proof (which is PROVED) establishes
  sigmaprops.sigma_bound_step

Its premises are:
  sigmaprops.alt_sigma_bound_step
  arithmetics.mult_ax
```

```
sigmaprops.alt_sigma_bound_step
  is the conclusion of the proof
    sigmaprops.alt_sb_step_proof

Proof sigmaprops.alt_sb_step_proof (which is PROVED) establishes
  sigmaprops.alt_sigma_bound_step

Its premises are:
  sigmaprops.bounded_ax
  sigmaprops.sigma_ax
  natprops.pred_lemma
  natprops.natpos

sigmaprops.bounded_ax
  has already been justified

sigmaprops.sigma_ax
  has already been justified

natprops.pred_lemma
  has already been justified

natprops.natpos
  has already been justified

arithmetics.mult_ax
  is an axiom

sigmaprops.bounded_ax
  has already been justified

arithmetics.mult_ax
  has already been justified

natprops.pred_diff
  is the conclusion of the proof
    natprops.pred_diff_proof

Proof natprops.pred_diff_proof (which is PROVED) establishes
  natprops.pred_diff

Its premises are:
  natprops.pred_ax
  natprops.diff_ax
```

```
  natprops.diff_ax

natprops.pred_ax
  has already been justified

natprops.diff_ax
  is an axiom

natprops.diff_ax
  has already been justified

natprops.diff_ax
  has already been justified

natprops.diff_ax
  has already been justified

arithmetics.mult_ax
  has already been justified

sums.mean_ax
  is an axiom

arithmetics.div_prod
  is the conclusion of the proof
    arithmetics.div_prod_proof

Proof arithmetics.div_prod_proof (which is PROVED) establishes
  arithmetics.div_prod

Its premises are:
  arithmetics.div_mult
  arithmetics.mult_ax

arithmetics.div_mult
  is the conclusion of the proof
    arithmetics.div_mult_proof

Proof arithmetics.div_mult_proof (which is PROVED) establishes
  arithmetics.div_mult

Its premises are:
  arithmetics.div_mon
  arithmetics.cancellation_mult
```

```
arithmetics.div_mon
  is the conclusion of the proof
    arithmetics.div_mon_proof

Proof arithmetics.div_mon_proof (which is PROVED) establishes
  arithmetics.div_mon

Its premises are:
  arithmetics.mult_mon
  arithmetics.quotient_mult
  arithmetics.quotient_mult
  arithmetics.quotient_ax2

arithmetics.mult_mon
  is an axiom

arithmetics.quotient_mult
  is the conclusion of the proof
    arithmetics.quotient_mult_proof

Proof arithmetics.quotient_mult_proof (which is PROVED) establishes
  arithmetics.quotient_mult

Its premises are:
  arithmetics.quotient_ax
  arithmetics.mult_ax

arithmetics.quotient_ax
  is an axiom

arithmetics.mult_ax
  has already been justified

arithmetics.quotient_mult
  has already been justified

arithmetics.quotient_ax2
  is an axiom

arithmetics.cancellation_mult
  is the conclusion of the proof
    arithmetics.cancellation_mult_proof
```

```
Proof arith-
metics.cancellation_mult_proof (which is PROVED) establishes
  arithmetics.cancellation_mult

Its premises are:
  arithmetics.cancellation
  arithmetics.mult_ax

arithmetics.cancellation
  is the conclusion of the proof
    arithmetics.cancellation_proof

Proof arithmetics.cancellation_proof (which is PROVED) establishes
  arithmetics.cancellation

Its premises are:
  arithmetics.div_times
  arithmetics.quotient_ax1

arithmetics.div_times
  is the conclusion of the proof
    arithmetics.div_times_proof

Proof arithmetics.div_times_proof (which is PROVED) establishes
  arithmetics.div_times

Its premises are:
  arithmetics.quotient_ax
  arithmetics.quotient_ax

arithmetics.quotient_ax
  has already been justified

arithmetics.quotient_ax
  has already been justified

arithmetics.quotient_ax1
  is an axiom

arithmetics.mult_ax
  has already been justified

arithmetics.mult_ax
  has already been justified
```

```
sums.abs_mean
  is the conclusion of the proof
    sums.abs_mean_proof

Proof sums.abs_mean_proof (which is PROVED) establishes
  sums.abs_mean

Its premises are:
  sums.mean_ax
  sums.mean_ax
  sums.abs_sum
  arithmetics.abs_div2
  arithmetics.div_mon2
  absolutes.abs_ax0

sums.mean_ax
  has already been justified

sums.mean_ax
  has already been justified

sums.abs_sum
  is the conclusion of the proof
    sums.abs_sum_proof

Proof sums.abs_sum_proof (which is PROVED) establishes
  sums.abs_sum

Its premises are:
  sums.sum_ax
  sums.sum_ax
  sigmaprops.sigma_abs
  absolutes.abs_ax0

sums.sum_ax
  has already been justified

sums.sum_ax
  has already been justified

sigmaprops.sigma_abs
  is the conclusion of the proof
    sigmaprops.sa_proof
```

```
Proof sigmaprops.sa_proof (which is PROVED) establishes
  sigmaprops.sigma_abs

Its premises are:
  natinduction.induction
  sigmaprops.sigma_abs_basis
  sigmaprops.sigma_abs_step

natinduction.induction
  is the conclusion of the proof
    natinduction.induction_proof

Proof natinduction.induction_proof (which is PROVED) establishes
  natinduction.induction

Its premises are:
  natinduction.induction_m
  natprops.natpos

natinduction.induction_m
  has already been justified

natprops.natpos
  has already been justified

sigmaprops.sigma_abs_basis
  is the conclusion of the proof
    sigmaprops.sa_basis_proof

Proof sigmaprops.sa_basis_proof (which is PROVED) establishes
  sigmaprops.sigma_abs_basis

Its premises are:
  sigmaprops.sigma_ax
  sigmaprops.sigma_ax
  absolutes.abs_ax0

sigmaprops.sigma_ax
  has already been justified

sigmaprops.sigma_ax
  has already been justified
```

```
absolutes.abs_ax0
  has already been justified

sigmaprops.sigma_abs_step
  is the conclusion of the proof
    sigmaprops.sa_step_proof

Proof sigmaprops.sa_step_proof (which is PROVED) establishes
  sigmaprops.sigma_abs_step

Its premises are:
  sigmaprops.sigma_ax
  sigmaprops.sigma_ax
  absolutes.abs_ax2
  natprops.natpos
  natprops.pred_lemma

sigmaprops.sigma_ax
  has already been justified

sigmaprops.sigma_ax
  has already been justified

absolutes.abs_ax2
  is the conclusion of the proof
    absolutes.abs_proof2

Proof absolutes.abs_proof2 (which is PROVED) establishes
  absolutes.abs_ax2

Its premises are:
  absolutes.abs_ax
  absolutes.abs_ax
  absolutes.abs_ax

absolutes.abs_ax
  has already been justified

absolutes.abs_ax
  has already been justified

absolutes.abs_ax
  has already been justified
```

```
natprops.natpos
  has already been justified

natprops.pred_lemma
  has already been justified

absolutes.abs_ax0
  has already been justified

arithmetics.abs_div2
  is the conclusion of the proof
    arithmetics.abs_div2_proof

Proof arithmetics.abs_div2_proof (which is PROVED) establishes
  arithmetics.abs_div2

Its premises are:
  absolutes.abs_div
  absolutes.pos_abs

absolutes.abs_div
  is an axiom

absolutes.pos_abs
  is the conclusion of the proof
    absolutes.pos_abs_proof

Proof absolutes.pos_abs_proof (which is PROVED) establishes
  absolutes.pos_abs

Its premises are:
  absolutes.abs_ax

absolutes.abs_ax
  has already been justified

arithmetics.div_mon2
  is the conclusion of the proof
    arithmetics.div_mon2_proof

Proof arithmetics.div_mon2_proof (which is PROVED) establishes
  arithmetics.div_mon2

Its premises are:
```

```
  arithmetics.div_mon

arithmetics.div_mon
  has already been justified

absolutes.abs_ax0
  has already been justified

algorithm.C3
  is an axiom

The proof chain is complete

The axioms and assumptions at the base are:
  absolutes.abs_ax
  absolutes.abs_div
  algorithm.Alg1
  algorithm.Alg2
  algorithm.Alg3
  algorithm.C0_a
  algorithm.C0_c
  algorithm.C3
  algorithm.S2_ax
  arithmetics.mult_ax
  arithmetics.mult_mon
  arithmetics.quotient_ax
  arithmetics.quotient_ax1
  arithmetics.quotient_ax2
  natinduction.induction_m
  natprops.diff_ax
  natprops.natpos
  natprops.pred_ax
  sigmaprops.bounded_ax
  sigmaprops.sigma_ax
  sums.mean_ax
  sums.sum_ax
```

C.2 Clock Synchronization Condition S1

An extract from the proof chain for Theorem_1 in the specification is given below. The full proof chain listing contains over 3100 lines and enumerates

158 proofs and 48 axioms. As discussed in the text, the proof chain is apparently circular. The circularity is an artifact of the inductive nature of the proof.

```
Proof chain for formula Theorem_1 in module algorithm

algorithm.Theorem_1
  is the conclusion of the proof
    main.Theorem_1_proof

Proof main.Theorem_1_proof (which is PROVED) establishes
  algorithm.Theorem_1

Its premises are:
  main.basis
  main.ind_step
  natinduction.mod_induction
  clockprops.S1A_lemma
        .
        .
        .
        .
********* approximately 3000 lines omitted *********
        .
        .
        .
        .
The proof chain is complete

The axioms and assumptions at the base are:
  absolutes.abs_ax
  absolutes.abs_div
  algorithm.A0
  algorithm.A2
  algorithm.A2_aux
  algorithm.Alg1
  algorithm.Alg2
  algorithm.Alg3
  algorithm.C0_a
  algorithm.C0_b
  algorithm.C0_c
  algorithm.C2
  algorithm.C3
  algorithm.C4
```

```
algorithm.C5
algorithm.C6
algorithm.S1Adef
algorithm.S1Cdef
algorithm.S2_ax
arithmetics.half_ax
arithmetics.mult1
arithmetics.mult_ax
arithmetics.mult_mon
arithmetics.quotient_ax
arithmetics.quotient_ax1
arithmetics.quotient_ax2
clocks.A1
clocks.clockdef
clocks.gc_ax
clocks.rho_pos
clocks.zero_correction
functionprops.extensionality
natinduction.induction2
natinduction.induction_m
natprops.diff_ax
natprops.natpos
natprops.pred_ax
sigmaprops.bounded_ax
sigmaprops.revsigma_ax
sigmaprops.sigma_ax
sums.mean_ax
sums.sum_ax
time.C1
time.Rdef
time.Sdef
time.T_sup_ax
time.posR
time.posS

The proof chain is circular.  The directly circluar formulas are:

algorithm.Theorem_1
```

Appendix D

Plain EHDM Specification Transcripts

This appendix reproduces our specifications and proofs for the Interactive Convergence Clock Synchronization Algorithm exactly as processed by the EHDM system.

Module	Page
Absolutes	164
Algorithm	191
Arithmetics	166
Clockprops	194
Clocks	189
Functionprops	172
Juggle	212
Lemma1	197
Lemma2	198
Lemma3	200
Lemma4	201
Lemma5	203
Lemma6	205
Main	217
Natinduction	173
Natprops	170
Sigmaprops	180
Summations	207
Sums	175
Time	187

Table D.1: Page References to raw EHDM Specification Modules

```
absolutes: MODULE

EXPORTING abs

THEORY

  a, b, w, x, y, z: VAR number

  abs: function[number -> number]

  abs_ax: AXIOM abs(a) = IF a < 0 THEN -a ELSE a END IF

  abs_times: AXIOM abs(a*b) = abs(a) * abs(b)

  abs_div: AXIOM b /= 0 IMPLIES abs(a / b) = abs(a) / abs(b)

  abs_ax0: LEMMA 0 = abs(0)

  abs_ax1: LEMMA 0 <= abs(x)

  abs_ax2: LEMMA abs(x + y) <= abs(x) + abs(y)

  abs_ax2b: LEMMA abs(x + y + z) <= abs(x) + abs(y) + abs(z)

  abs_ax2c: LEMMA
    abs(w + x + y + z) <= abs(w) + abs(x) + abs(y) + abs(z)

  abs_ax3: LEMMA abs(-x) = abs(x)

  abs_ax4: LEMMA abs(x - y) = abs(y - x)

  abs_ax5: LEMMA
    0 <= x AND x <= z AND 0 <= y AND y <= z IMPLIES abs(x - y) <= z

  abs_ax6: LEMMA abs(x) <= y IMPLIES -y <= x AND x <= y

  abs_ax7: LEMMA abs(x) = abs(abs(x))

  abs_ax8: LEMMA abs(x - y) <= abs(x) + abs(y)

  pos_abs: LEMMA 0 <= x IMPLIES abs(x) = x

PROOF
```

```
  abs_proof0: PROVE abs_ax0 FROM abs_ax {a <- 0}

  abs_proof1: PROVE abs_ax1 FROM abs_ax {a <- x}

  abs_proof2: PROVE abs_ax2 FROM
    abs_ax {a <- x + y}, abs_ax {a <- x}, abs_ax {a <- y}

  abs_proof2b: PROVE abs_ax2b FROM
    abs_ax2 {y <- y + z}, abs_ax2 {x <- y, y <- z}

  abs_proof2c: PROVE abs_ax2c FROM
    abs_ax2 {x <- w, y <- x + y + z}, abs_ax2b

  abs_proof3: PROVE abs_ax3 FROM abs_ax {a <- x}, abs_ax {a <- -x}

  abs_proof4: PROVE abs_ax4 FROM
    abs_ax {a <- x - y}, abs_ax {a <- y - x}

  abs_proof5: PROVE abs_ax5 FROM abs_ax {a <- x - y}

  abs_proof6: PROVE abs_ax6 FROM abs_ax {a <- x}

  abs_proof7: PROVE abs_ax7 FROM abs_ax1, abs_ax {a <- abs(x)}

  abs_proof8: PROVE abs_ax8 FROM
    abs_ax {a <- x - y}, abs_ax {a <- x}, abs_ax {a <- y}

  pos_abs_proof: PROVE pos_abs FROM abs_ax {a <- x}

END absolutes
```

```
arithmetics: MODULE

USING absolutes

EXPORTING mult, half WITH absolutes

THEORY

  a, b, c, u, v, w, x, y, z: VAR number

  mult: function[number, number -> number]

  half: function[number -> number]

  (* ------------------------------------------------- *)

  quotient_ax: AXIOM y /= 0 IMPLIES x / y = x * (1 / y)

  quotient_ax1: AXIOM x /= 0 IMPLIES x / x = 1

  quotient_ax2: AXIOM z > 0 IMPLIES 1 / z > 0

  (* ------------------------------------------------- *)

  div_times: LEMMA y /= 0 IMPLIES (x / y) * z = (x * z) / y

  div_distr: LEMMA z /= 0 IMPLIES x / z + y / z = (x + y) / z

  abs_div2: LEMMA y > 0 IMPLIES abs(x / y) = abs(x) / y

  div_mon: LEMMA x < y AND z > 0 IMPLIES x / z < y / z

  div_mon2: LEMMA x <= y AND z > 0 IMPLIES x / z <= y / z

  div_prod: LEMMA y > 0 AND a < x * y IMPLIES a / y < x

  div_prod2: LEMMA y > 0 AND a <= x * y IMPLIES a / y <= x

  cancellation: LEMMA y /= 0 IMPLIES (y * x) / y = x

  (* ------------------------------------------------- *)

  mult_ax: AXIOM mult(x, y) = x * y

  mult1: AXIOM x >= 0 AND y >= 0 IMPLIES mult(x, y) >= 0
```

```
  mult_mon: AXIOM x < y AND z > 0 IMPLIES mult(x, z) < mult(y, z)

  (* -------------------------------------------------- *)

  mult_mon2: LEMMA x <= y AND z > 0 IMPLIES mult(x, z) <= mult(y, z)

  cancellation_mult: LEMMA y /= 0 IMPLIES mult(x, y) / y = x

  mult0: LEMMA y = 0 IMPLIES mult(x, y) = 0

  mult_div: LEMMA y /= 0 IMPLIES mult(x / y, y) = x

  (* -------------------------------------------------- *)

  half_ax: AXIOM half(x) = x / 2

  (* -------------------------------------------------- *)

  times_half: LEMMA 2 * half(x) = x

  half2: LEMMA half(x) + half(x) = x

  half3: LEMMA 2 * mult(half(x), y) = mult(x, y)

  mult2: LEMMA 2 * (mult(x, y)) = mult((2 * x), y)

  mult3: LEMMA mult(x, y + z) = mult(x, y) + mult(x, z)

  mult4: LEMMA 0 <= x AND y <= z IMPLIES mult(x, y) <= mult(x, z)

  rearrange: LEMMA
    abs(x - y)
      <= abs(x - (u + v)) + abs(y - (w + z)) + abs(u + v - (w + z))

  rearrange_alt: LEMMA
    abs(x - y) <= abs(x - (u + v)) + abs(u - w) + abs(y - (w + v))

PROOF

  div_times_proof: PROVE div_times FROM
    quotient_ax, quotient_ax {x <- x * z}

  div_distr_proof: PROVE div_distr FROM
    quotient_ax {y <- z},
    quotient_ax {x <- y, y <- z},
    quotient_ax {x <- x + y, y <- z}
```

```
abs_div2_proof: PROVE abs_div2 FROM
  abs_div {a <- x, b <- y}, pos_abs {x <- y}

quotient_mult: LEMMA y /= 0 IMPLIES x / y = mult(x, 1 / y)

quotient_mult_proof: PROVE quotient_mult FROM
  quotient_ax, mult_ax {y <- 1 / y}

div_mon_proof: PROVE div_mon FROM
  mult_mon {z <- 1 / z},
  quotient_mult {y <- z},
  quotient_mult {x <- y, y <- z},
  quotient_ax2

div_mon2_proof: PROVE div_mon2 FROM div_mon

div_mult: LEMMA y > 0 AND a < mult(x, y) IMPLIES a / y < x

div_mult_proof: PROVE div_mult FROM
  div_mon {z <- y, x <- a, y <- mult(x, y)}, cancellation_mult

div_mult2: LEMMA y > 0 AND a <= mult(x, y) IMPLIES a / y <= x

div_mult2_proof: PROVE div_mult2 FROM
  div_mon {z <- y, x <- a, y <- mult(x, y)}, cancellation_mult

div_prod_proof: PROVE div_prod FROM div_mult, mult_ax

div_prod2_proof: PROVE div_prod2 FROM div_mult2, mult_ax

cancellation_proof: PROVE cancellation FROM
  div_times {x <- y, z <- x}, quotient_ax1 {x <- y}

mult_mon2_proof: PROVE mult_mon2 FROM mult_mon

cancellation_mult_proof: PROVE cancellation_mult FROM
  cancellation, mult_ax

mult0_proof: PROVE mult0 FROM mult_ax {y <- 0}

mult_div_proof: PROVE mult_div FROM
  mult_ax {x <- x / y}, div_times {z <- y}, cancellation

times_half_proof: PROVE times_half FROM
  half_ax, div_times {y <- 2, z <- 2}, cancellation {y <- 2}
```

```
  half2_proof: PROVE half2 FROM times_half

  half3_proof: PROVE half3 FROM mult2 {x <- half(x)}, times_half

  mult2_proof: PROVE mult2 FROM mult_ax, mult_ax {x <- 2 * x}

  mult3_proof: PROVE mult3 FROM
    mult_ax, mult_ax {y <- z}, mult_ax {y <- y + z}

  mult4_proof: PROVE mult4 FROM mult3 {z <- z - y}, mult1 {y <- z - y}

  rearrange1: LEMMA
    x - y = (x - (u + v)) + (w + z - y) + (u + v - (w + z))

  rearrange1_proof: PROVE rearrange1

  rearrange2: LEMMA
    abs((x - (u + v)) + (w + z - y) + (u + v - (w + z)))
      <= abs(x - (u + v)) + abs(y - (w + z)) + abs(u + v - (w + z))

  rearrange2_proof: PROVE rearrange2 FROM
    abs_ax2b {x <- x - (u + v), y <- u + v - (w + z), z <- w + z - y},
    abs_ax3 {x <- w + z - y}

  rearrange_proof: PROVE rearrange FROM rearrange1, rearrange2

  rearrange_alt_proof: PROVE rearrange_alt FROM rearrange {z <- v}

END arithmetics
```

```
natprops: MODULE

EXPORTING pred, diff

THEORY

  i, m, n: VAR nat

  pred: function[nat -> nat]

  natpos: AXIOM n >= 0

  pred_ax: AXIOM n /= 0 IMPLIES pred(n) = n - 1

  diff: function[nat, nat -> nat]

  diff_ax: AXIOM n >= m IMPLIES diff(n, m) = n - m

  pred_lemma: LEMMA pred(n + 1) = n

  diff_zero: LEMMA n > m IMPLIES diff(n, m) > 0

  pred_diff: LEMMA n > m IMPLIES pred(diff(n, m)) = diff(n, m + 1)

  diff1: LEMMA n >= m IMPLIES diff(n + 1, m + 1) = diff(n, m)

  diff_diff: LEMMA
    n >= m AND n >= i AND m >= i
      IMPLIES diff(diff(n, i), diff(m, i)) = diff(n, m)

  diff_plus: LEMMA n >= m IMPLIES m + diff(n, m) = n

  diff_ineq: LEMMA
    n >= m AND n >= i AND m >= i IMPLIES diff(n, i) >= diff(m, i)

PROOF

  pred_lemma_proof: PROVE pred_lemma FROM pred_ax {n <- n + 1}, natpos

  diff_zero_proof: PROVE diff_zero FROM diff_ax

  pred_diff_proof: PROVE pred_diff FROM
    pred_ax {n <- diff(n, m)}, diff_ax, diff_ax {m <- m + 1}

  diff1_proof: PROVE diff1 FROM
```

```
    diff_ax, diff_ax {n <- n + 1, m <- m + 1}

  diff_diff_proof: PROVE diff_diff FROM
    diff_ax,
    diff_ax {m <- 1},
    diff_ax {n <- m, m <- 1},
    diff_ax {n <- diff(n, 1), m <- diff(m, 1)}

  diff_plus_proof: PROVE diff_plus FROM diff_ax

  diff_ineq_proof: PROVE diff_ineq FROM
    diff_ax {m <- 1}, diff_ax {n <- m, m <- 1}

END natprops
```

```
functionprops: MODULE

THEORY

  F, G: VAR function[nat -> number]

  x: VAR nat

  extensionality: AXIOM (FORALL x : F(x) = G(x)) IMPLIES F = G

END functionprops
```

```
natinduction: MODULE

USING natprops

THEORY

  i, i0, i1, i2, i3, j, m, n: VAR nat

  prop, A, B: VAR function[nat -> bool]

  prop2: VAR function[nat, nat -> bool]

  induction_m: AXIOM
    (prop(m) AND (FORALL i : i >= m AND prop(i) IMPLIES prop(i + 1)))
      IMPLIES (FORALL n : n >= m IMPLIES prop(n))

  induction2: AXIOM
    (FORALL i0 : prop2(i0, 0))
        AND (FORALL j :
            (FORALL i1 : prop2(i1, j))
              IMPLIES (FORALL i2 : prop2(i2, j + 1)))
      IMPLIES (FORALL i3, n : prop2(i3, n))

  mod_induction_m: LEMMA
    (FORALL j : j >= m AND A(j + 1) IMPLIES A(j))
        AND ((A(m) IMPLIES B(m))
              AND (FORALL i :
                  i >= m AND A(i + 1) AND B(i) IMPLIES B(i + 1)))
      IMPLIES (FORALL n : n >= m AND A(n) IMPLIES B(n))

  induction: LEMMA
    (prop(0) AND (FORALL i : prop(i) IMPLIES prop(i + 1)))
      IMPLIES (FORALL n : prop(n))

  mod_induction: LEMMA
    (FORALL j : A(j + 1) IMPLIES A(j))
        AND ((A(0) IMPLIES B(0))
              AND (FORALL i : A(i + 1) AND B(i) IMPLIES B(i + 1)))
      IMPLIES (FORALL n : A(n) IMPLIES B(n))

  induction1: LEMMA
    (prop(1) AND (FORALL i : i >= 1 AND prop(i) IMPLIES prop(i + 1)))
      IMPLIES (FORALL n : n >= 1 IMPLIES prop(n))

  mod_induction1: LEMMA
```

```
    (FORALL j : j >= 1 AND A(j + 1) IMPLIES A(j))
        AND ((A(1) IMPLIES B(1))
             AND (FORALL i :
                i >= 1 AND A(i + 1) AND B(i) IMPLIES B(i + 1)))
      IMPLIES (FORALL n : n >= 1 AND A(n) IMPLIES B(n))

PROOF

  mod_m_proof: PROVE mod_induction_m {i <- i@p1, j <- i} FROM
    induction_m {prop <- (LAMBDA i -> bool : A(i) IMPLIES B(i))}

  induction_proof: PROVE induction {i <- i@p1} FROM
    induction_m {m <- 0}, natpos

  mod_induction_proof: PROVE mod_induction {i <- i@p1, j <- j@p1} FROM
    mod_induction_m {m <- 0}, natpos

  induction1_proof: PROVE induction1 {i <- i@p1} FROM
    induction_m {m <- 1}

  mod_induction1_proof: PROVE mod_induction1 {i <- i@p1, j <- j@p1} FROM
    mod_induction_m {m <- 1}

END natinduction
```

```
sums: MODULE

USING arithmetics, natprops, sigmaprops

EXPORTING sum, mean

THEORY

  i, j, k, n, pp, qq, rr: VAR nat

  x, y, z: VAR number

  F, G: VAR function[nat -> number]

  sum: function[nat, nat, function[nat -> number] -> number]

  mean: function[nat, nat, function[nat -> number] -> number]

  sum_ax: AXIOM
    sum(i, j, F)
      = IF i <= j + 1 THEN sigma(i, diff(j + 1, i), F) ELSE 0 END IF

  mean_ax: AXIOM
    mean(i, j, F)
      = IF i <= j THEN sum(i, j, F) / (j + 1 - i) ELSE 0 END IF

  mean_lemma: LEMMA
    mean(i, j, F)
      = IF i <= j
        THEN sigma(i, diff(j + 1, i), F) / (j + 1 - i)
        ELSE 0
        END IF

  split_sum: LEMMA
    i <= j + 1 AND i <= k + 1 AND k <= j
      IMPLIES sum(i, j, F) = sum(i, k, F) + sum(k + 1, j, F)

  split_mean: LEMMA
    i <= j AND i <= k + 1 AND k <= j
      IMPLIES mean(i, j, F)
        = (sum(i, k, F) + sum(k + 1, j, F)) / (j - i + 1)

  sum_bound: LEMMA
    i <= j + 1 AND (FORALL pp : i <= pp AND pp <= j IMPLIES F(pp) < x)
      IMPLIES sum(i, j, F) <= x * (j - i + 1)
```

```
  mean_bound: LEMMA
    i <= j AND (FORALL pp : i <= pp AND pp <= j IMPLIES F(pp) < x)
      IMPLIES mean(i, j, F) < x

  mean_const: LEMMA
    i <= j IMPLIES x = mean(i, j, (LAMBDA qq -> number : x))

  mean_mult: LEMMA
    mean(i, j, F) * x = mean(i, j, (LAMBDA qq -> number : F(qq) * x))

  mean_sum: LEMMA
    mean(i, j, F) + mean(i, j, G)
      = mean(i, j, (LAMBDA qq -> number : F(qq) + G(qq)))

  mean_diff: LEMMA
    mean(i, j, F) - mean(i, j, G)
      = mean(i, j, (LAMBDA qq -> number : F(qq) - G(qq)))

  abs_mean: LEMMA
    abs(mean(i, j, F)) <= mean(i, j, (LAMBDA qq -> number : abs(F(qq))))

  rearrange_sum: LEMMA
    i <= j IMPLIES x + mean(i, j, F) - (y + mean(i, j, G))
        = mean(i, j, (LAMBDA qq -> number : x + F(qq) - (y + G(qq))))

PROOF

  mean_lemma_proof: PROVE mean_lemma FROM mean_ax, sum_ax

  (* ---------------------------------------------------------------- *)

  split_sum_proof: PROVE split_sum FROM
    sum_ax,
    sum_ax {j <- k},
    sum_ax {i <- k + 1},
    split_sigma {n <- diff(j + 1, i), m <- diff(k + 1, i), i <- i},
    diff_diff {n <- j + 1, m <- k + 1},
    diff_plus {n <- k + 1, m <- i},
    diff_ineq {n <- j + 1, m <- k + 1}

  split_mean_proof: PROVE split_mean FROM split_sum, mean_ax

  (* ---------------------------------------------------------------- *)

  sigma_bound2: LEMMA
    n > 0 AND (FORALL k : i <= k AND k <= i + pred(n) IMPLIES F(k) < x)
```

```
    IMPLIES sigma(i, n, F) < mult(x, n)

sigma_bound2_proof: PROVE sigma_bound2 {k <- k@p1} FROM
  sigma_bound, mult_ax {y <- n}

sum_bound_mod: LEMMA
  i <= j AND (FORALL pp : i <= pp AND pp <= j IMPLIES F(pp) < x)
    IMPLIES sum(i, j, F) < mult(x, (j + 1 - i))

sum_bound_mod_proof: PROVE sum_bound_mod {pp <- k@p2} FROM
  sum_ax,
  sigma_bound2 {n <- diff(j + 1, i), i <- i},
  pred_diff {n <- j + 1, m <- i},
  diff_ax {n <- j + 1, m <- i},
  diff_ax {n <- j + 1, m <- i + 1}

sum_bound1: LEMMA
  i <= j AND (FORALL pp : i <= pp AND pp <= j IMPLIES F(pp) < x)
    IMPLIES sum(i, j, F) < x * (j - i + 1)

sum_bound1_proof: PROVE sum_bound1 {pp <- pp@p1} FROM
  sum_bound_mod, mult_ax {y <- j + 1 - i}

sum_bound0: LEMMA
  i = j + 1 AND (FORALL pp : i <= pp AND pp <= j IMPLIES F(pp) < x)
    IMPLIES sum(i, j, F) <= mult(x, (j + 1 - i))

sum_bound0_proof: PROVE sum_bound0 FROM
  sum_ax {i <- j + 1},
  diff_ax {n <- j + 1, m <- j + 1},
  sigma_ax {i <- j + 1, n <- 0},
  mult0 {y <- j + 1 - i}

sum_bound2: LEMMA
  i <= j + 1 AND (FORALL pp : i <= pp AND pp <= j IMPLIES F(pp) < x)
    IMPLIES sum(i, j, F) <= mult(x, (j + 1 - i))

sum_bound2_proof: PROVE sum_bound2 {pp <- pp@p1} FROM
  sum_bound_mod, sum_bound0

sum_bound_proof: PROVE sum_bound {pp <- pp@p1} FROM
  sum_bound2, mult_ax {y <- j + 1 - i}

(* ---------------------------------------------------------------- *)

mean_bound_proof: PROVE mean_bound {pp <- pp@p1} FROM
  sum_bound1, mean_ax, div_prod {a <- sum(i, j, F), y <- j - i + 1}
```

```
(* -------------------------------------------------------------- *)

mean_const_proof: PROVE mean_const FROM
  mean_lemma {F <- (LAMBDA qq -> number : x)},
  sigma_const {n <- diff(j + 1, i), i <- i},
  diff_ax {n <- j + 1, m <- i},
  cancellation {y <- j + 1 - i}

(* -------------------------------------------------------------- *)

sum_mult: LEMMA
  sum(i, j, F) * x = sum(i, j, (LAMBDA qq -> number : F(qq) * x))

sum_mult_proof: PROVE sum_mult FROM
  sum_ax,
  sum_ax {F <- (LAMBDA qq -> number : F(qq) * x)},
  mod_sigma_mult {i <- i, n <- diff(j + 1, i)}

mean_mult_proof: PROVE mean_mult FROM
  mean_ax,
  mean_ax {F <- (LAMBDA qq -> number : F(qq) * x)},
  sum_mult,
  div_times {x <- sum(i, j, F@p3), y <- j + 1 - i, z <- x}

(* -------------------------------------------------------------- *)

mean_sum_proof: PROVE mean_sum FROM
  mean_lemma {F <- (LAMBDA qq -> number : F(qq) + G(qq))},
  mean_lemma,
  mean_lemma {F <- G},
  sigma_sum {n <- diff(j + 1, i), i <- i},
  div_distr
    {x <- sigma(i, diff(j + 1, i), F),
     y <- sigma(i, diff(j + 1, i), G),
     z <- j + 1 - i}

(* -------------------------------------------------------------- *)

mean_diff_proof: PROVE mean_diff FROM
  mean_mult {F <- G, x <- -1},
  mean_sum {G <- (LAMBDA qq -> number : G(qq) * -1)}

(* -------------------------------------------------------------- *)

abs_sum: LEMMA
  abs(sum(i, j, F)) <= sum(i, j, (LAMBDA qq -> number : abs(F(qq))))
```

```
  abs_sum_proof: PROVE abs_sum FROM
    sum_ax,
    sum_ax {F <- (LAMBDA qq -> number : abs(F(qq)))},
    sigma_abs {n <- diff(j + 1, i), i <- i},
    abs_ax0

  abs_mean_proof: PROVE abs_mean FROM
    mean_ax,
    mean_ax {F <- (LAMBDA qq -> number : abs(F(qq)))},
    abs_sum,
    abs_div2 {x <- sum(i, j, F), y <- j + 1 - i},
    div_mon2
      {x <- abs(sum(i, j, F)),
       y <- sum(i, j, F@p2),
       z <- j + 1 - i},
    abs_ax0

  (* -------------------------------------------------------------- *)

  rearrange_sub: LEMMA
    i <= j IMPLIES x + mean(i, j, F)
        = mean(i, j, (LAMBDA qq -> number : x + F(qq)))

  rearrange_sub_proof: PROVE rearrange_sub FROM
    mean_const, mean_sum {G <- (LAMBDA qq -> number : x)}

  rearrange_sum_proof: PROVE rearrange_sum FROM
    rearrange_sub,
    rearrange_sub {x <- y, F <- G},
    mean_diff
      {F <- (LAMBDA pp -> number : x + F@c(pp)),
       G <- (LAMBDA pp -> number : y + G@c(pp))}

END sums
```

```
sigmaprops: MODULE

USING arithmetics, natprops, functionprops, natinduction

EXPORTING sigma

THEORY

  i, i1, i2, j, k, l: VAR nat

  F, G: VAR function[nat -> number]

  n, m, mm, nn, qq: VAR nat

  x, y: VAR number

  sigma: function[nat, nat, function[nat -> number] -> number]

  sigma_ax: AXIOM
    sigma(i, n, F)
      = IF n = 0
        THEN 0
        ELSE F(i + pred(n)) + sigma(i, pred(n), F)
        END IF

  sigma_const: LEMMA sigma(i, n, (LAMBDA qq -> number : x)) = n * x

  sigma_mult: LEMMA
    sigma(i, n, (LAMBDA qq -> number : x * F(qq))) = x * sigma(i, n, F)

  mod_sigma_mult: LEMMA
    sigma(i, n, (LAMBDA qq -> number : F(qq) * x)) = sigma(i, n, F) * x

  sigma_sum: LEMMA
    sigma(i, n, F) + sigma(i, n, G)
      = sigma(i, n, (LAMBDA qq -> number : F(qq) + G(qq)))

  split_sigma: LEMMA
    n >= m IMPLIES sigma(i, n, F)
        = sigma(i, m, F) + sigma(i + m, diff(n, m), F)

  sigma_abs: LEMMA
    abs(sigma(i, n, F))
      <= sigma(i, n, (LAMBDA qq -> number : abs(F(qq))))
```

```
  sigma_bound: LEMMA
    n > 0 AND (FORALL k : i <= k AND k <= i + pred(n) IMPLIES F(k) < x)
      IMPLIES sigma(i, n, F) < n * x

 (* ====================================================== *)

  bounded: function[nat, nat, function[nat -> number], number -> bool]

  bounded_ax: AXIOM
    n > 0 IMPLIES (bounded(i, n, F, x)
          = (FORALL k : i <= k AND k <= i + pred(n) IMPLIES F(k) < x))

  revsigma: function[nat, nat, function[nat -> number] -> number]

  revsigma_ax: AXIOM
    revsigma(i, n, F)
      = IF n = 0 THEN 0 ELSE F(i) + revsigma(i + 1, pred(n), F) END IF

  sigma_rev: LEMMA sigma(i, n, F) = revsigma(i, n, F)

PROOF

  sigma_const_basis: LEMMA sigma(i, 0, (LAMBDA qq -> number : x)) = 0

  sc_basis_proof: PROVE sigma_const_basis FROM
    sigma_ax {n <- 0, F <- (LAMBDA qq -> number : x)}

  sigma_const_step: LEMMA
    sigma(i, n, (LAMBDA qq -> number : x)) = n * x
      IMPLIES sigma(i, n + 1, (LAMBDA qq -> number : x)) = (n + 1) * x

  sc_step_proof: PROVE sigma_const_step FROM
    sigma_ax {n <- n + 1, F <- (LAMBDA qq -> number : x)}, pred_lemma

  sc_proof: PROVE sigma_const FROM
    induction
      {prop <- (LAMBDA nn -> bool :
           sigma(i, nn, (LAMBDA qq -> number : x)) = nn * x)},
    sigma_const_basis,
    sigma_const_step {n <- i@p1}

  (* ------------------------------------------------------------- *)

  sigma_mult_basis: LEMMA
    sigma(i, 0, (LAMBDA qq -> number : x * F(qq))) = x * sigma(i, 0, F)

  sm_basis_proof: PROVE sigma_mult_basis FROM
```

```
  sigma_ax {n <- 0},
  sigma_ax {n <- 0, F <- (LAMBDA qq -> number : x * F(qq))}

sigma_mult_step: LEMMA
  sigma(i, n, (LAMBDA qq -> number : x * F(qq))) = x * sigma(i, n, F)
    IMPLIES sigma(i, n + 1, (LAMBDA qq -> number : x * F(qq)))
      = x * sigma(i, n + 1, F)

sm_step_proof: PROVE sigma_mult_step FROM
  sigma_ax {n <- n + 1, F <- (LAMBDA qq -> number : x * F(qq))},
  sigma_ax {n <- n + 1},
  pred_lemma

sm_proof: PROVE sigma_mult FROM
  induction
    {prop <- (LAMBDA nn -> bool :
         sigma(i, nn, (LAMBDA qq -> number : x * F(qq)))
           = x * sigma(i, nn, F))},
  sigma_mult_basis,
  sigma_mult_step {n <- i@p1}

(* ---------------------------------------------------------------- *)

mod_sigma_mult_proof: PROVE mod_sigma_mult FROM
  sigma_mult,
  extensionality
    {F <- (LAMBDA qq -> number : x * F(qq)),
     G <- (LAMBDA qq -> number : F(qq) * x)}

(* ---------------------------------------------------------------- *)

sigma_sum_basis: LEMMA
  sigma(i, 0, F) + sigma(i, 0, G)
    = sigma(i, 0, (LAMBDA qq -> number : F(qq) + G(qq)))

ss_basis_proof: PROVE sigma_sum_basis FROM
  sigma_ax {n <- 0, F <- (LAMBDA qq -> number : F(qq) + G(qq))},
  sigma_ax {n <- 0, F <- (LAMBDA qq -> number : G(qq))},
  sigma_ax {n <- 0}

sigma_sum_step: LEMMA
  sigma(i, n, F) + sigma(i, n, G)
      = sigma(i, n, (LAMBDA qq -> number : F(qq) + G(qq)))
    IMPLIES sigma(i, n + 1, F) + sigma(i, n + 1, G)
      = sigma(i, n + 1, (LAMBDA qq -> number : F(qq) + G(qq)))

ss_step_proof: PROVE sigma_sum_step FROM
```

```
    sigma_ax {n <- n + 1, F <- (LAMBDA qq -> number : F(qq) + G(qq))},
    sigma_ax {n <- n + 1, F <- (LAMBDA qq -> number : G(qq))},
    sigma_ax {n <- n + 1},
    pred_lemma

  ss_proof: PROVE sigma_sum FROM
    induction
      {prop <- (LAMBDA nn -> bool :
           sigma(i, nn, F) + sigma(i, nn, G)
             = sigma(i, nn, (LAMBDA qq -> number : F(qq) + G(qq))))},
    sigma_sum_basis,
    sigma_sum_step {n <- i@p1}

  (* ---------------------------------------------------------------- *)

  split_sigma_basis: LEMMA
    sigma(i, n, F) = sigma(i, 0, F) + sigma(i, diff(n, 0), F)

  split_basis_proof: PROVE split_sigma_basis FROM
    sigma_ax, sigma_ax {n <- 0}, diff_ax {m <- 0}, natpos

  split_sigma_step: LEMMA
    (n >= m IMPLIES sigma(i, n, F)
              = sigma(i, m, F) + sigma(i + m, diff(n, m), F))
      IMPLIES (n >= m + 1
            IMPLIES sigma(i, n, F)
              = sigma(i, m + 1, F) + sigma(i + m + 1, diff(n, m + 1), F))

  split_step_proof: PROVE split_sigma_step FROM
    sigma_ax {n <- m + 1},
    sigma_rev {i <- i + m + 1, n <- diff(n, m + 1)},
    revsigma_ax {i <- i + m, n <- diff(n, m)},
    sigma_rev {i <- i + m, n <- diff(n, m)},
    pred_lemma {n <- m},
    pred_diff,
    diff_zero,
    natpos {n <- m}

  split_proof: PROVE split_sigma FROM
    induction
      {n <- m,
       prop <- (LAMBDA nn -> bool :
            n >= nn
              IMPLIES sigma(i, n, F)
                = sigma(i, nn, F) + sigma(i + nn, diff(n, nn), F))},
    split_sigma_basis,
    split_sigma_step {m <- i@p1}
```

```
(* ------------------------------------------------------------------ *)

sigma_abs_basis: LEMMA
  abs(sigma(i, 0, F))
    <= sigma(i, 0, (LAMBDA qq -> number : abs(F(qq))))

sa_basis_proof: PROVE sigma_abs_basis FROM
  sigma_ax {n <- 0},
  sigma_ax {n <- 0, F <- (LAMBDA qq -> number : abs(F(qq)))},
  abs_ax0

sigma_abs_step: LEMMA
  abs(sigma(i, n, F))
      <= sigma(i, n, (LAMBDA qq -> number : abs(F(qq))))
    IMPLIES abs(sigma(i, n + 1, F))
      <= sigma(i, n + 1, (LAMBDA qq -> number : abs(F(qq))))

sa_step_proof: PROVE sigma_abs_step FROM
  sigma_ax {n <- n + 1},
  sigma_ax {n <- n + 1, F <- (LAMBDA qq -> number : abs(F(qq)))},
  abs_ax2 {x <- F(i + n), y <- sigma(i, n, F)},
  natpos,
  pred_lemma

sa_proof: PROVE sigma_abs FROM
  induction
    {prop <- (LAMBDA nn -> bool :
         abs(sigma(i, nn, F))
           <= sigma(i, nn, (LAMBDA qq -> number : abs(F(qq)))))},
  sigma_abs_basis,
  sigma_abs_step {n <- i@p1}

(* ------------------------------------------------------------------ *)

bounded_lemma: LEMMA
  n > 0 AND bounded(i, n + 1, F, x) IMPLIES bounded(i, n, F, x)

bounded_proof: PROVE bounded_lemma FROM
  bounded_ax {k <- k@p1},
  bounded_ax {n <- n + 1, k <- k@p1},
  pred_lemma,
  pred_ax

sigma_bound_basis: LEMMA
  bounded(i, 1, F, x) IMPLIES sigma(i, 1, F) < x
```

```
sb_basis_proof: PROVE sigma_bound_basis FROM
  bounded_ax {n <- 1, k <- i},
  sigma_ax {n <- 0},
  sigma_ax {n <- 1},
  pred_ax {n <- 1}

alt_sigma_bound_step: LEMMA
  n > 0 AND bounded(i, n + 1, F, x) AND sigma(i, n, F) < mult(n, x)
    IMPLIES sigma(i, n + 1, F) < x + mult(n, x)

alt_sb_step_proof: PROVE alt_sigma_bound_step FROM
  bounded_ax {n <- n + 1, k <- i + n},
  sigma_ax {n <- n + 1},
  pred_lemma,
  natpos

sigma_bound_step: LEMMA
  n > 0 AND bounded(i, n + 1, F, x) AND sigma(i, n, F) < n * x
    IMPLIES sigma(i, n + 1, F) < (n + 1) * x

sb_step_proof: PROVE sigma_bound_step FROM
  alt_sigma_bound_step, mult_ax {x <- n, y <- x}

sb: LEMMA n > 0 AND bounded(i, n, F, x) IMPLIES sigma(i, n, F) < n * x

sb_proof: PROVE sb FROM
  mod_induction1
    {A <- (LAMBDA nn -> bool : bounded(i, nn, F, x)),
     B <- (LAMBDA mm -> bool : sigma(i, mm, F) < mm * x)},
  bounded_lemma {n <- j@p1},
  sigma_bound_basis,
  sigma_bound_step {n <- i@p1}

sigma_bound_proof: PROVE sigma_bound {k <- k@p2} FROM sb, bounded_ax

(* ------------------------------------------------------ *)

sigma1: LEMMA sigma(i, n + 1, F) = F(i) + sigma(i + 1, n, F)

sigma1_basis: LEMMA sigma(i, 1, F) = F(i) + sigma(i + 1, 0, F)

s1b_proof: PROVE sigma1_basis FROM
  sigma_ax {n <- 0},
  sigma_ax {i <- i + 1, n <- 0},
  sigma_ax {n <- 1},
  pred_ax {n <- 1}
```

```
  sigma1_step: LEMMA
    sigma(i, n + 1, F) = F(i) + sigma(i + 1, n, F)
      IMPLIES sigma(i, n + 2, F) = F(i) + sigma(i + 1, n + 1, F)

  s1s_proof: PROVE sigma1_step FROM
    sigma_ax {i <- i + 1, n <- n + 1},
    sigma_ax {n <- n + 2},
    pred_lemma,
    pred_lemma {n <- n + 1},
    natpos

  sigma1_proof: PROVE sigma1 FROM
    induction
      {prop <- (LAMBDA nn -> bool :
           sigma(i, nn + 1, F) = F(i) + sigma(i + 1, nn, F))},
    sigma1_basis,
    sigma1_step {n <- i@p1}

  (* ------------------------------------------------------- *)

  sigma_rev_basis: LEMMA sigma(i, 0, F) = revsigma(i, 0, F)

  srb_proof: PROVE sigma_rev_basis FROM
    sigma_ax {n <- 0}, revsigma_ax {n <- 0}

  sigma_rev_step: LEMMA
    (FORALL i1 : sigma(i1, n, F) = revsigma(i1, n, F))
      IMPLIES (FORALL i2 : sigma(i2, n + 1, F) = revsigma(i2, n + 1, F))

  srp_proof: PROVE sigma_rev_step {i1 <- i2 + 1} FROM
    revsigma_ax {i <- i2, n <- n + 1},
    sigma1 {i <- i2},
    pred_lemma,
    natpos

  sigma_rev_proof: PROVE sigma_rev FROM
    induction2
      {i1 <- i1@p3,
       i3 <- i,
       prop2 <- (LAMBDA i, nn -> bool :
           sigma(i, nn, F) = revsigma(i, nn, F))},
    sigma_rev_basis {i <- i0@p1},
    sigma_rev_step {i2 <- i2@p1, n <- j@p1}

END sigmaprops
```

```
time: MODULE

USING arithmetics

EXPORTING clocktime, realtime, period, R, S, T_ZERO, T_sup, in_R_interval,
  in_S_interval WITH arithmetics

THEORY

  clocktime: TYPE IS number

  realtime: TYPE IS number

  period: TYPE IS nat

  R, S: clocktime(* Synchronizing periods *)

  posR: AXIOM 0 < R

  posS: AXIOM 0 < S

  C1: AXIOM R >= 3 * S

  SinR: LEMMA S < R

  i: VAR period

  T_sup: function[period -> clocktime]

  T_ZERO: clocktime

  T_sup_ax: AXIOM T_sup(i) = T_ZERO + i * R

  T_next: LEMMA T_sup(i+1) = T_sup(i) + R

  T, T1, T2, PI: VAR clocktime

  in_R_interval: function[clocktime, period -> boolean]

  Rdef: AXIOM in_R_interval(T, i)
      = (EXISTS PI : 0 <= PI AND PI <= R AND T = T_sup(i) + PI)

  Ti_in_R: LEMMA in_R_interval(T_sup(i), i)

  in_S_interval: function[clocktime, period -> boolean]
```

```
  Sdef: AXIOM in_S_interval(T, i)
      = (EXISTS PI : 0 <= PI AND PI <= S AND T = T_sup(i) + R - S + PI)

  inRS: LEMMA in_S_interval(T, i) IMPLIES in_R_interval(T, i)

  Ti_in_S: LEMMA in_S_interval(T_sup(i + 1), i)

  in_S_lemma: LEMMA
    in_S_interval(T1, i) AND in_S_interval(T2, i) IMPLIES abs(T1 - T2) <= S

PROOF

  SinR_proof: PROVE SinR FROM C1, posS, posR

  Ti_proof: PROVE Ti_in_R FROM Rdef {T <- T_sup(i), PI <- 0}, abs_ax0, posR

  inRS_proof: PROVE inRS FROM Sdef, Rdef {PI <- R - S + PI@p1}, SinR

  T_next_proof: prove T_next from T_sup_ax, T_sup_ax{i<-i+1}

  Ti_in_S_proof: PROVE Ti_in_S FROM Sdef{PI<-S, T<-
T_sup(i+1)}, posS, T_next

  in_S_proof: PROVE in_S_lemma FROM
    Sdef {T <- T1}, Sdef {T <- T2}, abs_ax5 {x <- PI@p1, y <- PI@p2, z <- S}

END time
```

```
clocks: MODULE

USING time

EXPORTING proc, clock, rho, Corr, adjusted, rt, nonfaulty
  WITH time

THEORY

  proc: TYPE IS nat

  p: VAR proc

  clock: function[proc, clocktime -> realtime]

  Corr: function[proc, period -> clocktime]

  zero_correction: AXIOM Corr(p, 0) = 0

  i: VAR period

  T, T0, T1, T2, TN: VAR clocktime

  adjusted: function[proc, period, clocktime -> clocktime] =
    (LAMBDA p, i, T -> clocktime : T + Corr(p, i))

  rt: function[proc, period, clocktime -> realtime]

  clockdef: AXIOM rt(p, i, T) = clock(p, adjusted(p, i, T))

  goodclock: function[proc, clocktime, clocktime -> bool]

  rho: number

  rho_pos: AXIOM half(rho) >= 0

  rho_small: AXIOM half(rho) < 1

  gc_ax: AXIOM
    goodclock(p, T0, TN)
      = (FORALL T1, T2 :
          T0 <= T1 AND T0 <= T2 AND T1 <= TN AND T2 <= TN
            IMPLIES abs(clock(p, T1) - clock(p, T2) - (T1 - T2))
              < mult(half(rho), abs(T1 - T2)))
```

```
  monotonicity: THEOREM
    goodclock(p, TO, TN)
        AND TO <= T1 AND TO <= T2 AND T1 <= TN AND T2 <= TN
      IMPLIES (T1 > T2 IMPLIES clock(p, T1) > clock(p, T2))

  nonfaulty: function[proc, period -> boolean]

  A1: AXIOM nonfaulty(p, i)
      = goodclock(p,
                  adjusted(p, 0, T_sup(0)),
                  adjusted(p, i, T_sup(i + 1)))

PROOF

  x, y: VAR number

  diminish: LEMMA x > 0 IMPLIES mult(half(rho), x) <= x

  diminish_proof: PROVE diminish FROM
    mult_mon {x <- half(rho), y <- 1, z <- x},
    rho_small,
    mult_ax {x <- 1, y <- x}

  monoproof: PROVE monotonicity FROM
    gc_ax,
    diminish {x <- abs(T1 - T2)},
    abs_ax {a <- clock(p, T1) - clock(p, T2) - (T1 - T2)},
    abs_ax {a <- T1 - T2}

END clocks
```

```
algorithm: MODULE

USING clocks, sums

EXPORTING Sigma, Delta, Delta1, Delta2, D2bar, skew, S1A, S1C, S2,
  delta, eps, delta0, n, m WITH clocks

THEORY

  T, T0, T1, X, PI: VAR clocktime

  i: VAR period

  p, q, r: VAR proc

  Delta1: function[proc, period -> clocktime]

  Delta2, D2bar: function[proc, proc, period -> clocktime]

  m, n: proc

  eps, delta0, delta: realtime

  Sigma, Delta: clocktime

  C0_a: AXIOM n > 0

  C0_b: AXIOM 0 <= m AND m < n

  C0_c: AXIOM Delta > 0

  C2: AXIOM S >= Sigma

  C3: AXIOM Sigma >= Delta

  C4: AXIOM Delta >= delta + eps + mult(half(rho), S)

  C5: AXIOM delta >= delta0 + rho * R

  C6: AXIOM delta
      >= 2 * (eps + rho * S) + 2 * m * Delta / (n - m)
            + n * rho * R / (n - m)
          + rho * Delta
        + n * rho * Sigma / (n - m)
```

```
C2and3: LEMMA Delta <= S

Alg1: AXIOM Corr(p, i + 1) = Corr(p, i) + Delta1(p, i)

Alg2: AXIOM
  Delta1(p, i) = mean(1, n, (LAMBDA r -> number : D2bar(r, p, i)))

Alg3: AXIOM
  D2bar(r, p, i)
    = IF r /= p AND abs(Delta2(r, p, i)) < Delta
      THEN Delta2(r, p, i)
      ELSE 0
      END IF

clock_prop: LEMMA rt(p, i + 1, T) = rt(p, i, T + Delta1(p, i))

D2bar_prop: LEMMA abs(D2bar(p, q, i)) < Delta

skew: function[proc, proc, clocktime, period -> clocktime] =
  (LAMBDA p, q, T, i -> clocktime : abs(rt(p, i, T) - rt(q, i, T)))

S1A: function[period -> bool]

S1Adef: AXIOM
  S1A(i)
    = (FORALL r : (m + 1 <= r AND r <= n) IMPLIES nonfaulty(r, i))

S1C: function[proc, proc, period -> bool]

S1Cdef: AXIOM
  S1C(p, q, i)
    = (nonfaulty(p, i) AND nonfaulty(q, i) AND in_R_interval(T, i)
         IMPLIES skew(p, q, T, i) <= delta)

S1C_lemma: LEMMA S1C(p, q, i) IMPLIES S1C(q, p, i)

S2: function[proc, period -> bool]

S2_ax: AXIOM S2(p, i) = (abs(Corr(p, i + 1) - Corr(p, i)) < Sigma)

A0: AXIOM skew(p, q, T_sup(0), 0) < delta0

A2: AXIOM nonfaulty(p, i)
      AND nonfaulty(q, i) AND S1C(p, q, i) AND S2(p, i)
    IMPLIES abs(Delta2(q, p, i)) <= S
      AND (EXISTS T0 :
         in_S_interval(T0, i)
```

```
            AND abs(rt(p, i, T0 + Delta2(q, p, i)) - rt(q, i, T0))
              < eps)

  A2_aux: AXIOM Delta2(p, p, i) = 0

  Theorem_1: THEOREM S1A(i) IMPLIES S1C(p, q, i)

  Theorem_2: THEOREM S2(p, i)

PROOF

  C2and3_proof: PROVE C2and3 FROM C2, C3

  clock_proof: PROVE clock_prop FROM
    clockdef {T <- T + Delta1(p, i)}, clockdef {i <- i + 1}, Alg1

  D2bar_prop_proof: PROVE D2bar_prop FROM
    Alg3 {r <- p, p <- q}, C0_c, abs_ax0

  S1C_lemma_proof: PROVE S1C_lemma FROM
    S1Cdef,
    S1Cdef {p <- q, q <- p},
    abs_ax4 {x <- rt(q, i, T@p1), y <- rt(p, i, T@p1)}

  Theorem_2_proof: PROVE Theorem_2 FROM
    S2_ax,
    Alg1,
    D2bar_prop {p <- pp@p7, q <- p},
    Alg2,
    C0_a,
    C0_c,
    mean_bound
      {i <- 1,
       j <- n,
       x <- Delta,
       F <- (LAMBDA r -> number : abs(D2bar(r, p, i)))},
    abs_mean
      {i <- 1,
       j <- n,
       F <- (LAMBDA r -> number : D2bar(r, p, i))},
    C3

END algorithm
```

```
clockprops: MODULE

USING clocks, algorithm, natinduction

THEORY

  T, TO, T1, T2, TN, PI: VAR clocktime

  p, q: VAR proc

  i: VAR period

  upper_bound: LEMMA
    in_S_interval(T, i) AND abs(PI) <= R - S
      IMPLIES adjusted(p, i, T + PI) <= adjusted(p, i + 1, T_sup(i + 2))

  lower_bound: LEMMA
    0 <= PI IMPLIES adjusted(p, 0, T_sup(0))
        <= adjusted(p, i, T_sup(i) + PI)

  lower_bound2: LEMMA
    in_S_interval(T, i) AND abs(PI) <= R - S
      IMPLIES adjusted(p, 0, T_sup(0)) <= adjusted(p, i, T + PI)

  adj_always_pos: LEMMA adjusted(p, i, T_sup(i)) >= T_ZERO

  nonfx: LEMMA nonfaulty(p, i + 1) IMPLIES nonfaulty(p, i)

  S1A_lemma: LEMMA S1A(i + 1) IMPLIES S1A(i)

PROOF

  i2R: LEMMA T_sup(i + 2) = T_sup(i) + 2 * R

  i2R_proof: PROVE i2R FROM T_sup_ax {i <- i + 2}, T_sup_ax

  upper_bound_proof: PROVE upper_bound FROM
    Sdef,
    i2R,
    abs_ax6 {x <- PI, y <- R - S},
    S2_ax,
    Theorem_2,
    abs_ax6 {x <- Corr(p, i + 1) - Corr(p, i), y <- Sigma},
    C2
```

```
basis: LEMMA adjusted(p, 0, T_sup(0)) >= T_ZERO

basis_proof: PROVE basis FROM zero_correction, T_sup_ax {i <- 0}

small_shift: LEMMA Corr(p, i + 1) - Corr(p, i) >= -R

small_shift_proof: PROVE small_shift FROM
  S2_ax,
  Theorem_2,
  abs_ax {a <- Corr(p, i + 1) - Corr(p, i)},
  C2,
  SinR

inductive_step: LEMMA
  adjusted(p, i, T_sup(i)) >= T_ZERO
    IMPLIES adjusted(p, i + 1, T_sup(i + 1)) >= T_ZERO

ind_proof: PROVE inductive_step FROM small_shift, T_next

adj_pos_proof: PROVE adj_always_pos FROM
  induction
    {n <- i,
     prop <- (LAMBDA i -> bool : adjusted(p, i, T_sup(i)) >= T_ZERO)},
  basis,
  inductive_step {i <- i@p1}

lower_bound_proof: PROVE lower_bound FROM
  adj_always_pos, T_sup_ax {i <- 0}, zero_correction

lower_bound2_proof: PROVE lower_bound2 FROM
  lower_bound {PI <- T - T_sup(i) + PI@c},
  Sdef,
  abs_ax {a <- PI},
  SinR

gc_prop: LEMMA
  goodclock(p, T0, TN) AND T0 <= T AND T <= TN
    IMPLIES goodclock(p, T0, T)

gc_proof: PROVE gc_prop FROM
  gc_ax {T1 <- T1@p2, T2 <- T2@p2}, gc_ax {TN <- T}

bounds: LEMMA
  adjusted(p, 0, T_sup(0)) <= adjusted(p, i, T_sup(i + 1))
    AND adjusted(p, i, T_sup(i + 1))
      <= adjusted(p, i + 1, T_sup(i + 2))
```

```
  bounds_proof: PROVE bounds FROM
    upper_bound {PI <- 0, T <- T_sup(i + 1)},
    lower_bound2 {PI <- 0, T <- T_sup(i + 1)},
    abs_ax0,
    SinR,
    Ti_in_S

  nonfx_proof: PROVE nonfx FROM
    A1 {i <- i + 1},
    A1,
    gc_prop
      {T0 <- adjusted(p, 0, T_sup(0)),
       TN <- adjusted(p, i + 1, T_sup(i + 2)),
       T <- adjusted(p, i, T_sup(i + 1))},
    bounds

  S1A_lemma_proof: PROVE S1A_lemma FROM
    S1Adef,
    S1Adef {i <- i + 1, r <- r@p1},
    nonfx {p <- r@p1}

END clockprops
```

```
lemma1: MODULE

USING algorithm, lemma2

THEORY

  p, q: VAR proc

  i: VAR period

  lemma1def: LEMMA
    S1C(p, q, i)
        AND S2(p, i) AND nonfaulty(p, i + 1) AND nonfaulty(q, i + 1)
      IMPLIES abs(Delta2(q, p, i)) < Delta

PROOF

  lemma1_proof: PROVE lemma1def FROM
    A2,
    lemma2c {PI <- Delta2(q, p, i), T <- T0@p1},
    S1Cdef {T <- T0@p1},
    abs_ax4 {x <- rt(p, i, T0@p1), y <- rt(q, i, T0@p1)},
    abs_ax4
      {x <- rt(p, i, T0@p1 + PI@p2),
       y <- rt(p, i, T0@p1) + PI@p2},
    abs_ax2b {x <- y@p5 - x@p5, y <- y@p4 - x@p4, z <- x@p5 - y@p4},
    nonfx,
    nonfx {p <- q},
    inRS {T <- T0@p1},
    mult4 {x <- half(rho), y <- abs(Delta2(q, p, i)), z <- S},
    rho_pos,
    C4

END lemma1
```

```
lemma2: MODULE

USING algorithm, clockprops

THEORY

  p, q, r: VAR proc

  i: VAR period

  T: VAR clocktime

  PI, PHI: VAR realtime

  lemma2def: LEMMA
    nonfaulty(p, i + 1)
        AND adjusted(p, i, T) <= adjusted(p, i + 1, T_sup(i + 2))
          AND adjusted(p, 0, T_sup(0)) <= adjusted(p, i, T)
            AND adjusted(p, i, T + PI)
                <= adjusted(p, i + 1, T_sup(i + 2))
              AND adjusted(p, 0, T_sup(0)) <= adjusted(p, i, T + PI)
      IMPLIES abs(rt(p, i, T + PI) - (rt(p, i, T) + PI))
        < mult(half(rho), abs(PI))

  lemma2a: LEMMA
    nonfaulty(p, i + 1)
        AND abs(PI + PHI) <= R - S
          AND abs(PHI) <= R - S AND in_S_interval(T, i)
      IMPLIES abs(rt(p, i, T + PHI + PI) - (rt(p, i, T + PHI) + PI))
        < mult(half(rho), abs(PI))

  lemma2b: LEMMA
    nonfaulty(p, i + 1)
        AND abs(PHI) <= S AND abs(PI) <= S AND in_S_interval(T, i)
      IMPLIES abs(rt(p, i, T + PHI + PI) - (rt(p, i, T + PHI) + PI))
        < mult(half(rho), abs(PI))

  lemma2c: LEMMA
    nonfaulty(p, i + 1) AND abs(PI) <= S AND in_S_interval(T, i)
      IMPLIES abs(rt(p, i, T + PI) - (rt(p, i, T) + PI))
        < mult(half(rho), abs(PI))

  lemma2d: LEMMA
    nonfaulty(p, i) AND 0 <= PI AND PI <= R
      IMPLIES abs(rt(p, i, T_sup(i) + PI) - (rt(p, i, T_sup(i)) + PI))
```

```
      < mult(half(rho), PI)

PROOF

  lemma2_proof: PROVE lemma2def FROM
    A1 {i <- i + 1},
    gc_ax
      {T0 <- adjusted(p, 0, T_sup(0)),
       TN <- adjusted(p, i + 1, T_sup(i + 2)),
       T2 <- adjusted(p, i, T),
       T1 <- adjusted(p, i, T + PI)},
    clockdef,
    clockdef {T <- T + PI}

  lemma2a_proof: PROVE lemma2a FROM
    lemma2def {T <- T + PHI},
    upper_bound {PI <- PHI + PI},
    lower_bound2 {PI <- PHI + PI},
    upper_bound {PI <- PHI},
    lower_bound2 {PI <- PHI}

  lemma2b_proof: PROVE lemma2b FROM
    lemma2a,
    abs_ax1 {x <- PI},
    abs_ax2 {x <- PHI, y <- PI},
    C1,
    posS,
    posR

  lemma2c_proof: PROVE lemma2c FROM lemma2b {PHI <- 0}, abs_ax0, posS

  lemma2d_proof: PROVE lemma2d FROM
    A1,
    gc_ax
      {T0 <- adjusted(p, 0, T_sup(0)),
       TN <- adjusted(p, i, T_sup(i + 1)),
       T1 <- adjusted(p, i, T_sup(i) + PI),
       T2 <- adjusted(p, i, T_sup(i))},
    clockdef {T <- T_sup(i)},
    clockdef {T <- T_sup(i) + PI},
    posR,
    pos_abs {x <- PI},
    lower_bound,
    lower_bound {PI <- 0},
    T_next

END lemma2
```

```
lemma3: MODULE

USING algorithm, lemma2

THEORY

  p, q: VAR proc

  i: VAR period

  T, T0, T1, T2: VAR clocktime

  PI: VAR realtime

  lemma3def: LEMMA
    S1C(p, q, i)
        AND S2(p, i)
          AND nonfaulty(p, i + 1)
            AND nonfaulty(q, i + 1) AND in_S_interval(T, i)
      IMPLIES abs(rt(p, i, T + Delta2(q, p, i)) - rt(q, i, T))
        < eps + rho * S

PROOF

  lemma3_proof: PROVE lemma3def FROM
    A2,
    rearrange_alt
      {x <- rt(p, i, T + Delta2(q, p, i)),
       y <- rt(q, i, T),
       u <- rt(p, i, T0@p1 + Delta2(q, p, i)),
       v <- T - T0@p1,
       w <- rt(q, i, T0@p1)},
    lemma2b {T <- T0@p1, PHI <- Delta2(q, p, i), PI <- T - T0@p1},
    lemma2c {p <- q, T <- T0@p1, PI <- T - T0@p1},
    nonfx,
    nonfx {p <- q},
    mult4 {x <- half(rho), y <- abs(T - T0@p1), z <- S},
    rho_pos,
    half3 {x <- rho, y <- S},
    mult_ax {x <- rho, y <- S},
    in_S_lemma {T1 <- T, T2 <- T0@p1}

END lemma3
```

```
lemma4: MODULE

USING algorithm, lemma1, lemma2, lemma3

THEORY

  p, q, r: VAR proc

  i: VAR period

  T: VAR clocktime

  lemma4def: LEMMA
    S1C(q, r, i)
        AND S1C(p, q, i)
          AND S1C(p, r, i)
            AND S2(p, i)
              AND S2(q, i)
                AND S2(r, i)
                  AND nonfaulty(p, i + 1)
                    AND nonfaulty(q, i + 1)
                      AND nonfaulty(r, i + 1) AND in_S_interval(T, i)
      IMPLIES abs(rt(p, i, T) + D2bar(r, p, i)
                  - (rt(q, i, T) + D2bar(r, q, i)))
        < 2 * (eps + rho * S + mult(half(rho), Delta))

PROOF

  T0, T1, T2: VAR clocktime

  PI: VAR realtime

  u, v, w, x, y, z: VAR number

  rearrange1: LEMMA x - y = (u - y) - (v - x) + (v - w) - (u - w)

  rearrange1_proof: PROVE rearrange1

  rearrange2: LEMMA
    abs((u - y) - (v - x) + (v - w) - (u - w))
      <= abs(u - y) + abs(v - x) + abs(v - w) + abs(u - w)

  rearrange2_proof: PROVE rearrange2 FROM
    abs_ax2c {w <- (u - y), x <- (x - v), y <- (v - w), z <- (w - u)},
    abs_ax3 {x <- (v - x)},
```

```
    abs_ax3 {x <- (u - w)}

  rearrange3: LEMMA
    abs(x - y) <= abs(u - y) + abs(v - x) + abs(v - w) + abs(u - w)

  rearrange3_proof: PROVE rearrange3 FROM rearrange1, rearrange2

  sublemma1: LEMMA
    S1C(p, r, i)
        AND S2(p, i) AND nonfaulty(p, i + 1) AND nonfaulty(r, i + 1)
      IMPLIES D2bar(r, p, i) = Delta2(r, p, i)

  sublemma1_proof: PROVE sublemma1 FROM
    lemma1def {q <- r}, Alg3, A2_aux

  lemma2x: LEMMA
    S1C(p, r, i)
        AND S2(p, i)
          AND nonfaulty(p, i + 1)
            AND nonfaulty(r, i + 1) AND in_S_interval(T, i)
      IMPLIES abs(rt(p, i, T + Delta2(r, p, i))
                     - (rt(p, i, T) + Delta2(r, p, i)))
        < mult(half(rho), Delta)

  lemma2x_proof: PROVE lemma2x FROM
    lemma2c {PI <- Delta2(r, p, i)},
    lemma1def {q <- r},
    C2and3,
    mult4 {x <- half(rho), y <- abs(Delta2(r, p, i)), z <- Delta},
    rho_pos

  lemma4_proof: PROVE lemma4def FROM
    rearrange3
      {x <- rt(p, i, T) + D2bar(r, p, i),
       y <- rt(q, i, T) + D2bar(r, q, i),
       u <- rt(q, i, T + Delta2(r, q, i)),
       v <- rt(p, i, T + Delta2(r, p, i)),
       w <- rt(r, i, T)},
    sublemma1,
    sublemma1 {p <- q},
    lemma2x,
    lemma2x {p <- q},
    lemma3def {q <- r},
    lemma3def {p <- q, q <- r},
    S1C_lemma

END lemma4
```

```
lemma5: MODULE

USING algorithm, clockprops

THEORY

  p, q, r: VAR proc

  T: VAR clocktime

  i, j: VAR period

  lemma5def: LEMMA
    S1C(p, q, i)
        AND nonfaulty(p, i + 1)
          AND nonfaulty(q, i + 1) AND in_S_interval(T, i)
      IMPLIES abs(rt(p, i, T) + D2bar(r, p, i)
                   - (rt(q, i, T) + D2bar(r, q, i)))
        < delta + 2 * Delta

PROOF

  a, b, x, y: VAR clocktime

  rearrange1: LEMMA (a + x) - (b + y) = (a - b) + x - y

  rearrange1_proof: PROVE rearrange1

  rearrange2: LEMMA
    abs((a + x) - (b + y)) <= abs(a - b) + abs(x) + abs(y)

  rearrange2_proof: PROVE rearrange2 FROM
    rearrange1, abs_ax8, abs_ax2 {x <- (a - b), y <- (x - y)}

  lemma5proof: PROVE lemma5def FROM
    rearrange2
      {a <- rt(p, i, T),
       b <- rt(q, i, T),
       x <- D2bar(r, p, i),
       y <- D2bar(r, q, i)},
    D2bar_prop {p <- r, q <- p},
    D2bar_prop {p <- r, q <- q},
    inRS,
    S1Cdef,
    nonfx,
```

```
    nonfx {p <- q}

END lemma5
```

```
lemma6: MODULE

USING algorithm, clockprops, lemma2

THEORY

  p, q: VAR proc

  i: VAR period

  T, PI: VAR clocktime

  sublemma_A: LEMMA
    nonfaulty(p, i)
        AND nonfaulty(q, i) AND in_R_interval(T, i)
      IMPLIES skew(p, q, T, i)
        < skew(p, q, T_sup(i), i) + rho * R

  lemma6def: LEMMA
    nonfaulty(p, i + 1)
        AND nonfaulty(q, i + 1) AND in_R_interval(T, i + 1)
      IMPLIES skew(p, q, T, i + 1)
        < abs(rt(p, i, T_sup(i + 1)) + Delta1(p, i)
                - (rt(q, i, T_sup(i + 1)) + Delta1(q, i)))
            + rho * R
          + rho * Sigma

PROOF

  sublemma1: LEMMA
    0 <= PI AND PI <= R IMPLIES 2 * mult(half(rho), PI) <= rho * R

  sub1_proof: PROVE sublemma1 FROM
    mult2 {x <- half(rho), y <- R},
    times_half {x <- rho},
    mult4 {x <- half(rho), y <- PI, z <- R},
    rho_pos,
    mult_ax {x <- rho, y <- R}

  sub_A_proof: PROVE sublemma_A FROM
    Rdef,
    rearrange_alt
      {x <- rt(p, i, T),
       y <- rt(q, i, T),
       u <- rt(p, i, T_sup(i)),
```

```
        v <- PI@p1,
        w <- rt(q, i, T_sup(i))},
     lemma2d {PI <- PI@p1},
     lemma2d {p <- q, PI <- PI@p1},
     sublemma1 {PI <- PI@p1}

  sublemma2: LEMMA
     skew(p, q, T, i + 1)
       = abs(rt(p, i, T + Delta1(p, i)) - rt(q, i, T + Delta1(q, i)))

  sub2_proof: PROVE sublemma2 FROM clock_prop, clock_prop {p <- q}

  lemma6_proof: PROVE lemma6def FROM
     sublemma_A{i <- i + 1},
     sublemma2 {T <- T_sup(i + 1)},
     rearrange
       {x <- rt(p, i, T_sup(i + 1) + Delta1(p, i)),
        y <- rt(q, i, T_sup(i + 1) + Delta1(q, i)),
        u <- rt(p, i, T_sup(i + 1)),
        v <- Delta1(p, i),
        w <- rt(q, i, T_sup(i + 1)),
        z <- Delta1(q, i)},
     lemma2c {T <- T_sup(i + 1), PI <- Delta1(p, i)},
     lemma2c
       {T <- T_sup(i + 1),
        PI <- Delta1(q, i),
        p <- q},
     Alg1,
     Alg1 {p <- q},
     S2_ax,
     S2_ax {p <- q},
     Theorem_2,
     Theorem_2 {p <- q},
     mult4 {x <- half(rho), y <- abs(Delta1(p,i)) , z <- Sigma},
     mult4 {x <- half(rho), y <- abs(Delta1(q,i)) , z <- Sigma},
     rho_pos,
     Ti_in_S,
     C2,
     half3 {x <- rho, y <- Sigma},
     mult_ax {x <- rho, y <- Sigma}

END lemma6
```

```
summations: MODULE

USING algorithm, sums, lemma4, lemma5, lemma6

THEORY

  p, q, r: VAR proc

  T: VAR clocktime

  i: VAR period

  culmination: LEMMA
    S1A(i + 1) AND S1C(p, q, i)
      IMPLIES (nonfaulty(p, i + 1)
             AND nonfaulty(q, i + 1) AND in_R_interval(T, i + 1)
           IMPLIES skew(p, q, T, i + 1)
             <= ((delta + 2 * Delta) * m
                        + 2 * (rho * S + eps
                                 + mult(half(rho), Delta))
                          * (n - m))
                   / n
                + rho * R
              + rho * Sigma)

PROOF

  l1: LEMMA abs(rt(p, i, T_sup(i + 1)) + Deltai(p, i)
                - (rt(q, i, T_sup(i + 1)) + Deltai(q, i)))
      <= mean(1,
              n,
              (LAMBDA r -> number :
                 abs(rt(p, i, T_sup(i + 1)) + D2bar(r, p, i)
                       - (rt(q, i, T_sup(i + 1)) + D2bar(r, q, i)))))

  l2: LEMMA abs(rt(p, i, T_sup(i + 1)) + Deltai(p, i)
                - (rt(q, i, T_sup(i + 1)) + Deltai(q, i)))
      <= (sum(1,
              m,
              (LAMBDA r -> number :
                 abs(rt(p, i, T_sup(i + 1)) + D2bar(r, p, i)
                       - (rt(q, i, T_sup(i + 1)) + D2bar(r, q, i)))))
             + sum(m + 1,
                   n,
                   (LAMBDA r -> number :
```

```
                  abs(rt(p, i, T_sup(i + 1)) + D2bar(r, p, i)
                       - (rt(q, i, T_sup(i + 1))
                           + D2bar(r, q, i))))))
      / n

13: LEMMA S1A(i + 1)
      AND S1C(p, q, i) AND nonfaulty(p, i + 1) AND nonfaulty(q, i + 1)
    IMPLIES sum(1,
                m,
                (LAMBDA r -> number :
                   abs(rt(p, i, T_sup(i + 1)) + D2bar(r, p, i)
                        - (rt(q, i, T_sup(i + 1))
                            + D2bar(r, q, i)))))
      <= (delta + 2 * Delta) * m

14: LEMMA S1A(i + 1)
      AND S1C(p, q, i) AND nonfaulty(p, i + 1) AND nonfaulty(q, i + 1)
    IMPLIES sum(m + 1,
                n,
                (LAMBDA r -> number :
                   abs(rt(p, i, T_sup(i + 1)) + D2bar(r, p, i)
                        - (rt(q, i, T_sup(i + 1))
                            + D2bar(r, q, i)))))
      <= 2 * (rho * S + eps + mult(half(rho), Delta)) * (n - m)

15: LEMMA S1A(i + 1)
      AND S1C(p, q, i) AND nonfaulty(p, i + 1) AND nonfaulty(q, i + 1)
    IMPLIES abs(rt(p, i, T_sup(i + 1)) + Delta1(p, i)
                 - (rt(q, i, T_sup(i + 1)) + Delta1(q, i)))
      <= ((delta + 2 * Delta) * m
            + 2 * (rho * S + eps + mult(half(rho), Delta))
              * (n - m))
        / n

l1_proof: PROVE l1 FROM
  Alg2,
  Alg2 {p <- q},
  rearrange_sum
    {x <- rt(p, i, T_sup(i + 1)),
     y <- rt(q, i, T_sup(i + 1)),
     F <- (LAMBDA r -> number : D2bar(r, p, i)),
     G <- (LAMBDA r -> number : D2bar(r, q, i)),
     i <- 1,
     j <- n},
  abs_mean
    {i <- 1,
     j <- n,
```

```
    F <- (LAMBDA r -> number :
        x@p3 + D2bar(r, p, i) - (y@p3 + D2bar(r, q, i)))},
  CO_a

l2_proof: PROVE l2 FROM
  l1,
  split_mean
    {i <- 1,
     j <- n,
     k <- m,
     F <- (LAMBDA r -> number :
         abs(rt(p, i, T_sup(i + 1)) + D2bar(r, p, i)
               - (rt(q, i, T_sup(i + 1)) + D2bar(r, q, i))))},
  CO_a,
  CO_b

bound_faulty: LEMMA
  S1A(i + 1)
      AND S1C(p, q, i)
        AND 1 <= r
          AND r <= m AND nonfaulty(p, i + 1) AND nonfaulty(q, i + 1)
    IMPLIES abs(rt(p, i, T_sup(i + 1)) + D2bar(r, p, i)
                  - (rt(q, i, T_sup(i + 1)) + D2bar(r, q, i)))
      < delta + 2 * Delta

bound_faulty_proof: PROVE bound_faulty FROM
  lemma5def {T <- T_sup(i + 1)}, Ti_in_S

l3_proof: PROVE l3 FROM
  sum_bound
    {F <- (LAMBDA r -> number :
         abs(rt(p, i, T_sup(i + 1)) + D2bar(r, p, i)
               - (rt(q, i, T_sup(i + 1)) + D2bar(r, q, i)))),
     x <- delta + 2 * Delta,
     i <- 1,
     j <- m},
  bound_faulty {r <- pp@p1},
  CO_b

S2_pqr: LEMMA S2(p, i) AND S2(q, i) AND S2(r, i)

S2_pqr_proof: PROVE S2_pqr FROM
  Theorem_2, Theorem_2 {p <- q}, Theorem_2 {p <- r}

bound_nonfaulty: LEMMA
  S1A(i + 1)
      AND S1C(p, q, i)
```

```
      AND m + 1 <= r
        AND r <= n AND nonfaulty(p, i + 1) AND nonfaulty(q, i + 1)
    IMPLIES abs(rt(p, i, T_sup(i + 1)) + D2bar(r, p, i)
                 - (rt(q, i, T_sup(i + 1)) + D2bar(r, q, i)))
      < 2 * (rho * S + eps + mult(half(rho), Delta))

bound_nonfaulty_proof: PROVE bound_nonfaulty FROM
  S1Adef {i <- i + 1},
  S1A_lemma,
  S1Adef,
  nonfx,
  nonfx {p <- q},
  Theorem_1 {q <- r},
  Theorem_1 {p <- q, q <- r},
  S2_pqr,
  lemma4def {T <- T_sup(i + 1)},
  Ti_in_S

14_proof: PROVE 14 FROM
  sum_bound
    {F <- (LAMBDA r -> number :
         abs(rt(p, i, T_sup(i + 1)) + D2bar(r, p, i)
               - (rt(q, i, T_sup(i + 1)) + D2bar(r, q, i)))),
     x <- 2 * (rho * S + eps + mult(half(rho), Delta)),
     i <- m + 1,
     j <- n},
  bound_nonfaulty {r <- pp@p1},
  C0_b

15_proof: PROVE 15 FROM
  12,
  13,
  14,
  div_mon2
    {x <- sum(1,
              m,
              (LAMBDA r -> number :
                 abs(rt(p, i, T_sup(i + 1)) + D2bar(r, p, i)
                       - (rt(q, i, T_sup(i + 1)) + D2bar(r, q, i)))))
       + sum(m + 1,
              n,
              (LAMBDA r -> number :
                 abs(rt(p, i, T_sup(i + 1)) + D2bar(r, p, i)
                       - (rt(q, i, T_sup(i + 1)) + D2bar(r, q, i))))),
     y <- (delta + 2 * Delta) * m
       + 2 * (rho * S + eps + mult(half(rho), Delta)) * (n - m),
     z <- n},
```

```
    CO_a

  culm_proof: PROVE culmination FROM lemma6def, 15, S1Adef {i <- i + 1}

END summations
```

```
juggle: MODULE

USING algorithm

THEORY

  rearrange_delta: LEMMA
    delta >= 2 * (eps + rho * S) + 2 * m * Delta / (n - m)
               + n * rho * R / (n - m)
             + rho * Delta
           + n * rho * Sigma / (n - m)
      IMPLIES delta
        >= ((delta + 2 * Delta) * m
                  + 2 * (eps + rho * S + mult(half(rho), Delta))
                    * (n - m))
              / n
            + rho * R
          + rho * Sigma

PROOF

  a, b, b1, b2, b3, b4, b5, b6, c, x, y: VAR number

  distrib6: LEMMA
    (b1 + b2 + b3 + b4 + b5 + b6) * c
      = b1 * c + b2 * c + b3 * c + b4 * c + b5 * c + b6 * c

  distrib6_proof: PROVE distrib6

  distrib6_mult: LEMMA
    mult((b1 + b2 + b3 + b4 + b5 + b6), c)
      = mult(b1, c) + mult(b2, c) + mult(b3, c) + mult(b4, c)
          + mult(b5, c)
        + mult(b6, c)

  distrib6_mult_proof: PROVE distrib6_mult FROM
    distrib6,
    mult_ax {x <- b1 + b2 + b3 + b4 + b5 + b6, y <- c},
    mult_ax {x <- b1, y <- c},
    mult_ax {x <- b2, y <- c},
    mult_ax {x <- b3, y <- c},
    mult_ax {x <- b4, y <- c},
    mult_ax {x <- b5, y <- c},
    mult_ax {x <- b6, y <- c}
```

```
mult_ineq1: LEMMA
  a >= b1 + b2 + b3 + b4 + b5 AND c > 0
    IMPLIES mult(a, c)
      >= mult(b1, c) + mult(b2, c) + mult(b3, c) + mult(b4, c)
        + mult(b5, c)

mult_ineq1_proof: PROVE mult_ineq1 FROM
  distrib6_mult {b6 <- 0},
  mult_mon2 {x <- b1 + b2 + b3 + b4 + b5, y <- a, z <- c},
  mult_ax {x <- 0, y <- c}

distrib6_div: LEMMA
  c > 0 IMPLIES (b1 + b2 + b3 + b4 + b5 + b6) / c
      = b1 / c + b2 / c + b3 / c + b4 / c + b5 / c + b6 / c

reciprocal: LEMMA y /= 0 IMPLIES mult(x, 1 / y) = x / y

reciprocal_proof: PROVE reciprocal FROM quotient_ax, mult_ax {y <- 1/y}

distrib6_div_proof: PROVE distrib6_div FROM
  distrib6_mult {c <- 1 / c},
  reciprocal {x <- b1 + b2 + b3 + b4 + b5 + b6, y <- c},
  reciprocal {x <- b1, y <- c},
  reciprocal {x <- b2, y <- c},
  reciprocal {x <- b3, y <- c},
  reciprocal {x <- b4, y <- c},
  reciprocal {x <- b5, y <- c},
  reciprocal {x <- b6, y <- c}

cancel_mult: LEMMA c > 0 AND mult(a, c) >= b IMPLIES a >= b / c

cancel_mult_proof: PROVE cancel_mult FROM
  div_mon2 {z <- c, x <- b, y <- mult(a, c)},
  cancellation_mult {x <- a, y <- c}

mult_ineq2: LEMMA
  c > 0 AND mult(a, c) >= b1 + b2 + b3 + b4 + b5 + b6
    IMPLIES a >= b1 / c + b2 / c + b3 / c + b4 / c + b5 / c + b6 / c

mult_ineq2_proof: PROVE mult_ineq2 FROM
  cancel_mult {b <- b1 + b2 + b3 + b4 + b5 + b6}, distrib6_div

distrib4_div: LEMMA
  c > 0 IMPLIES b1 / c + b2 / c + b3 / c + b4 / c
      = (b1 + b2 + b3 + b4) / c

distrib4_div_proof: PROVE distrib4_div FROM
```

```
    distrib6_mult {b5 <- 0, b6 <- 0, c <- 1 / c},
    reciprocal {x <- b1 + b2 + b3 + b4, y <- c},
    reciprocal {x <- b1, y <- c},
    reciprocal {x <- b2, y <- c},
    reciprocal {x <- b3, y <- c},
    reciprocal {x <- b4, y <- c},
    mult_ax {x <- 0, y <- 1 / c}

  step1: LEMMA
    delta >= 2 * (eps + rho * S) + 2 * m * Delta / (n - m)
              + n * rho * R / (n - m)
            + rho * Delta
          + n * rho * Sigma / (n - m)
      IMPLIES mult(delta, n - m)
        >= mult(2 * (eps + rho * S), n - m) + 2 * m * Delta
              + n * rho * R
            + mult(rho * Delta, n - m)
          + n * rho * Sigma

  step1_proof: PROVE step1 FROM
    mult_ineq1
      {a <- delta,
       c <- n - m,
       b1 <- 2 * (eps + rho * S),
       b2 <- 2 * m * Delta / (n - m),
       b3 <- n * rho * R / (n - m),
       b4 <- rho * Delta,
       b5 <- n * rho * Sigma / (n - m)},
    mult_div {x <- 2 * m * Delta, y <- n - m},
    mult_div {x <- n * rho * R, y <- n - m},
    mult_div {x <- n * rho * Sigma, y <- n - m},
    CO_b

  step2: LEMMA
    mult(delta, n - m)
        >= mult(2 * (eps + rho * S), n - m) + 2 * m * Delta
              + n * rho * R
            + mult(rho * Delta, n - m)
          + n * rho * Sigma
      IMPLIES mult(delta, n)
        >= mult(delta, m) + mult(2 * (eps + rho * S), n - m)
                + 2 * m * Delta
              + n * rho * R
            + mult(rho * Delta, n - m)
          + n * rho * Sigma

  step2_proof: PROVE step2 FROM
```

```
  mult_ax {x <- delta, y <- n - m},
  mult_ax {x <- delta, y <- n},
  mult_ax {x <- delta, y <- m}

step3: LEMMA
  mult(delta, n)
      >= mult(delta, m) + mult(2 * (eps + rho * S), n - m)
              + 2 * m * Delta
            + n * rho * R
          + mult(rho * Delta, n - m)
        + n * rho * Sigma
    IMPLIES delta
      >= mult(delta, m) / n + mult(2 * (eps + rho * S), n - m) / n
              + 2 * m * Delta / n
            + rho * R
          + mult(rho * Delta, n - m) / n
        + rho * Sigma

step3_proof: PROVE step3 FROM
  mult_ineq2
    {a <- delta,
     c <- n,
     b1 <- mult(delta, m),
     b2 <- mult(2 * (eps + rho * S), n - m),
     b3 <- 2 * m * Delta,
     b4 <- n * rho * R,
     b5 <- mult(rho * Delta, n - m),
     b6 <- n * rho * Sigma},
  cancellation {x <- rho * R, y <- n},
  cancellation {x <- rho * Sigma, y <- n},
  CO_a

step4: LEMMA
  delta >= mult(delta, m) / n + mult(2 * (eps + rho * S), n - m) / n
              + 2 * m * Delta / n
            + rho * R
          + mult(rho * Delta, n - m) / n
        + rho * Sigma
    IMPLIES delta
      >= (mult(delta, m) + mult(2 * (eps + rho * S), n - m)
                  + 2 * m * Delta
                + mult(rho * Delta, n - m))
            / n
          + rho * R
        + rho * Sigma

step4_proof: PROVE step4 FROM
```

```
    CO_a,
    distrib4_div
      {c <- n,
       b1 <- mult(delta, m),
       b2 <- mult(2 * (eps + rho * S), n - m),
       b3 <- 2 * m * Delta,
       b4 <- mult(rho * Delta, n - m)}

  step5: LEMMA
    delta >= (mult(delta, m) + mult(2 * (eps + rho * S), n - m)
                    + 2 * m * Delta
                  + mult(rho * Delta, n - m))
             / n
            + rho * R
          + rho * Sigma
      IMPLIES delta
        >= ((delta + 2 * Delta) * m
                  + 2 * (eps + rho * S + mult(half(rho), Delta))
                    * (n - m))
             / n
            + rho * R
          + rho * Sigma

  step5_proof: PROVE step5 FROM
    mult_ax {x <- delta, y <- m},
    mult_ax {x <- rho * Delta, y <- n - m},
    mult_ax {x <- 2 * (eps + rho * S), y <- n - m},
    half3 {x <- rho, y <- Delta},
    mult_ax {x <- rho, y <- Delta}

  final: PROVE rearrange_delta FROM step1, step2, step3, step4, step5

END juggle
```

```
main: MODULE

USING natinduction, algorithm, lemma6, summations, juggle

PROOF

  p, q, r: VAR proc

  i, j, k: VAR period

  T: VAR clocktime

  basis: LEMMA S1A(0) IMPLIES S1C(p, q, 0)

  basis_proof: PROVE basis FROM
    S1Adef {i <- 0}, sublemma_A {i <- 0}, S1Cdef {i <- 0}, A0, C5

  ind_step: LEMMA
    S1A(i + 1) AND S1C(p, q, i) IMPLIES S1C(p, q, i + 1)

  ind_proof: PROVE ind_step FROM
    culmination, rearrange_delta, S1Cdef {i <- i + 1}, C6

  Theorem_1_proof: PROVE Theorem_1 FROM
    basis,
    ind_step {i <- i@p3},
    mod_induction
      {n <- i,
       A <- (LAMBDA k -> bool : S1A(k)),
       B <- (LAMBDA k -> bool : S1C(p, q, k))},
    S1A_lemma {i <- j@p3}

END main
```

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-4239		

4. Title and Subtitle	5. Report Date
Formal Verification of a Fault Tolerant Clock Synchronization Algorithm	June 1989
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
John Rushby and Frieder von Henke	
	10. Work Unit No. 505-66-21-01
9. Performing Organization Name and Address SRI International 333 Ravenswood Avenue Menlo Park, CA 94025	11. Contract or Grant No. NAS1-17067
	13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Ricky W. Butler
Task 4 Final Report

16. Abstract

We describe a formal specification and mechanically-assisted verification of the interactive convergence clock synchronization algorithm of Lamport and Melliar-Smith. In the course of this work we discovered several technical flaws in the analysis given by Lamport and Melliar-Smith, even though their presentation is unusually precise and detailed. As far as we know these flaws were not detected by the "social process" of informal peer scrutiny to which the paper has been subjected since its publication. We discuss the flaws in the published proof and give a revised presentation of the analysis that not only corrects the flaws in the original, but is also more precise and, we believe, easier to follow. This informal presentation was derived directly from our formal specification and verification. Some of our corrections to the flaws in the original require slight modifications to the assumptions underlying the algorithm and to the constraints on its parameters, and thus change the external specifications of the algorithm. The formal analysis of the interactive convergence clock synchronization algorithm was performed using our Enhanced Hierarchical Development Methodology (EHDM) formal specification and verification environment. This application of EHDM provides a demonstration of some of the capabilities of the system.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Verification Clock Synchronization Design Proof Formal Verification	Unclassified - Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	232	A11

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171