Verifying the Interactive Convergence Clock Synchronization Algorithm Using the Boyer-Moore Theorem Prover

William D. Young

Technical Report 77 April 1992

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was sponsored in part at Computational Logic, Inc. by National Aeronautics and Space Administration Langley Research Center (NAS1-18878). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., NASA Langley Research Center or the U.S. Government.

1. Introduction

The application of formal methods to the analysis of computing systems promises to provide higher and higher levels of assurance as the sophistication of our tools and techniques increases. Improvements in tools and techniques come about as we pit the current state of the art against new and challenging problems. A promising area for the application of formal methods is in real-time and distributed computing. Some of the algorithms in this area are both subtle and important. Their proofs are an ideal testing ground for formal methods because they involve detailed and sophisticated reasoning that is challenging even for a competent human mathematician. We believe that formal methods are already demonstrating that they can make a genuine contribution toward the clarity and correctness of these algorithms [11, 2].

One important algorithm in this field is the Interactive Convergence Clock Synchronization Algorithm (ICCSA) of Lamport and Melliar-Smith [9]. This algorithm maintains approximate synchronization among a number of clocks even when the clocks begin running at slightly different times, run at slightly varying rates, and some percentage of them may be faulty. The presentation of Lamport and Melliar-Smith both develops the algorithm and states formally the assumptions and desired properties required to state and prove its correctness properties.

A mechanical verification of the algorithm using EHDM was performed by John Rushby and Friedrich von Henke and described in [11].[1] The EHDM effort resulted is a completely formal presentation of the algorithm and its proof, a presentation that is arguably somewhat clearer and more rigorous than the original published proof. Rushby and von Henke challenged users of proof systems other than EHDM as follows.

> 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. [W]e invite the developers and users of other verification systems to repeat the experiment described here. We suggest 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 artificial 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.

In response to this challenge and as part of an ongoing attempt to verify an implementation of the Interactive Convergence Clock Synchronization Algorithm, we decided to undertake a proof of the correctness of the algorithm using the Boyer-Moore theorem prover.

This paper describes our approach to proving the ICCSA using the Boyer-Moore prover. Since our proof follows closely that of Rushby and von Henke, we will not dwell on the details of the proof but assume that the reader is familiar with their quite cogent description of the EHDM version of the proof [11]. Instead we concentrate on the use of features of the Boyer-Moore logic and theorem prover that were especially helpful in the specification and proof and on the differences from the Rushby and von Henke version. We assume that the reader is somewhat familiar with the Boyer-Moore logic and theorem prover [3, 5].

This paper is organized as follows. The next section introduces briefly the Interactive Convergence Clock Synchronization Algorithm and the problem it is designed to solve. Sections 3 and 4 describe some interesting aspects of the specification and proof, respectively, and some of the more significant ways in which these differ from the EHDM version. Finally, section 5 contains some conclusions from this study.

---

[1] Rushby subsequently revised this proof, partly in response to our observations.

2. The Interactive Convergence Clock Synchronization Algorithm

A difficult problem facing designers of systems that achieve fault-tolerance via redundant processing capability is synchronizing the processors so that they deliver their results at approximately the ‘‘same time.’’ One solution to this problem is the Interactive Convergence Clock Synchronization Algorithm of Lamport and Melliar-Smith [9]. This algorithm maintains approximate synchronization among a number of clocks even when the clocks begin running at slightly different times, run at slightly varying rates, and some percentage of them may be faulty.

We desire an algorithm in which each processor periodically resynchronizes with all of the other processors in such a way that:

S1. all nonfaulty clocks have approximately the same value at any time; and,

S2. the adjustment to any clock during a synchronization period is bounded.

Proving that any algorithm achieves these two conditions is difficult because it requires accounting for a number of continually changing quantities.[2] Lamport and Melliar-Smith were able to prove that the ICCSA algorithm has these properties. Their proof is quite detailed involving approximate reasoning and neglect of various terms.

Conceptually the algorithm operates as follows. Each processor p_i maintains an offset or correction to its private (hardware) clock; the private clock value plus correction is the *adjusted clock value*. The correction is periodically updated by adding to it the mean of the differences between p_i's adjusted clock value and all other processors' adjusted clock values. Any processor p_j with adjusted value too divergent from p_i's is assumed to be faulty and a difference of 0 between p_i's and p_j's clocks is used in computing the mean.

Though the algorithm is conceptually quite simple, the statement of the correctness properties and their proof is complex. The correctness of the algorithm is stated in terms of certain quantities listed in figure 1 and others computed in terms of them. The proof shows that under certain conditions on the relationships among these parameters, this algorithm does maintain adequate synchronization among n processes with at most m faulty processes. Here *adequate synchronization* is defined in terms of formal statements of conditions S1 and S2 above. A completely formal description of the algorithm and its correctness conditions is given by Rushby and von Henke [11]. Our formalization follows fairly closely the Rushby and von Henke version and is given as the sequence of Boyer-Moore ‘‘events’’ in the appendix. In the following two sections we highlight features of the Boyer-Moore logic and prover that were particularly helpful in our specification with emphasis on the differences from the EHDM version.

3. Specifying the ICCSA in the Boyer-Moore Logic

Capturing formally the Interactive Convergence Clock Synchronization Algorithm within the Boyer-Moore logic was a challenge despite the fact that we had as a model a fully formal version in EHDM. There are difference in the languages that make translating from one to the other nontrivial. In particular, EHDM allows full first order quantification and uses higher-order functions in a manner that cannot be specified in the Boyer-Moore logic. However, recent additions to the logic—particularly `CONSTRAIN` and `DEFN-SK`—made our task much easier than it otherwise would have been.

---

[2] In fact, a proof of an implementation of the ICCSA algorithm was asserted to be ‘‘probably beyond the ability of any current mechanical verifier’’ [10]. However, there have been tremendous advances in the state of the art of automated reasoning in the nearly 10 years since this comment was made.

n number of clocks.

m number of faulty clocks.

R clock time between synchronizations.

S clock time to perform the synchronization algorithm.

δ maximum real time skew between any two good clocks.

δ_0 maximum initial real time skew between any two good clocks.

ε maximum real time clock read error.

ρ maximum clock drift rate.

Σ maximum correction permitted.

Figure 1: Some Quantities Required for Specifying the ICCSA

3.1 Computing with Rationals

The Boyer-Moore logic provides as ‘‘primitives’’ the data types of booleans, naturals, literal atoms, negative integers, and lists. Describing the ICCSA and proving its synchronization properties requires the manipulation of numerous *rational* quantities. The Boyer-Moore ‘‘shell’’ mechanism allows the user to add new recursively defined data types. The rationals have been added as a new shell and explored to some extent in some previous specification efforts. Until recently, however, there has not been a well thought out library of definitions and rewrite rules for rationals as have been developed for several other data types including naturals and integers. Recently Matt Wilding of CLI created a useful library for the rationals; this library provided a solid basis for our proof.

The rationals library is built on top of an earlier library for the integers.[3] Operations defined include equality, the various arithmetic operations on rationals, and the relational operator `RLESSP`. A number of useful rewrite rules are proved about these functions and included in the library. On top of the basis provided by Wilding’s library, we defined some additional operations required for the ICCSA specification, such as rational absolute value, operations coercing integers to rationals, and arithmetic operations taking both integer and rational arguments. Proving properties of these functions was usually quite easy because the underlying library was well thought-out. This contrasts with some earlier proof efforts [1, 7, 12] in which all theories had to be built ‘‘from scratch.’’

There are some quirks in dealing with rationals in the logic that are not present for most data types. In particular, there are an infinite number of representations for each rational number. This leads to the need to reduce all rationals to a canonical form before comparing them. All of the operations in Wilding’s rationals library leave rationals in reduced form.

Rational equality is defined in terms of these reduced forms:

```
(DEFN REQUAL (X Y)
  (EQUAL (REDUCE X) (REDUCE Y))).
```

However, since the prover has extensive built-in heuristics for `EQUAL`, but not for `REQUAL`, it was convenient to open up this definition whenever possible. This leads to a continual need to deal with terms of the form `(REDUCE X)`. Luckily, the rationals library contains an extensive collection of rewrites such as the following two

[3] A rational is represented as a pair of integers `(RATIONAL I J)` with appropriate constraints on the signs of i and j.

for the RPLUS function that eliminate most appearances of the REDUCE operator.

```
(PROVE-LEMMA RPLUS-REDUCE (REWRITE)
       (AND (EQUAL (RPLUS (REDUCE X) Y)
                   (RPLUS X Y))
            (EQUAL (RPLUS X (REDUCE Y))
                   (RPLUS X Y))))

(PROVE-LEMMA REDUCE-RPLUS (REWRITE)
       (EQUAL (REDUCE (RPLUS X Y))
              (RPLUS X Y)))
```

These ubiquitous REDUCE expressions caused one oddity in the specification. The library contains a number of rewrite rules such as:

```
(PROVE-LEMMA RPLUS-RZEROP (REWRITE)
       (IMPLIES (RZEROP X)
                (AND (EQUAL (RPLUS X Y) (REDUCE Y))
                     (EQUAL (RPLUS Y X) (REDUCE Y)))))
```

However, using this rewrite rule an expression such as

```
(EQUAL (RHO) (RPLUS (RHO) (RATIONAL 0 1)))
```

rewrites to

```
(EQUAL (RHO) (REDUCE (RHO)))
```

that is not provable unless (RHO) is known to be in reduced form, eg., if (RHO) is a constrained constant (see section 3.2 below). This led to the need to require that most constrained constants be in reduced form.

This rather odd but innocuous requirement could be avoided by consistently using REQUAL rather than EQUAL whenever referring to rational quantities. However, even with extensive theory development, heuristic reasoning support for REQUAL would not equal that available for EQUAL. An experimental facility supporting reasoning with congruence relations recently implemented by Bishop Brock of CLI might have alleviated some of this difficulty, but was not used here.

The utility of the rationals library is greatly enhanced by the addition of several useful *metafunctions*. Metafunctions [4] are user-defined term simplification routines that are proven to preserve the meaning (evaluation) of the term to which they are applied. For example, a function in the rationals library ‘‘cancels’’ complementary terms in a rationals RPLUS expression. Proving that this function preserves the meaning (value) of the term to which it is applied sanctions the installation of this code as an additional simplification routine within the prover. The code is installed automatically by the prover upon proof of the required theorems and replaces a potentially infinite collection of rewrite rules.

The collection of metafunctions within the rationals library greatly simplifies reasoning about RPLUS, RTIMES, and RLESSP expressions. We added an additional metalemma for the RLEQ (rationals less than or equal) function. This was not strictly necessary since RLEQ is defined in terms of REQUAL and RLESSP. However, to avoid the explosion of cases on theorems involving many RLEQ hypotheses, we decided to develop a theory for RLEQ on top of Wilding’s rationals library and leave RLEQ disabled (so that it would not be automatically opened up by the prover). We are not completely convinced of the wisdom of this decision.

3.2 Uses of `CONSTRAIN`

As we saw in section 2 above, the ICCSA is described in terms of a large number of integer and rational-valued parameters; these are conceptually global constants for purposes of the specification. There is a sizable collections of assumptions about the relative sizes of these quantities. In the Rushby and von Henke specification, these are given as EHDM axioms. Within the Boyer-Moore logic there are various options for how to introduce these constants into the specification:

- pass them as parameters to each function requiring them and add the assumptions as explicit hypotheses on each theorem requiring them;
- define each constant as a declared function of no arguments and add any required assumptions as axioms;
- use the Boyer-Moore `CONSTRAIN` [6] mechanism to introduce the constants as new function symbols and introduce the assumptions axiomatically within the `CONSTRAIN`.

The advantage of this final approach is that it guarantees that the introduced axioms are consistent without cluttering up definitions and theorems with a multitude of additional parameters and hypotheses. Moreover, using a `CONSTRAIN` event to introduce a new function symbol avoids the overspecification often occasioned by introducing functions via explicit definitions; only the required properties of the function need be specified.

A `CONSTRAIN` event introduces one or more function symbols along with axioms that they must satisfy. To guarantee the consistency of the axioms, the user must supply *witness* functions that satisfy the axioms. For example, we model the function $\Delta^{(i)}_{p,q}$ that computes the difference in clock values between processes p and q in period i, with the `CONSTRAIN` event:

```
(CONSTRAIN DELTA2-INTRO (REWRITE)
       (AND (RATIONALP (DELTA2 R P I))
            (EQUAL (DELTA2 P P I) (RATIONAL 0 1))
            (IMPLIES (NOT (NUMBERP I))
                     (EQUAL (DELTA2 R P I)
                            (DELTA2 R P 0)))
            (EQUAL (REDUCE (DELTA2 P Q I))
                   (DELTA2 P Q I)))
     ((DELTA2 (LAMBDA (R P I) (RATIONAL 0 1)))))
```

This asserts that the newly introduced function `DELTA2` is rational-valued, returns zero as the difference from a process's own clock value, always coerces it's third argument to a natural number, and returns a rational in reduced form.[4] We also supply a function that satisfies these axioms, namely the function of three arguments that always returns rational zero. A `CONSTRAIN` event is not accepted unless the axioms, appropriately instantiated with the witness functions, can be proved. This assures the consistency of the axioms by exhibiting a model.

Most of the constant parameters of our specification are introduced in a single large `CONSTRAIN` event `PARAMETERS-INTRO` given in Figure 2. It might have been better to introduce these via several different `CONSTRAIN` events. In that case, it could have been more difficult to find appropriate witness functions, however.

A very strong advantage of introducing the various parameters in this way is that their names and properties become ‘‘globally’’ visible. This allows us to give our theorems in a succinct form very close to those of the EHDM representation. Figure 3, for example, shows the same lemma in both its EHDM form[5] and in the Boyer-Moore logic.

---

[4]This is not a minimal set. The first axiom follows from the fourth one.

[5]Forms in [11] were pretty printed using a special facility described in that report; raw input to EHDM is much less elegant. The version here is in the prettified format.

```
(CONSTRAIN PARAMETERS-INTRO (REWRITE)
      ;; R and S
     (AND (RATIONALP (R))
          (RATIONALP (S))
          (RLESSP (RATIONAL 0 1) (R))                          ;; posR
          (RLESSP (RATIONAL 0 1) (S))                          ;; posS
          (RLEQ (RTIMES (RATIONAL 3 1) (S)) (R))               ;; C1
      ;; rho
          (RATIONALP (RHO))
          (RLEQ (RATIONAL 0 1)
                (RTIMES (RATIONAL 1 2) (RHO)))                 ;; rho_pos
          (RLESSP (RTIMES (RATIONAL 1 2) (RHO))                ;; rho_small
                  (RATIONAL 1 1))
      ;; other parameters
          (RATIONALP (EPSILON))
          (RATIONALP (DELTA))
          (RATIONALP (DELTA0))
          (RATIONALP (BIG-SIGMA))
          (RATIONALP (BIG-DELTA))
          (NUMBERP (N))
          (NOT (EQUAL (N) 0))                                  ;; C0_a
          (NUMBERP (M))                                        ;; C0_b
          (LESSP (M) (N))
          (RLESSP (RATIONAL 0 1) (BIG-DELTA))                  ;; C0_c
          (RLEQ (BIG-SIGMA) (S))                               ;; C2
          (RLEQ (BIG-DELTA) (BIG-SIGMA))                       ;; C3
          (RLEQ (RPLUS (DELTA)                                 ;; C4
                       (RPLUS (EPSILON)
                              (RTIMES (RATIONAL 1 2)
                                      (RTIMES (RHO) (S)))))
                (BIG-DELTA))
          (RLEQ (RPLUS (DELTA0) (RTIMES (RHO) (R)))            ;; C5
                (DELTA))
          (RLEQ                                                ;; C6
           (RPLUS
            (RTIMES (RATIONAL 2 1)
                    (RPLUS (EPSILON) (RTIMES (RHO) (S))))
            (RPLUS (RQUOTIENT-NAT
                    (RTIMES-NAT (TIMES 2 (M)) (BIG-DELTA))
                    (DIFFERENCE (N) (M)))
                   (RPLUS
                    (RQUOTIENT-NAT
                     (RTIMES-NAT (N) (RTIMES (RHO) (R)))
                     (DIFFERENCE (N) (M)))
                    (RPLUS (RTIMES (RHO) (BIG-DELTA))
                           (RQUOTIENT-NAT
                            (RTIMES-NAT (N) (RTIMES (RHO)
                                                    (BIG-SIGMA)))
                            (DIFFERENCE (N) (M)))))))
           (DELTA)))
     ((R (LAMBDA () (RATIONAL 3 1)))
      (S (LAMBDA () (RATIONAL 1 1)))
      (RHO (LAMBDA () (RATIONAL 0 1)))
      (EPSILON   (LAMBDA () (RATIONAL 0 1)))
      (DELTA     (LAMBDA () (RATIONAL 0 1)))
      (DELTA0    (LAMBDA () (RATIONAL 0 1)))
      (BIG-SIGMA (LAMBDA () (RATIONAL 1 2)))
      (BIG-DELTA (LAMBDA () (RATIONAL 1 2)))
      (N         (LAMBDA () 1))
      (M         (LAMBDA () 0))))
```

Figure 2: `CONSTRAIN` Introducing ICCSA Parameters

The EHDM Version:

lemma1def: Lemma
$\mathbf{S1C(}p, q, i\mathbf{)} \wedge \mathbf{S2(}p, i\mathbf{)} \wedge \mathbf{nonfaulty(}p, i \mathbf{+ 1)} \wedge \mathbf{nonfaulty(}q, i \mathbf{+ 1)} \supset |\Delta_{\mathbf{q}\ \mathbf{p}}^{\mathbf{(i)}}| < \Delta$

The Boyer-Moore Version:

```
(PROVE-LEMMA LEMMA1 (REWRITE)
      (IMPLIES (AND (S1C P Q I)
                    (S2 P I)
                    (NONFAULTY P (ADD1 I))
                    (NONFAULTY Q (ADD1 I)))
               (RLESSP (RABS (DELTA2 Q P I)) (BIG-DELTA))))
```

Figure 3: EHDM and Boyer-Moore Versions of the Same Lemma

3.3 DEFN-SK

Another relatively new feature of the logic that proved useful was the DEFN-SK facility [8] that allows the introduction of quantified expressions into the specification. Several important constructs in the Rushby and von Henke version were defined via quantification. Earlier versions of the Boyer-Moore logic could not express many of these conveniently. In particular, to prove an existential statement required exhibiting a witness constructively.

A DEFN-SK event allows the definition of an explicitly quantified term and the use of this term in other definitions and theorems. For example, the notion of a *good clock* (within the interval $[T_0..T_N]$) is defined by Rushby and von Henke as:[6]

goodclock: function[proc, clocktime, clocktime → bool] =
$(\lambda p, T_0, T_N :$
$(\forall\, T_1, T_2 :$
$T_0 \le T_1 \wedge T_0 \le T_2 \wedge T_1 \le T_N \wedge T_2 \le T_N$
$\supset |c_p(T_1) - c_p(T_2) - (T_1 - T_2)|$
$\le \rho/2 * |T_1 - T_2|))$

Our definition is given by the DEFN-SK event:

[6] Notice that this definition is from the revised specification. The first published version had ‘‘<’’ where the current version has ‘‘≤’’. This has the rather curious consequence that there are no good clocks in a system in which the parameter ρ that gives the maximum clock drift rate is zero. Intuitively, this means that if all clocks are perfect no clocks are good. This illustrates the need for extreme care in defining these functions.

```
(DEFN-SK+ GOOD-CLOCK (P LOW HIGH)
  (FORALL (T1 T2)
    (IMPLIES (AND (IN-INTERVAL T1 LOW HIGH)
                  (IN-INTERVAL T2 LOW HIGH))
             (RLEQ (RABS (RPLUS (CLOCK P T1)
                                (RPLUS (RNEG (CLOCK P T2))
                                       (RPLUS (RNEG T1) T2))))
                   (RTIMES (RATIONAL 1 2)
                           (RTIMES (RHO)
                                   (RABS (RPLUS T1 (RNEG T2)))))))))
```

A `DEFN-SK` event causes two axioms to be added to the database; these two axioms corresponding to the skolemization of the event in each ‘‘direction’’ and together allow us to use an instance of a quantified expression appearing in a hypothesis to a theorem and to prove an instance appearing as the conclusion. The macro version `DEFN-SK+` of the event also causes these axioms to be encapsulated and proved as rewrite rules. For `GOOD-CLOCK` these two theorems are shown in figure 4. See [8] for details on how these are generated and a proof of the soundness of the approach.

```
(PROVE-LEMMA GOOD-CLOCK-SUFF (REWRITE)
  (IMPLIES
     (IMPLIES
       (AND (IN-INTERVAL (T1 HIGH LOW P) LOW HIGH)
            (IN-INTERVAL (T2 HIGH LOW P)
                         LOW HIGH))
       (RLEQ (RABS (RPLUS (CLOCK P (T1 HIGH LOW P))
                          (RPLUS (RNEG (CLOCK P (T2 HIGH LOW P)))
                                 (RPLUS (RNEG (T1 HIGH LOW P))
                                        (T2 HIGH LOW P)))))
             (RTIMES (RATIONAL 1 2)
                     (RTIMES (RHO)
                             (RABS (RPLUS (T1 HIGH LOW P)
                                          (RNEG (T2 HIGH LOW P))))))))
    (GOOD-CLOCK P LOW HIGH)))

(PROVE-LEMMA GOOD-CLOCK-NECC (REWRITE)
  (IMPLIES
     (NOT (IMPLIES
            (AND (IN-INTERVAL T1 LOW HIGH)
                 (IN-INTERVAL T2 LOW HIGH))
            (RLEQ (RABS (RPLUS (CLOCK P T1)
                               (RPLUS (RNEG (CLOCK P T2))
                                      (RPLUS (RNEG T1) T2))))
                  (RTIMES (RATIONAL 1 2)
                          (RTIMES (RHO)
                                  (RABS (RPLUS T1 (RNEG T2))))))))
     (NOT (GOOD-CLOCK P LOW HIGH))))
```

Figure 4: Theorems Generated for a `DEFN-SK+` Event

Use of `DEFN-SK` allows us to define concepts involving quantifiers in a fashion that is very analogous to their EHDM counterparts. However, we did not always find this convenient. For example, Rushby and von Henke define SDEF as follows:

Sdef: **Axiom** $T \in S^{(i)} = (\exists\, \Pi\colon 0 \le \Pi \wedge \Pi \le R \wedge T = T^{(i)} + \Pi)$

A close analogue in the Boyer-Moore logic using DEFN-SK would be:

```
(DEFN-SK+ SDEF (TM I)
      (EXISTS PI
        (AND (RLEQ (RATIONAL 0 1) PI)
             (RLEQ PI (S))
             (EQUAL (REDUCE TM)
                    (RPLUS (TI I) (RPLUS (RDIFFERENCE (R) (S)) PI))))))
```

However, we found the following definition to be more convenient and to eliminate an unnecessary existential quantifier.

```
(DEFN IN-S (TM I)
      (IN-INTERVAL TM
                   (RDIFFERENCE (TI (ADD1 I)) (S))
                   (TI (ADD1 I))))
```

This illustrates that often the use of one style of definition is more ‘‘natural’’ in a given logic even when others styles are available. It is not surprising then that some of our definitions were quite different than the corresponding EHDM versions. However, we believe them to be equivalent in all relevant aspects. As an exercise, we proved the lemma that shows the equivalence of the definitions SDEF and IN-S.

```
(PROVE-LEMMA SDEF-IN-S-EQUIVALENCE ()
      (IFF (SDEF TM I) (IN-S TM I)))
```

Using CONSTRAINs and DEFN-SKs, we were able to write theorems that are textually very close to the EHDM versions in most cases. It is evident from the two versions of LEMMA1 listed in Figure 3 that, except for minor textual differences, there is very little difference in the presentation of theorems in the two logics. This was the rule rather than the exception for the lemmas required in our proof.

3.4 Avoiding Higher Order Functions

The ability within EHDM to define higher order functions is a definite benefit from the perspective of writing clear and elegant specifications. However, many of the uses of higher order functions can be avoided by careful use of facilities available within the Boyer-Moore logic. This was true of each of the uses of the EHDM higher order facilities in the ICCSA specification.

As an example, consider the MEAN function defined by Rushby and von Henke as follows:

$\oplus_{*1}^{*2}$ *3: function[nat, nat, function [nat $\rightarrow$ number] $\rightarrow$ number] =
$(\lambda i, j, F$: **if** $i \le j$ **then** $\Sigma_i^j\, F/(j + 1 - i)$ **else** 0 **end if**).

Notice that one parameter to this definition is a function F. From this definition, Rushby and von Henke prove a number of quite general lemmas.

There is not a similar facility within the Boyer-Moore logic though many of the advantages of such higher order definitions are available via other routes. For example, our version of the MEAN is defined as follows:

```
(DEFN RSUM (LST)
      (IF (NLISTP LST)
          (RATIONAL 0 1)
        (RPLUS (CAR LST) (RSUM (CDR LST)))))

(DEFN RMEAN (LST)
      (RQUOTIENT-NAT (RSUM LST) (LENGTH LST)))
```

Rather than parameterizing `RMEAN` with a function, we parameterize it with a list of elements returned by the function. This is conceptually equivalent and we can prove all of the nice properties of the EHDM version. Most of the interesting properties are really properties of `RSUM` rather than of `RMEAN`.

This style of trivial transformation is not the only way to deal with higher order functions and properties in the logic. An interpreter is available for the logic and permits reasoning about functions at the meta-level. Also, it is possible to ‘‘fake’’ higher-order properties in other ways. We have checked the proof, for example, that there is *no algorithm* that solves a certain version of the Byzantine General’s problem. [2] This is inherently a second order property.

4. Aspects of the Proof

4.1 Restraining the Prover

Our proof of ICCSA was somewhat atypical of most proofs using the Boyer-Moore prover. Rushby and von Henke had done much of the difficult work of finding a sequence of lemmas leading up to the proofs of the desired correctness theorems. Moreover, because of the way the EHDM prover operates, the collection of lemmas necessary for a given proof are displayed along with their specific instantiations. Given this information, constructing a formal proof is largely a matter of intelligent simplification and tautology checking.[7]

A proof in the Boyer-Moore theorem prover typically relies more on the prover’s heuristics to choose among previously proven lemmas and instantiate them correctly. However, the prover can be used in a more restrained fashion by disabling most functions and rewrite rules and using the prover as a simple proof checker. This is done by enabling only those lemmas known to be relevant and adding `USE` hints to specify particular instantiations of the variables in needed lemmas. This was the approach we followed in our proof of the ICCSA; most functions and lemmas were globally disabled. We also made use of an experimental feature for encapsulating the names of a group of events into a ‘‘theory’’ that can be enabled or disabled collectively. Figure 5 shows a particular lemma in our script that is an example of the use of `USE` hints, selective enabling, and theory enabling to obtain the proof.

4.2 Order of Steps in the Proof

The Boyer-Moore prover allows very little flexibility in the order of steps in a proof. Each function must be fully defined or constrained before it is used; each lemma must be proven before it can be used in proofs. For definitions this means that there is no genuine mutual recursion[8] For proofs it means that the proof is presented (though not necessarily discovered) in a very ‘‘bottom-up’’ style. This approach guarantees that there are no circularities in the proof.

EHDM does not impose such a limited ordering on the steps in the proof. To assure that there are no circularities in

[7] The EHDM proof of ICCSA used only the EHDM *ground prover*. [11]

[8] There is a standard way to gain the effects of mutual recursion by defining several ‘‘functions’’ within one and using a flag to distinguish among them. [5] Also, there is an available read macro for the prover that turns a list of mutually recursive definitions into an event of this type.

```
(LEMMA SUBLEMMA-A (REWRITE)
       (IMPLIES (AND (NONFAULTY P I)
                     (NONFAULTY Q I)
                     (IN-R TM I))
                (RLEQ (SKEW P Q TM I)
                        (RPLUS (SKEW P Q (TI I) I)
                               (RTIMES (RHO) (R)))))
      ((USE (REARRANGE-ALT (X (C P I TM))
                           (Y (C Q I TM))
                           (U (C P I (TI I)))
                           (V (RPLUS TM (RNEG (TI I))))
                           (W (C Q I (TI I))))
            (LEMMA2D (PI (RPLUS TM (RNEG (TI I)))))
            (LEMMA2D (P Q) (PI (RPLUS TM (RNEG (TI I))))))
       (ENABLE-THEORY REDUCTIONS)
       (ENABLE SKEW RDIFFERENCE RNEG-RPLUS C-REDUCE TI-NEXT RABS-POSITIVE2
               RPLUS-RLEQ-REWRITE RPLUS-RLEQ-REWRITE2 RLEQ-RTIMES-HACK
               IN-R IN-INTERVAL RHO-RLEQ0 RLEQ-TRANSITIVE RLEQ-RPLUS-HACK3
               RLEQ-HALF-RPLUS RLEQ-RTIMES-HACK RPLUS-RLEQ-REWRITE)))
```

Figure 5: Proof of `SUBLEMMA-A` Showing `USE` Hints

the resulting proof, a tool called the EHDM Proof Chain Analyzer is run over the final proof and checks for circularities. In the proof of ICCSA there is a circularity in the proof of the main theorem `THEOREM1`. This is explained as follows:

> 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. [11]

Unfortunately, determining whether such a circularity is apparent or real requires a fairly deep understanding of the proof. The Boyer-Moore approach does not allow even an apparent circularity but the cost is a much more regimented approach to proof presentation.

As an interesting aside, just as Rushby and von Henke had to deal with the structure of the inductive proof of `THEOREM1` in EHDM, we had to confront the same issue in the Boyer-Moore system. We could approach it either by defining an appropriate induction schema to make available the required inductive hypotheses (the typical approach in the Boyer-Moore system) or by using another approach altogether. Defining an appropriate induction schema would have been difficult because the inductive hypothesis was really required in the proof of a large subsidiary lemma `CULMINATION`. We would have needed to prove `THEOREM1` and `CULMINATION` simultaneously by packaging them into one lemma. This trick is used often in the Boyer-Moore prover. Our solution was again somewhat atypical and illustrates a clever (we think) use of `DEFN-SK`.

`THEOREM1` has form:

```
(PROVE-LEMMA THEOREM1 (REWRITE)
      (IMPLIES (S1A I) (S1C P Q I)))
```

We introduced the `DEFN-SK` event below to define the structure of the theorem:

```
(DEFN-SK THEOREM1-ONE-STEP (I)
  (FORALL (P Q)
       (IMPLIES (S1A I)
                (S1C P Q I))))
```

Notice that this is parameterized by `i`. Asserting `(THEOREM1-ONE-STEP I)` is equivalent to asserting that `THEOREM1` holds through period `i`. Wherever the EHDM approach used `THEOREM1` in the proof, we simply asserted `THEOREM1-ONE-STEP` as an additional hypothesis on the lemma, as in `CULMINATION` below:

```
(PROVE-LEMMA CULMINATION (REWRITE)
  (IMPLIES
    (AND (S1A (ADD1 I))
         (S1C P Q I)
         (NONFAULTY P (ADD1 I))
         (NONFAULTY Q (ADD1 I))
         (IN-R TM (ADD1 I))
         (THEOREM1-ONE-STEP I))
    (RLEQ (SKEW P Q TM (ADD1 I))
          (RPLUS
           (RQUOTIENT-NAT
            (RPLUS
             (RTIMES-NAT (M)
                         (RPLUS (DELTA)
                                (RTIMES (RATIONAL 2 1) (BIG-DELTA))))
             (RTIMES-NAT (DIFFERENCE (N) (M))
                         (RTIMES (RATIONAL 2 1)
                                 (RPLUS (EPSILON)
                                        (RPLUS (RTIMES (RHO) (S))
                                               (RTIMES (RATIONAL 1 2)
                                                       (RTIMES (RHO) (BIG-DELTA))))))))
            (N))
           (RPLUS (RTIMES (RHO) (R))
                  (RTIMES (RHO) (BIG-SIGMA))))))).
```

This made available, in the proof of `CULMINATION` exactly the instance of `THEOREM1` required in that proof. We then used `CULMINATION` in the proof (by induction on `i`) of the lemma:

```
(PROVE-LEMMA THEOREM1-VERSION1 (REWRITE)
       (THEOREM1-ONE-STEP I).
```

`CULMINATION` is used in the proof of the induction step, its `THEOREM1-ON-STEP` hypothesis being relieved by the inductive hypothesis. `THEOREM1` follows straightforwardly from `THEOREM1-VERSION1`. This approach indicates again the utility of `DEFN-SK` in adding clarity and proof power.

4.3 Proof Encapsulation

The Boyer-Moore logic has no convenient way of structuring a specification and proof into a collection of ‘‘modules.’’ This is largely dictated by the requirement that the specification and proof be presented in a very ‘‘bottom up’’ fashion. A collection of related units may be grouped together in the script, but there is no formal mechanism within the logic of encapsulating them into a module or structure of any sort. This is not often a problem but makes a large script somewhat harder to browse effectively.

In contrast, EHDM has a simple but useful structuring mechanism. Related units are grouped into modules. Modules implement a style of information hiding by making visible only certain declarations within an `EXPORTING` section. Modules gain access to one another by including a `USING` section.

4.4 Syntax

The Boyer-Moore logic is sometimes criticized for its Lisp-like syntax. This syntax has the advantage of being uniquely-readable (unambiguous) and very easily parsed. It has the disadvantage of being different from traditional mathematical syntax. Several papers have described proofs in the Boyer-Moore logic using a more traditional syntax; however, these may mislead a prospective user of the theorem prover. We feel that the small effort of learning a new syntax is well rewarded by gaining access to a powerful proof tool.

EHDM has reaped the benefits of both a readily parsable syntax and a more familiar ‘‘display’’ syntax by implementing a table driven translator from standard EHDM syntax into a LaTeX format. This gives a nice customizable syntax for presentation that Rushby and von Henke claim ‘‘enabled us to do most of our work using compact and familiar notation and thereby contributed greatly to our productivity’’ [11]. We believe that this overstates the value of this ‘‘compact and familiar notation.’’

Our experience with attempting a similar translator for the Boyer-Moore logic is that it is generally counterproductive to try to integrate such a translator with a theorem prover that uses a different syntax for its internal representation, proof diagnostics, and output script. If the translation could be entirely transparent to the user, there would be no difficulty. However, users of mechanical proof tools often need to be aware of the details of the internal representation of rewrite rules, the particular transformations on terms that they effect, and other things that are most efficiently expressed in a syntax that is close to that used by the machine. When this is no longer true, then syntax will not be an issue. Until then, we feel that the need to continually deal with two different forms is confusing and largely unnecessary. Nevertheless, we have recently developed a simple facility for *printing* Boyer-Moore events in a more familiar infix notation. It is not well integrated with the prover and does not serve as an input facility to the prover. It does allow publishing Boyer-Moore forms in a syntax which may be more familiar to typical audience.

Another problem of the EHDM translator is that the notation is *not* always compact and familiar. For example, we found the expression

$$\rho * \Delta \times n\text{-}m / n,$$

(that appears in a number of lemmas) to be impossibly confusing until we realized that the term *n-m* is treated as though it were parenthesized. Here the apparent familiarity of the syntax is detrimental because the expected precedence rules are not observed, with the result that the expression is unnecessarily confusing. This is probably a simple flaw in the translator table. But it points up the difficulty of having the correctness of a published proof rely not only on the prover and proof chain analyzer, but also on another tool that translates from one notation to another in a moderately complex fashion.

5. Conclusions

There are a number of other differences between the Boyer-Moore and EHDM versions of the ICCSA proofs that could be covered in a more detailed comparison of the two versions.

We believe that the exercise of specifying and proving the ICCSA using the Boyer-Moore prover was useful in several ways.

- It exercised and further displayed the value of a number of the newer features of the Boyer-Moore logic and their support in the theorem prover. These features include the `CONSTRAIN` and `DEFN-SK` events.
- It provided the basis for a comparison with the EHDM system and a style of proof possible within that system. This aspect will lead to a joint paper comparing the two systems on this problem.
- It provided a verified specification of the Interactive Convergence Clock Synchronization algorithm as

a basis for possible future work building toward a verified implementation.

We believe that two important goals of proof are to increase one's understanding and intuition about the content and significance of a theorem, and to provide a convincing argument that it is, in fact, valid. Mechanically supported proofs like those in EHDM and ours contribute to both of these goals. We understand this quite subtle algorithm and the reason it works much better for the effort. Moreover, our success in convincing a congenitally skeptical mechanical proof checker of the validity of the correctness theorems practically guarantees that we have eliminated any errors that the much touted ‘‘social process’’ might overlook. Such confidence is particularly comforting in domains such as this where a well-developed intuition is difficult to cultivate; the theorem prover is not subject to being misled by the urgings of a misguided or ill-informed intuition.

References

1. W. R. Bevier. "Kit and the Short Stack". *Journal of Automated Reasoning 5*, 4 (December 1989), 519-530.

2. W. R. Bevier and W. D. Young. "Machine Checked Proofs of the Design of a Fault-Tolerant Circuit". To appear *Formal Aspects of Computing*, 1992.

3. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

4. R. S. Boyer and J S. Moore. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

5. R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

6. R.S. Boyer, D. Goldschlag, M. Kaufmann, J S. Moore. Functional Instantiation in First Order Logic. Proceedings of the 1989 Workshop on Programming Logic, May, 1989.

7. W. A. Hunt, Jr. "Microprocessor Design Verification". *Journal of Automated Reasoning 5*, 4 (December 1989), 429-460.

8. M. Kaufmann. "An Extension of the Boyer-Moore Theorem Prover to Handle First-Order Quantifiers". To appear in *Journal of Automated Reasoning*, 1992.

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

10. NASA Conference Publication 2377. Peer Review of a Formal Verification/Design Proof Methodology. NASA, July, 1983.

11. J. Rushby and F. von Henke. Formal Verification of a Fault-Tolerant Clock Synchronization Algorithm. NASA CR-4239, June 1989.

12. W. D. Young. "A Mechanically Verified Code Generator". *Journal of Automated Reasoning 5*, 4 (December 1989), 493-518.

Appendix
The ICCSA Event List

This appendix contains the Boyer-Moore event list representing the specification and proof of the Interactive Convergence Clock Synchronization Algorithm. It does *not* contain the entire proof since it is built ‘‘on top of’’ a standard library of integer facts. For brevity we have also not included the collection of definitions and lemmas defining the rationals library on which our proof is constructed. The complete script is available on request.

`LEMMA` events are macro expanded into a `PROVE-LEMMA` followed by a `DISABLE`.

;; LENGTH

```
(defn length (x)
  (if (nlistp x)
      0
    (add1 (length (cdr x)))))

(prove-lemma length-append (rewrite)
      (equal (length (append x y))
             (plus (length x) (length y))))

(lemma length-0 (rewrite)
       (equal (equal (length x) 0)
              (nlistp x)))
```

;; PLISTP

```
(defn plistp (x)
  (if (nlistp x)
      (equal x nil)
    (plistp (cdr x))))

(defn plist (x)
  (if (nlistp x)
      nil
    (cons (car x) (plist (cdr x)))))
```

;; FIRSTN and RESTN

```
(defn firstn (lst n)
  (if (zerop n)
      nil
    (cons (car lst)
          (firstn (cdr lst) (sub1 n)))))

(lemma firstn-n (rewrite)
       (implies (equal n (length lst))
                (equal (firstn lst n)
                       (plist lst))))

(lemma firstn-append-lessp (rewrite)
       (implies (leq m (length lst))
                (equal (firstn (append lst lst2) m)
                       (firstn lst m))))

(defn restn (lst n)
  (if (zerop n)
      lst
    (restn (cdr lst) (sub1 n))))

(lemma restn-n (rewrite)
       (implies (and (plistp lst)
                     (equal n (length lst)))
                (equal (restn lst n) nil))
      ((enable restn)))
```

```
(lemma restn-1 (rewrite)
       (implies (lessp m 1)
                (equal (restn (list x) m)
                       (list x)))
      ((enable restn)))

(lemma restn-append (rewrite)
       (implies (leq n (length lst1))
                (equal (restn (append lst1 lst2) n)
                       (append (restn lst1 n) lst2)))
      ((enable restn)))

(lemma firstn-append-restn (rewrite)
       (implies (leq m (length lst))
                (equal (append (firstn lst m)
                               (restn lst m))
                       lst)))
```

;; RATIONALS WITH NATURALS

```
(defn rinverse-nat (n)
      (reduce (rational 1 (fix n))))

(lemma reduce-rinverse-nat (rewrite)
      (equal (reduce (rinverse-nat n))
             (rinverse-nat n))
     ((enable reduce-reduce)))

(defn rtimes-nat (i r)
      (rtimes (rational (fix i) 1) r))

(defn rtimes-nat2 (r i)
      (rtimes r (rational (fix i) 1)))

(lemma reduce-rtimes-nat (rewrite)
       (and (equal (reduce (rtimes-nat x y))
                   (rtimes-nat x y))
            (equal (reduce (rtimes-nat2 x y))
                   (rtimes-nat2 x y)))
      ((enable rtimes-nat rtimes-nat2)
       (enable-theory reductions)))

(lemma rtimes-nat-rtimes-nat2 (rewrite)
       (equal (rtimes-nat2 x y)
              (rtimes-nat y x))
      ((enable rtimes-nat rtimes-nat2 commutativity-of-rtimes)))

(lemma rneg-rtimes-nat (rewrite)
       (equal (rneg (rtimes-nat i r))
              (rtimes-nat i (rneg r)))
      ((enable rtimes-nat rneg-rtimes)
       (disable correctness-of-cancel-rneg-terms-from-equality)))

(defn rquotient-nat (r i)
      (rtimes r (rinverse-nat i)))

(lemma rneg-rquotient-nat (rewrite)
       (equal (rneg (rquotient-nat r n))
              (rquotient-nat (rneg r) n))
      ((enable rquotient-nat rneg-rtimes
               rneg-rtimes2)))

(disable rinverse-nat)
(disable rtimes-nat)
(disable rtimes-nat2)
(disable rquotient-nat)

(lemma rtimes-nat-add1 (rewrite)
      (equal (rtimes-nat (add1 i) r)
             (rplus r (rtimes-nat i r)))
     ((enable rtimes-nat rtimes-add1)))
```

```
(lemma rtimes-nat-zerop (rewrite)
      (implies (zerop i)
               (equal (rtimes-nat i r)
                      (rational 0 1)))
      ((enable rtimes-nat rzerop-rtimes)))

(lemma rinverse-nat-positive (rewrite)
       (rleq (rational 0 1)
             (rinverse-nat n))
      ((enable rinverse-nat numberp-inverse-nonnegative
               rleq-reduce)))

(lemma rabs-rinverse-nat (rewrite)
      (equal (rabs (rinverse-nat n))
             (rinverse-nat n))
      ((enable rabs-positive2 rinverse-nat-positive
               reduce-rinverse-nat)))

(lemma rquotient-nat-rtimes-nat (rewrite)
       (implies (not (zerop n))
                (equal (rquotient-nat (rtimes-nat n x) n)
                       (reduce x)))
       ((enable rtimes-inverse rtimes-nat rquotient-nat
                rinverse-nat nzerop-denominator-reduce)))

(lemma rtimes-nat-rquotient-nat (rewrite)
       (implies (not (zerop n))
                (equal (rtimes-nat n (rquotient-nat x n))
                       (reduce x)))
       ((enable rtimes-inverse rtimes-nat rquotient-nat rinverse-nat
                nzerop-denominator-reduce commutativity-of-rtimes
                commutativity2-of-rtimes rationalize-invert rtimes-1)))

(lemma rquotient-nat-rplus (rewrite)
       (equal (rquotient-nat (rplus x y) n)
              (rplus (rquotient-nat x n)
                     (rquotient-nat y n)))
      ((enable rquotient-nat rtimes-rplus-right-distributivity)))

(lemma div-mon2 (rewrite)
       (implies (and (rleq x y)
                     (not (zerop z)))
                (rleq (rquotient-nat x z)
                      (rquotient-nat y z)))
     ((enable rquotient-nat rtimes-right-cancellation rinverse-nat
              nzerop-denominator-reduce reciprocal-positive)))

(lemma rationalize-rleq2 (rewrite)
       (implies (not (zerop n))
                (rlessp (rational 0 1) (rational n 1)))
      ((enable rleq rlessp fix-rational ilessp rationalp
               integerp)))

(lemma rtimes-nat2-positive-preserves-rleq (rewrite)
       (implies (and (not (zerop n))
                     (rleq (rtimes-nat2 x n)
                           (rtimes-nat2 y n)))
                (rleq x y))
       ((enable rtimes-right-cancellation rtimes-nat2 rationalize-rleq2)))
```

;; RSUM and RMEAN

```
(defn rsum (lst)
      (if (nlistp lst)
          (rational 0 1)
        (rplus (car lst)
               (rsum (cdr lst)))))

(disable rsum)

(lemma reduce-rsum (rewrite)
       (equal (reduce (rsum lst))
              (rsum lst))
       ((enable reduce-rplus rsum)))
```

```
(lemma rplus-rsum (rewrite)
       (equal (rplus (rsum lst1) (rsum lst2))
              (rsum (append lst1 lst2)))
      ((enable rsum rplus-rzerop reduce-rsum
               associativity-of-rplus)))

(lemma rsum-append (rewrite)
       (equal (rsum (append lst1 lst2))
              (rplus (rsum lst1) (rsum lst2)))
      ((enable associativity-of-rplus rsum rplus-rzerop
               reduce-rsum)))

(defn rmean (lst)
      (rquotient-nat (rsum lst) (length lst)))

(defn all-rlessp (lst x)
      (if (nlistp lst)
          t
        (and (rlessp (car lst) x)
             (all-rlessp (cdr lst) x))))

(lemma all-rlessp-append (rewrite)
       (equal (all-rlessp (append x y) z)
              (and (all-rlessp x z)
                   (all-rlessp y z)))
      ((enable all-rlessp)))

(lemma all-rlessp-rleq-transitive (rewrite)
       (implies (and (all-rlessp lst x)
                     (rleq x y))
                (all-rlessp lst y))
       ((enable rlessp-rleq-transitivity)))

(lemma sum-bound (rewrite)
       (implies (and (listp lst)
                     (all-rlessp lst x))
                (rlessp (rsum lst)
                        (rtimes-nat (length lst) x)))
      ((enable all-rlessp rsum rtimes-nat-zerop
               rtimes-nat-add1 rlessp-rleq rplus-rzerop
               rlessp-reduce rlessp-rplus-pair)))

(lemma nzerop-inverse-positive (rewrite)
       (implies (not (zerop n))
                (RLESSP (RATIONAL 0 1)
                        (RATIONAL 1 n)))
      ((enable rlessp rationalp fix-rational ilessp)))

(lemma mean-bound (rewrite)
      ;; if all of the elements in the list are less than x,
      ;; then the mean of the list is less than x
      (implies (and (listp lst)
                    (all-rlessp lst x))
               (rlessp (rmean lst) x))
      ((use (sum-bound))
        (enable rlessp-invert-rtimes rinverse-numberp-inverse
                  nzerop-inverse-positive length-0 rtimes-nat
                  nzerop-denominator-reduce rquotient-nat rinverse-nat)))

;; MAP-RABS

(defn map-rabs (lst)
      (if (nlistp lst)
          nil
        (cons (rabs (car lst))
              (map-rabs (cdr lst)))))

(lemma length-map-rabs (rewrite)
      (equal (length (map-rabs lst))
             (length lst)))

(lemma map-rabs-append (rewrite)
       (equal (map-rabs (append x y))
              (append (map-rabs x) (map-rabs y))))
```

```
(lemma plistp-map-rabs (rewrite)
       (plistp (map-rabs lst))
      ((enable map-rabs plistp)))

(lemma plist-map-rabs (rewrite)
       (equal (plist (map-rabs x))
              (map-rabs x))
      ((enable map-rabs plist)))

(lemma rabs-rsum-map-rabs (rewrite)
      (rleq (rabs (rsum lst))
            (rsum (map-rabs lst)))
     ((enable rsum rabs-rplus-hack)))

(lemma abs-mean (rewrite)
      ;; the abs of the mean is leq the mean of the absolute values
     (rleq (rabs (rmean lst))
           (rmean (map-rabs lst)))
    ((enable rabs-rtimes rtimes-rleq2 rabs-rsum-map-rabs
             rinverse-nat-positive rquotient-nat rzerop-rtimes
             rabs-rinverse-nat length-map-rabs)))

(lemma listp-map-rabs (rewrite)
      (equal (listp (map-rabs x))
             (listp x)))

;; REARRANGE LEMMAS

(lemma rearrange1 (rewrite)
       (equal (rdifference x y)
              (rplus (rdifference x (rplus u v))
                     (rplus (rdifference (rplus w z) y)
                            (rdifference (rplus u v) (rplus w z)))))
      ((enable rdifference rneg-rplus associativity-of-rplus reduce-rneg)))

(lemma rabs-negation-equality-hack (rewrite)
       (equal (rabs (rplus y (rplus (rneg w) (rneg z))))
              (rabs (rplus w (rplus z (rneg y)))))
       ((use (rabs-rneg (x (rplus y (rplus (rneg w) (rneg z))))))
        (enable commutativity-of-rplus commutativity2-of-rplus
                rplus-reduce rneg-rneg rneg-rplus)))

(lemma rearrange2-transitivity (rewrite)
       (implies
        (rleq (rabs (rplus x (rneg y)))
              (rplus (rabs (rplus x (rplus (rneg u) (rneg v))))
                     (rplus (rabs (rplus u
                                         (rplus v (rplus (rneg w) (rneg z)))))
                            (rabs (rplus w (rplus z (rneg y)))))))
        (rleq (rabs (rplus x (rneg y)))
              (rplus (rabs (rplus x (rplus (rneg u) (rneg v))))
                     (rplus (rabs (rplus y (rplus (rneg w) (rneg z))))
                            (rabs (rplus u
                                         (rplus v
                                                (rplus (rneg w) (rneg z)))))))))
       ((use (rleq-transitive
              (x (rabs (rplus x (rneg y))))
              (y (rplus (rabs (rplus x (rplus (rneg u) (rneg v))))
                        (rplus (rabs (rplus u (rplus v (rplus (rneg w) (rneg z)))))
                               (rabs (rplus w (rplus z (rneg y)))))))
              (z (rplus (rabs (rplus x (rplus (rneg u) (rneg v))))
                        (rplus (rabs (rplus y (rplus (rneg w) (rneg z))))
                               (rabs (rplus u (rplus v (rplus (rneg w) (rneg z))))))))))
        (enable rabs-negation-equality-hack
                rleq-reflexive commutativity2-of-rplus
                commutativity-of-rplus rleq-reduce rplus-cancel)))

(lemma rearrange2 (rewrite)
       (rleq (rabs (rplus (rdifference x (rplus u v))
                          (rplus (rdifference (rplus w z) y)
                                 (rdifference (rplus u v) (rplus w z)))))
             (rplus (rabs (rdifference x (rplus u v)))
                    (rplus (rabs (rdifference y (rplus w z)))
                           (rabs (rplus u (rdifference v (rplus w z)))))))
```

```
       ((use (rabs-rplus-rleq2 (x (rdifference x (rplus u v)))
                      (y (rdifference (rplus u v) (rplus w z)))
                      (z (rdifference (rplus w z) y)))
             (rabs-rneg (x (rdifference (rplus w z) y))))
        (enable rplus-reduce rneg-rneg rdifference
                reduce-rneg associativity-of-rplus
                rneg-rplus rdifference rearrange2-transitivity)))

(lemma rearrange (rewrite)
       (rleq (rabs (rdifference x y))
             (rplus (rabs (rdifference x (rplus u v)))
                    (rplus (rabs (rdifference y (rplus w z)))
                           (rabs (rplus u (rdifference v (rplus w z)))))))
       ((use (rearrange1) (rearrange2))))

(lemma rearrange-alt (rewrite)
       (rleq (rabs (rplus x (rneg y)))
             (rplus (rabs (rplus x (rneg (rplus u v))))
                    (rplus (rabs (rplus u (rneg w)))
                           (rabs (rplus y (rneg (rplus w v)))))))

      ((use (rearrange (z v)))
       (enable rplus-reduce rneg-rneg reduce-rneg
               associativity-of-rplus rneg-rplus rdifference commutativity-of-rplus)))

(lemma rearrange3 (rewrite)
       (rleq (rabs (rdifference x y))
             (rplus (rabs (rdifference u y))
                    (rplus (rabs (rdifference v x))
                           (rplus (rabs (rdifference v w))
                                  (rabs (rdifference u w))))))

      ((use (rabs-rplus-rleq3 (x (rplus u (rneg y)))
                     (y (rplus x (rneg v)))
                     (z (rplus v (rneg w)))
                     (w (rplus w (rneg u))))
            (rleq-transitive
             (x (rabs (rplus x (rneg y))))
             (y (rabs (rplus (rplus u (rneg y))
                             (rplus (rplus x (rneg v))
                                    (rplus (rplus v (rneg w))
                                           (rplus w (rneg u)))))))
             (z (rplus (rabs (rplus u (rneg y)))
                       (rplus (rabs (rplus v (rneg x)))
                              (rplus (rabs (rplus v (rneg w)))
                                     (rabs (rplus u (rneg w)))))))))
       (enable rplus-reduce rneg-rneg reduce-rneg
               rabs-rdifference associativity-of-rplus rneg-rplus
               rdifference commutativity-of-rplus rleq-reflexive)))

(lemma rearrange4 (rewrite)
       (rleq (rabs (rdifference (rplus a x) (rplus b y)))
             (rplus (rabs (rdifference a b))
                    (rplus (rabs x) (rabs y))))
      ((use (rabs-rplus-rleq2 (x (rplus a (rneg b))) (y x) (z (rneg y))))
       (enable rdifference commutativity-of-rplus commutativity2-of-rplus
               associativity-of-rplus rabs-rneg rneg-rplus)))
```

;; REARRANGE-DELTA from module JUGGLE

```
(lemma rearrange-delta-step1 nil
       (implies (and (not (zerop i))
                     (rleq (rplus x (rplus y (rplus z (rplus w v)))) d))
                (rleq (rplus (rtimes-nat2 x i)
                             (rplus (rtimes-nat2 y i)
                                    (rplus (rtimes-nat2 z i)
                                           (rplus (rtimes-nat2 w i)
                                                  (rtimes-nat2 v i)))))
                      (rtimes-nat2 d i)))
       ((enable rleq requal rtimes-nat2 rtimes-rplus-right-factorization
                 rlessp-antisymmetric rlessp-trichotomy rationalized-non-zerop-rlessp)
        (enable-theory reductions)))
```

```
(lemma rearrange-delta-step2 nil
       (implies (and (lessp m n)
                     (rleq x (rtimes-nat2 d (difference n m))))
                (rleq (rplus (rtimes-nat2 d m) x) (rtimes-nat2 d n)))
      ((enable rtimes-nat2 rzerop-rtimes rplus-rzerop rleq-reduce
               rtimes-distributes-over-plus rplus-cancel rtimes-rzerop)))

(lemma rearrange-delta-step3 nil
       (implies (and (not (zerop i))
                     (rleq (rplus x (rplus y (rplus z (rplus w (rplus v u)))))
                           (rtimes-nat2 d i)))
                (rleq (rplus (rquotient-nat x i)
                             (rplus (rquotient-nat y i)
                                    (rplus (rquotient-nat z i)
                                           (rplus (rquotient-nat w i)
                                                  (rplus (rquotient-nat v i)
                                                         (rquotient-nat u i))))))
                      d))
       ((enable rtimes-nat2 rzerop-rtimes rplus-rzerop rleq-reduce
               rquotient-nat rtimes-rplus-right-factorization rplus-cancel
               rtimes-rzerop rinverse-nat nzerop-denominator-reduce
               rtimes-multiply-by-rinverse rinverse-numberp-inverse
               nzerop-inverse-positive)))

(lemma rearrange-delta-step4 (rewrite)
       (equal (rplus (rquotient-nat x n)
                     (rplus (rquotient-nat y n)
                            (rplus (rquotient-nat z n)
                                   (rplus w (rplus (rquotient-nat u n) v)))))
              (rplus (rquotient-nat (rplus x (rplus y (rplus z u))) n)
                     (rplus w v)))
      ((enable associativity-of-rplus commutativity-of-rplus reduce-rtimes
               commutativity2-of-rplus rquotient-nat rinverse-nat
               rtimes-rplus-right-factorization)))

(lemma rearrange-delta-step5 (rewrite)
       (equal (rplus (rtimes-nat m d)
                     (rplus (rtimes-nat y (rtimes (rational 2 1) (rplus e z)))
                            (rplus (rtimes-nat (plus m m) w) (rtimes-nat y x))))
              (rplus (rtimes-nat m (rplus d (rtimes (rational 2 1) w)))
                     (rtimes-nat y
                                 (rtimes (rational 2 1)
                                         (rplus e (rplus z (rtimes (rational 1 2) x)))))))
        ((enable-theory reductions)
         (enable rtimes-nat associativity-of-rplus
                       rtimes-distributes-over-rplus commutativity-of-rtimes
                       commutativity2-of-rtimes half3
                       commutativity-of-rplus commutativity2-of-rplus
                       rtimes-distributes-over-plus rtimes2-expand)))

(lemma rearrange-delta (rewrite)
       (implies (and (lessp m n)
                     (numberp m)
                     (rleq (rplus (rtimes (rational 2 1) (rplus epsilon (rtimes rho s)))
                                  (rplus (rquotient-nat (rtimes-nat (times 2 m) big-delta)
                                                        (difference n m))
                                         (rplus (rquotient-nat (rtimes-nat n (rtimes rho r))
                                                               (difference n m))
                                                (rplus (rtimes rho big-delta)
                                                       (rquotient-nat
                                                          (rtimes-nat n (rtimes rho big-sigma))
                                                          (difference n m))))))
                           delta))
```

```
              (rleq (rplus (rquotient-nat
                             (rplus
                              (rtimes-nat m
                                          (rplus delta
                                                 (rtimes (rational 2 1) big-delta)))
                              (rtimes-nat
                               (difference n m)
                               (rtimes (rational 2 1)
                                       (rplus epsilon
                                              (rplus (rtimes rho s)
                                                     (rtimes (rational 1 2)
                                                             (rtimes rho big-delta)))))))
                             n)
                            (rplus (rtimes rho r)
                                   (rtimes rho big-sigma)))
                    delta))
  ((use (rearrange-delta-step1
         (d delta)
         (i (difference n m))
         (x (rtimes (rational 2 1)
                    (rplus epsilon (rtimes rho s))))
         (y (rquotient-nat (rtimes-nat (times 2 m) big-delta)
                           (difference n m)))
         (z (rquotient-nat (rtimes-nat n (rtimes rho r))
                           (difference n m)))
         (w (rtimes rho big-delta))
         (v (rquotient-nat (rtimes-nat n (rtimes rho big-sigma))
                           (difference n m))))
        (rearrange-delta-step2
         (x
          (rplus
           (rtimes-nat2 (rtimes (rational 2 1)
                                (rplus epsilon (rtimes rho s)))
                        (difference n m))
           (rplus (reduce (rtimes-nat (times 2 m) big-delta))
                  (rplus (reduce (rtimes-nat n (rtimes rho r)))
                         (rplus (rtimes-nat2 (rtimes rho big-delta)
                                             (difference n m))
                                (reduce (rtimes-nat n
                                                    (rtimes rho big-sigma))))))))
         (d delta))
        (rearrange-delta-step3
         (i n)
         (d delta)
         (x (rtimes-nat2 delta m))
         (y (rtimes-nat2 (rtimes (rational 2 1)
                                 (rplus epsilon (rtimes rho s)))
                         (difference n m)))
         (z (rtimes-nat (times 2 m) big-delta))
         (w (rtimes-nat n (rtimes rho r)))
         (v (rtimes-nat2 (rtimes rho big-delta)
                         (difference n m)))
         (u (rtimes-nat n
                        (rtimes rho big-sigma)))))
       (enable rtimes-nat-rquotient-nat rtimes-nat-rtimes-nat2
               reduce-rtimes-nat
               rquotient-nat-rtimes-nat rearrange-delta-step4 rearrange-delta-step5)
       (enable-theory reductions)))
```

;; THE INTERACTIVE CONVERGENCE ALGORITHM PROOF

```
(constrain parameters-intro (rewrite)
      ;; R and S
     (and (rationalp (R))
          (rationalp (S))
          (rlessp (rational 0 1) (R))                          ;; posR
          (rlessp (rational 0 1) (S))                          ;; posS
          (rleq (rtimes (rational 3 1) (S)) (R))               ;; C1
      ;; rho
          (rationalp (rho))
          (rleq (rational 0 1) (rtimes (rational 1 2) (rho))) ;; rho_pos
          (rlessp (rtimes (rational 1 2) (rho))                ;; rho_small
                  (rational 1 1))
```

```
      ;; other parameters
          (rationalp (epsilon))
          (rationalp (delta))
          (rationalp (delta0))
          (rationalp (big-sigma))
          (rationalp (big-delta))
          (numberp (n))
          (not (equal (n) 0))                                ;; C0_a
          (numberp (m))                                      ;; C0_b
          (lessp (m) (n))
          (rlessp (rational 0 1) (big-delta))                ;; C0_c
          (rleq (big-sigma) (s))                             ;; C2
          (rleq (big-delta) (big-sigma))                     ;; C3
          (rleq (rplus (delta)                               ;; C4
                       (rplus (epsilon)
                              (rtimes (rational 1 2)
                                      (rtimes (rho) (s)))))
                (big-delta))
          (rleq (rplus (delta0) (rtimes (rho) (R)))          ;; C5
                (delta))
          (rleq (rplus (rtimes (rational 2 1)                ;; c6
                               (rplus (epsilon) (rtimes (rho) (S))))
                       (rplus (rquotient-nat (rtimes-nat (times 2 (m)) (big-delta))
                                                         (difference (n) (m)))
                              (rplus (rquotient-nat (rtimes-nat (n) (rtimes (rho) (r)))
                                                    (difference (n) (m)))
                                     (rplus (rtimes (rho) (big-delta))
                                            (rquotient-nat
                                               (rtimes-nat (n) (rtimes (rho) (big-sigma)))
                                               (difference (n) (m)))))))
                (delta)))
     ((R (lambda () (rational 3 1)))
      (S (lambda () (rational 1 1)))
      (rho (lambda () (rational 0 1)))
      (epsilon   (lambda () (rational 0 1)))
      (delta     (lambda () (rational 0 1)))
      (delta0    (lambda () (rational 0 1)))
      (big-sigma (lambda () (rational 1 2)))
      (big-delta (lambda () (rational 1 2)))
      (n         (lambda () 1))
      (m         (lambda () 0))))

(lemma big-sigma-positive (rewrite)
       (rlessp (rational 0 1) (big-sigma))
      ((use (rlessp-rleq-transitivity (x (rational 0 1)) (y (big-delta)) (z (big-sigma))))))

(lemma S-rleq (rewrite)
      (and (rlessp (rational 0 1) (S))
           (rleq (rational 0 1) (S)))
      ((use (rlessp-rleq (x (rational 0 1)) (y (s))))))

(lemma c5 (rewrite)
      (rleq (rplus (delta0) (rtimes (rho) (r))) (delta)))

;; This is just a part of parameters-intro isolated so that I could USE it more
;; conveniently.

(lemma c6 (rewrite)
          (rleq (rplus (rtimes (rational 2 1) (rplus (epsilon) (rtimes (rho) (S))))
                       (rplus (rquotient-nat (rtimes-nat (times 2 (m)) (big-delta))
                                                         (difference (n) (m)))
                              (rplus (rquotient-nat (rtimes-nat (n) (rtimes (rho) (r)))
                                                    (difference (n) (m)))
                                     (rplus (rtimes (rho) (big-delta))
                                            (rquotient-nat
                                               (rtimes-nat (n) (rtimes (rho) (big-sigma)))
                                               (difference (n) (m)))))))
                (delta))
          ((use (parameters-intro))))

(lemma SinR (rewrite)
       (rlessp (S) (R))
      ((enable rtimes3-rlessp)))
```

```
(constrain T0-intro (rewrite)
      (rationalp (T0))
     ((T0 (lambda () (rational 0 1)))))

(defn Ti (i)
      (rplus (T0) (rtimes-nat i (R))))

(disable ti)

(lemma ti-zerop (rewrite)
      (implies (zerop i)
               (equal (ti i) (reduce (t0))))
     ((enable ti rplus-rzerop rtimes-nat-zerop)))

(lemma ti-next (rewrite)
      (equal (ti (add1 i))
             (rplus (ti i) (R)))
     ((enable commutativity-of-rplus ti rtimes-nat-add1
              commutativity2-of-rplus
              rplus-reduce)))

(lemma not-numberp-ti (rewrite)
       (implies (not (numberp i))
                (equal (ti i) (ti 0)))
       ((enable ti rtimes-nat-zerop)))
```

;; We use a different but equivalent notion of Rdef. The Rushby approach uses
;; an unnecessary existential quantifier.

```
(defn-sk+ Rdef (tm i)
      (exists pi
              (and (rleq (rational 0 1) pi)
                   (rleq pi (R))
                   (equal (reduce tm) (rplus (ti i) pi)))))

(defn in-interval (tm low high)
      (and (rleq low tm)
           (rleq tm high)))

(disable in-interval)

(lemma in-interval-inclusion (rewrite)
      (implies (and (in-interval y low x)
                    (in-interval x low high))
               (in-interval y low high))
     ((enable rleq-transitive in-interval)))

(defn in-R (tm i)
      (in-interval tm (Ti i) (ti (add1 i))))

(lemma not-numberp-in-r (rewrite)
       (implies (not (numberp i))
                (equal (in-r tm i)
                       (in-r tm 0)))
      ((enable in-r not-numberp-ti)))

(disable in-r)
```

;; This shows that the two definitions of Rdef are equivalent. Subsequently, we won't
;; bother with Rushby's definition.

```
(prove-lemma Rdef-in-R-equivalence ()
      (iff (Rdef tm i)
           (in-R tm i))
     ((use (rdef-necc)
           (rdef-suff (pi (rplus tm (rneg (ti i))))))
      (enable in-interval rdifference rleq-rdifference3 in-r
              rleq-rdifference4 ti-next rleq-rplus rplus-preserves-rleq)
      (do-not-induct t)))
```

;; Again, the Rushby definition is quite different but we prove below
;; the equivalence of the two.

```
(defn-sk+ Sdef (tm i)
      (exists pi
              (and (rleq (rational 0 1) pi)
                   (rleq pi (S))
                   (equal (reduce tm) (rplus (ti i) (rplus (rdifference (R) (S)) pi))))))

(defn in-S (tm i)
      (in-interval tm
                   (rdifference (Ti (add1 i)) (S))
                   (Ti (add1 i))))

(disable in-s)

(lemma sdef-in-s-equivalence-case3 (rewrite)
       (implies (and (equal (rplus (s) tm)
                            (rplus (ti i)
                                   (rplus (r) (pi-1 i tm))))
                     (rleq (pi-1 i tm) (s))
                     (rleq (rational 0 1) (pi-1 i tm)))
                (rleq tm (rplus (ti i) (r))))
       ((use (rleq-rplus (z tm) (y (rplus (ti i) (r))) (x (rdifference (s) (pi-1 i tm)))))
        (enable reduce-rplus rdifference commutativity-of-rplus
                commutativity2-of-rplus associativity-of-rplus
                rleq-rdifference-rzero)))

(lemma sdef-equivalence-hack (rewrite)
      (implies (rleq tm (rplus (ti i) (r)))
               (rleq (rplus tm
                            (rplus (rneg (ti i)) (rneg (r))))
                     (rational 0 1)))
     ((enable-theory reductions)
      (enable rleq requal)))

(prove-lemma Sdef-in-S-equivalence ()
      (iff (Sdef tm i)
           (in-S tm i))
     ((use (sdef-necc)
           (sdef-suff (pi (rdifference tm (rplus (ti i) (rdifference (r) (s)))))))
      (enable-theory reductions)
      (enable in-s in-interval rdifference associativity-of-rplus
              rleq-rplus ti-next sdef-in-s-equivalence-case3 sdef-equivalence-hack
              rleq-rplus-hack rneg-rplus rneg-rneg rdifference rleq-rplus-hack2)))

(lemma inRS (rewrite)
      (implies (in-S tm i)
               (in-R tm i))
     ((enable rdifference sinr rlessp-rdifference2 in-s in-r in-interval
              associativity-of-rplus commutativity-of-rplus ti-next)
       (use (rplus-preserves-rleq3
             (x (ti i))
             (y tm)
             (z (rplus (r) (rneg (s))))))))

(lemma Ti-in-S (rewrite)
      (in-S (Ti (add1 i)) i)
     ((enable-theory reductions)
      (enable in-s in-interval rdifference ti-next rleq requal
              associativity-of-rplus rlessp-rleq s-rleq rleq-reflexive)))

(lemma in-S-lemma (rewrite)
      (implies (and (in-S t1 i)
                    (in-S t2 i))
               (rleq (rabs (rdifference t1 t2)) (S)))
     ((use (betweenness-distance (p1 t1) (p2 t2)
                                 (low (rplus (ti i) (rplus (r) (rneg (s)))))
                                 (high (rplus (ti i) (r)))))
      (enable in-s in-interval rdifference rneg-rplus rneg-rneg reduce-rneg
              ti-next associativity-of-rplus rleq-transitive rabs-positive2
              rleq-reduce rleq-reflexive s-rleq)))
```

```
(lemma in-S-lemma2 (rewrite)
      (implies (and (in-S t1 i)
                    (in-S t2 i))
               (rleq (rabs (rplus t1 (rneg t2))) (S)))
     ((use (in-s-lemma))
      (enable rdifference)))

(constrain clock-intro (rewrite)
     (rationalp (clock p ct))
    ((clock (lambda (x y) (rational 0 1)))))

(lemma rho-rleq0 (rewrite)
      (rleq (rational 0 1) (rho))
     ((use (rtimes-rleq (x (rational 1 2)) (y (rho))))))

(defn-sk+ good-clock (p low high)
  ;; This says that p's clock is good within the interval [low, high]
  ;; where rho is the maximum clock drift rate.
  (forall (t1 t2)
          (implies (and (in-interval t1 low high)
                        (in-interval t2 low high))
                   (rleq (rabs (rplus (clock p t1)
                                      (rplus (rneg (clock p t2))
                                             (rplus (rneg t1) t2))))
                         (rtimes (rational 1 2)
                                 (rtimes (rho)
                                         (rabs (rplus t1 (rneg t2)))))))))

;; Delta2 is the function that reads the difference between the clocks of r and p
;; in period i.  If either of r or p is not a process, then all bets are off.

(constrain delta2-intro (rewrite)
       (and (rationalp (delta2 r p i))
            (equal (delta2 p p i) (rational 0 1))
            (implies (not (numberp i))
                     (equal (delta2 r p i)
                            (delta2 r p 0)))
            (equal (reduce (delta2 p q i))
                   (delta2 p q i)))
     ((delta2 (lambda (r p i) (rational 0 1)))))

(defn d2-bar (r p i)
      (if (and (not (equal r p))
               (rlessp (rabs (delta2 r p i)) (big-delta)))
          (delta2 r p i)
        (rational 0 1)))

(disable d2-bar)

;; This assumes that processes are numbered from 1..(n).

(defn d2-bar-list (n p i)
      (if (zerop n)
          nil
        (cons (d2-bar n p i) (d2-bar-list (sub1 n) p i))))

(lemma length-d2-bar-list (rewrite)
       (equal (length (d2-bar-list n p i))
              (fix n)))

(defn d2-bar-mean (n p i)
      (rmean (d2-bar-list n p i)))

(disable d2-bar-mean)

(defn delta1 (p i)
      (d2-bar-mean (n) p i))

(disable delta1)
```

```
(lemma non-numberp-d2-bar-list (rewrite)
      (implies (not (numberp i))
               (equal (d2-bar-list n p i)
                      (d2-bar-list n p 0)))
     ((enable delta1 d2-bar)))

(lemma non-numberp-delta1 (rewrite)
      (implies (not (numberp i))
               (equal (delta1 p i)
                      (delta1 p 0)))
     ((enable delta1 non-numberp-d2-bar-list
              d2-bar-mean)))

(constrain corr0-intro (rewrite)
       (and (rationalp (corr0))
            (equal (reduce (corr0)) (corr0)))
     ((corr0 (lambda () (rational 0 1)))))

(defn corr (p i)
      (if (zerop i)
          (corr0)
        (rplus (corr p (sub1 i))
               (delta1 p (sub1 i)))))

(lemma corr-add1 (rewrite)
       (equal (corr p (add1 i))
              (rplus (corr p i) (delta1 p i)))
      ((enable non-numberp-delta1 corr)))

(defn adjusted (p i tm)
      (rplus tm (corr p i)))

(disable adjusted)

(lemma adjusted-zero (rewrite)
      (equal (adjusted p 0 tm)
             (rplus tm (corr0)))
     ((enable adjusted)))

(lemma adjusted-reduce (rewrite)
      (equal (adjusted p i (reduce tm))
             (adjusted p i tm))
     ((enable adjusted rplus-reduce)))

(lemma not-numberp-adjusted (rewrite)
      (implies (not (numberp i))
               (equal (adjusted p i tm)
                      (adjusted p 0 tm)))
     ((enable adjusted)))

(lemma adjusted-rplus (rewrite)
       (equal (adjusted p i (rplus x y))
              (rplus (adjusted p i x) y))
      ((enable adjusted associativity-of-rplus
               commutativity-of-rplus)))

(defn c (p i tm)
      (clock p (adjusted p i tm)))

(disable c)

(lemma clock-prop (rewrite)
      (equal (c p (add1 i) tm)
             (c p i (rplus tm (delta1 p i))))
      ((enable c adjusted corr non-numberp-delta1
               associativity-of-rplus commutativity-of-rplus)))

(lemma c-reduce (rewrite)
      (equal (c p i (reduce tm))
             (c p i tm))
     ((enable c adjusted-reduce)))
```

```
(lemma c-commutativity (rewrite)
       (equal (c p i (rplus y x))
              (c p i (rplus x y)))
      ((enable commutativity-of-rplus)))

(lemma d2-bar-prop (rewrite)
      (rlessp (rabs (d2-bar p q i)) (big-delta))
     ((enable d2-bar)))

(defn skew (p q tm i)
      (rabs (rdifference (c p i tm)
                         (c q i tm))))

(disable skew)

(lemma not-numberp-skew (rewrite)
       (implies (not (numberp i))
                (equal (skew p q tm i)
                       (skew p q tm 0)))
      ((enable skew c not-numberp-adjusted)))

(defn nonfaulty (p i)
      (good-clock p (adjusted p 0 (ti 0)) (adjusted p i (ti (add1 i)))))

(lemma not-numberp-nonfaulty (rewrite)
       (implies (not (numberp i))
                (equal (nonfaulty p i)
                       (nonfaulty p 0)))
      ((enable nonfaulty not-numberp-adjusted)))

(defn faulty (p i)
      (not (nonfaulty p i)))

(disable nonfaulty)

(defn-sk+ S1A (i)
     (forall r
             (implies (and (leq (add1 (m)) r)
                           (leq r (n)))
                      (nonfaulty r i))))

(defn-sk+ S1C (p q i)
     (forall tm
             (implies (and (nonfaulty p i)
                           (nonfaulty q i)
                           (in-R tm i))
                      (rleq (skew p q tm i) (delta)))))

(lemma not-numberp-S1C (rewrite)
       (implies (and (not (numberp i))
                     (S1C p q 0))
                (S1C p q i))
      ((use (S1C-necc (i 0) (tm (tm i p q))))
       (enable not-numberp-skew not-numberp-nonfaulty not-numberp-in-r)))

(defn S2 (p i)
      (rlessp (rabs (rdifference (corr p (add1 i))
                                 (corr p i)))
              (big-sigma)))

(disable s2)
```

;; These are the basic assumptions of the theorem

```
(axiom A0 (rewrite)
      (rlessp (skew p q (ti 0) 0) (delta0)))

(defn-sk+ some-ok-time (p q i)
      (exists t0
              (and (in-S t0 i)
                   (rlessp (rabs (rdifference (c p i (rplus t0 (delta2 q p i)))
                                              (c q i t0)))
                           (epsilon)))))
```

```
(axiom A2 (rewrite)
      (implies (and (nonfaulty p i)
                    (nonfaulty q i)
                    (S1C p q i)
                    (S2 p i))
               (and (rleq (rabs (delta2 q p i)) (s))
                    (some-ok-time p q i))))

(lemma d2-bar-list-listp (rewrite)
      (equal (listp (d2-bar-list n p i))
             (not (zerop n))))

(lemma d2-bar-list-all-rlessp-big-delta (rewrite)
       (all-rlessp (map-rabs (d2-bar-list n p i))
                   (big-delta))
      ((enable d2-bar-prop)))

(disable rmean)

(lemma delta1-rlessp (rewrite)
       (rlessp (rabs (delta1 p i))
               (big-sigma))
      ((use (rlessp-rleq-transitivity2 (x (rabs (delta1 p i)))
                                       (y (rmean (map-rabs (d2-bar-list (n) p i))))
                                       (z (big-sigma)))
            (all-rlessp-rleq-transitive (x (big-delta)) (y (big-sigma))
                                        (lst  (map-rabs (d2-bar-list (n) p i))))
            (mean-bound (lst (map-rabs (d2-bar-list (n) p i))) (x (big-sigma))))
       (enable delta1 d2-bar-mean abs-mean listp-map-rabs d2-bar-list-listp
               d2-bar-list-all-rlessp-big-delta)))

(lemma theorem2 (rewrite)
      (S2 p i)
     ((enable rdifference s2 associativity-of-rplus
              delta1-rlessp rabs-reduce
              rplus-rzerop)))

(lemma upper-bound (rewrite)
      (implies (and (in-s tm i)
                    (rleq (rabs pi) (rdifference (r) (s))))
               (rleq (adjusted p i (rplus tm pi))
                     (adjusted p (add1 i) (ti (add1 (add1 i))))))
    ((use (rleq-transitive (x pi)
                           (y (rplus (r) (rneg (big-sigma))))
                           (z (rplus (r) (delta1 p i))))
          (theorem2)
          (abs-ax6 (x pi)
                   (y (rdifference (r) (s))))
          (abs-ax6 (x (rdifference (corr p (add1 i))
                                   (corr p i)))
                   (y (big-sigma))))
     (enable rlessp-rleq big-sigma-positive adjusted-rplus
             rneg-greater-rleq rplus-cancel rplus-preserves-rleq-hack
             associativity-of-rplus ti-next in-s in-interval
             rdifference rleq-reduce adjusted corr-add1 s2 rlessp-rleq)))

(lemma small-shift (rewrite)
      (rleq (rneg (r)) (rdifference (corr p (add1 i)) (corr p i)))
     ((use (theorem2) (sinr))
      (disable corr theorem2 sinr)
      (enable rabs-rneg-rleq rlessp-transitive rdifference
              rlessp-rleq-transitivity2 s2)))

(lemma adj-inductive-step (rewrite)
      (implies (rleq t0 (adjusted p i (ti i)))
               (rleq t0 (adjusted p (add1 i) (ti (add1 i)))))
     ((use (small-shift))
      (disable corr)
      (enable rleq-transitive associativity-of-rplus rplus-cancel ti-next adjusted
              rleq-reduce rdifference rplus-rneg-rleq-hack)))
```

```
(defn sub1-induction (i)
      (if (zerop i)
          t
        (sub1-induction (sub1 i))))

(lemma adj-always-positive (rewrite)
      (rleq (rplus (t0) (corr0)) (adjusted p i (ti i)))
     ((induct (sub1-induction i))
      (enable ti-zerop adjusted-reduce not-numberp-adjusted
              adjusted-zero rleq-reduce rleq-reflexive adj-inductive-step)))

(lemma lower-bound (rewrite)
      (implies (rleq (rational 0 1) pi)
               (rleq (adjusted p 0 (ti 0))
                     (adjusted p i (rplus (ti i) pi))))
      ((use (adj-always-positive))
       (enable-theory reductions)
       (enable ti-zerop rleq-rplus2 adjusted-zero adjusted-rplus)))

(lemma lower-bound2 (rewrite)
      (implies (and (in-s tm i)
                    (rleq (rabs pi) (rdifference (r) (s))))
               (rleq (adjusted p 0 (ti 0))
                     (adjusted p i (rplus tm pi))))
      ((use (lower-bound (pi (rplus tm (rplus (rneg (ti i)) pi))))
            (rleq-transitive (x (rational 0 1))
                             (y (rplus tm
                                       (rplus (rneg (ti i))
                                              (rplus (rneg (r)) (s)))))
                             (z (rplus tm (rplus (rneg (ti i)) pi)))))
            (enable rplus-reduce rleq-hack
                    associativity-of-rplus rneg-rneg ti-next
                    rneg-rplus rdifference in-s in-interval
                    rleq-reduce rabs-rneg-rplus)))

(lemma gc-prop (rewrite)
       (implies (and (good-clock p t0 tn)
                     (in-interval tm t0 tn))
                (good-clock p t0 tm))
     ((use (good-clock-necc (t2 (t2 tm t0 p)) (t1 (t1 tm t0 p))
                            (high tn) (low t0) (p p))
           (in-interval-inclusion (low t0) (high tn) (x tm) (y (t1 tm t0 p)))
           (in-interval-inclusion (low t0) (high tn) (x tm) (y (t2 tm t0 p))))))

(lemma bounds (rewrite)
      (and (rleq (adjusted p 0 (ti 0))
                 (adjusted p i (ti (add1 i))))
           (rleq (adjusted p i (ti (add1 i)))
                 (adjusted p (add1 i) (ti (add1 (add1 i))))))
      ((use (lower-bound2 (pi (rational 0 1)) (tm (ti (add1 i))))
            (upper-bound(pi (rational 0 1)) (tm (ti (add1 i)))))
       (enable ti-in-s rlessp-rdifference2 sinr rplus-rzerop
               adjusted-reduce rdifference)))

(lemma nonfx (rewrite)
      (implies (nonfaulty p (add1 i))
               (nonfaulty p i))
     ((use (gc-prop (t0 (adjusted p 0 (ti 0)))
                    (tn (adjusted p
                                  (add1 i)
                                  (ti (add1 (add1 i)))))
                    (tm (adjusted p i (ti (add1 i))))))
      (enable nonfaulty in-interval bounds)))

(lemma s1a-lemma (rewrite)
      (implies (s1a (add1 i))
               (s1a i))
     ((use (s1a-necc (r (r-1 i)) (i (add1 i))))
      (enable nonfx)))
```

```
(lemma lemma2 (rewrite)
      (implies (and (nonfaulty p (add1 i))
                    (rleq (adjusted p i tm)
                          (adjusted p (add1 i) (ti (add1 (add1 i)))))
                    (rleq (adjusted p 0 (ti 0))
                          (adjusted p i tm))
                    (rleq (adjusted p i (rplus tm pi))
                          (adjusted p (add1 i) (ti (add1 (add1 i)))))
                    (rleq (adjusted p 0 (ti 0)) (adjusted p i (rplus tm pi))))
               (rleq (rabs (rplus (c p i (rplus tm pi))
                                  (rplus (rneg (c p i tm))
                                         (rneg pi))))
                     (rtimes (rational 1 2) (rtimes (rho) (rabs pi)))))
      ((use (good-clock-necc (low (adjusted p 0 (ti 0)))
                             (high (adjusted p
                                             (add1 i)
                                             (ti (add1 (add1 i)))))
                             (t2 (adjusted p i tm))
                             (t1 (adjusted p i (rplus tm pi)))))
       (enable adjusted-rplus associativity-of-rplus nonfaulty c
               rabs-reduce rneg-rplus reduce-rneg in-interval)))

(lemma lemma2a (rewrite)
       (implies (and (nonfaulty p (add1 i))
                     (rleq (rabs (rplus pi phi)) (rdifference (r) (s)))
                     (rleq (rabs phi) (rdifference (r) (s)))
                     (in-s tm i))
                (rleq (rabs (rplus (c p i (rplus tm (rplus phi pi)))
                                   (rplus (rneg (c p i (rplus tm phi)))
                                          (rneg pi))))
                      (rtimes (rational 1 2) (rtimes (rho) (rabs pi)))))
       ((use (lemma2 (tm (rplus tm phi))))
        (enable upper-bound lower-bound2 associativity-of-rplus
                rabs-commutativity-hack)))

(lemma lemma2b-step (rewrite)
       (implies (and (rleq (rabs phi) (s))
                     (rleq (rabs pi) (s)))
                (rleq (rabs (rplus pi phi))
                      (rplus (r) (rneg (s)))))
       ((use (rleq-transitive (x (rabs (rplus pi phi)))
                              (y (rplus (rabs pi) (rabs phi)))
                              (z (rplus (r) (rneg (s))))))
        (enable times-3-rleq-rewrite rabs-rplus-rleq s-rleq parameters-intro)))

(lemma lemma2b-step2 (rewrite)
       (implies (rleq (rabs phi) (s))
                (rleq (rabs phi)
                      (rplus (r) (rneg (s)))))
      ((enable times-3-rleq-rewrite2 s-rleq parameters-intro)))

(lemma lemma2b (rewrite)
      (implies (and (nonfaulty p (add1 i))
                    (rleq (rabs phi) (s))
                    (rleq (rabs pi) (s))
                    (in-s tm i))
               (rleq (rabs (rplus (c p i (rplus tm (rplus phi pi)))
                                  (rplus (rneg (c p i (rplus tm phi)))
                                         (rneg pi))))
                     (rtimes (rational 1 2) (rtimes (rho) (rabs pi)))))
      ((enable lemma2a lemma2b-step rdifference lemma2b-step2)))

(lemma lemma2c (rewrite)
      (implies (and (nonfaulty p (add1 i))
                    (rleq (rabs pi) (s))
                    (in-s tm i))
               (rleq (rabs (rplus (c p i (rplus tm pi))
                                  (rplus (rneg (c p i tm))
                                         (rneg pi))))
                     (rtimes (rational 1 2) (rtimes (rho) (rabs pi)))))
     ((use (lemma2b (phi (rational 0 1))))
      (enable s-rleq rplus-rzerop rplus-reduce
              c-reduce)))
```

```
(lemma lemma2d (rewrite)
       (implies (and (nonfaulty p i)
                     (rleq (rational 0 1) pi)
                     (rleq pi (r)))
                (rleq (rabs (rplus (c p i (rplus (ti i) pi))
                                   (rplus (rneg (c p i (ti i)))
                                          (rneg pi))))
                      (rtimes (rational 1 2) (rtimes (rho) (rabs pi)))))
       ((use (good-clock-necc (low (adjusted p 0 (ti 0)))
                              (high (adjusted p i (ti (add1 i))))
                              (t1 (adjusted p i (rplus (ti i) pi)))
                              (t2 (adjusted p i (ti i)))))
        (enable-theory reductions)
        (enable nonfaulty in-interval lower-bound adjusted-rplus ti-next
                rplus-cancel adjusted-zero ti-zerop adj-always-positive
                rleq-rplus2 rleq-reflexive rleq-transitive c rneg-rplus
                associativity-of-rplus rabs-reduce)))

(lemma rabs-negate-lemma1-hack (rewrite)
       (equal (rabs (rplus (c p i t0)
                           (rplus (delta2 q p i)
                                  (rneg (c p i
                                           (rplus t0
                                                  (delta2 q p i)))))))
              (rabs (rplus (c p i
                              (rplus t0 (delta2 q p i)))
                           (rplus (rneg (c p i t0))
                                  (rneg (delta2 q p i))))))
      ((use (rabs-negate-hack (x (c p i t0))
                              (y (delta2 q p i))
                              (z (c p i (rplus t0 (delta2 q p i))))))))

(lemma lemma1 (rewrite)
      (implies (and (s1c p q i)
                    (s2 p i)
                    (nonfaulty p (add1 i))
                    (nonfaulty q (add1 i)))
               (rlessp (rabs (delta2 q p i)) (big-delta)))
     ((use (a2)
           (some-ok-time-necc)
           (s1c-necc (tm (t0-1 i p q)))
           (rabs-rplus-rleq2
            (x (rplus (c p i (t0-1 i p q))
                      (rplus (delta2 q p i)
                             (rneg (c p i
                                      (rplus (t0-1 i p q)
                                             (delta2 q p i)))))))
            (y (rplus (c q i (t0-1 i p q))
                      (rneg (c p i (t0-1 i p q)))))
            (z (rplus (c p i
                         (rplus (t0-1 i p q) (delta2 q p i)))
                      (rneg (c q i (t0-1 i p q))))))
           (lemma2c (pi (delta2 q p i)) (tm (t0-1 i p q)))
           (rlessp-rleq-transitivity2
            (x (rabs (delta2 q p i)))
            (y (rplus (rabs (rplus (c p i (t0-1 i p q))
                                   (rplus (delta2 q p i)
                                          (rneg (c p i
                                                   (rplus (t0-1 i p q)
                                                          (delta2 q p i)))))))
                      (rplus (rabs (rplus (c q i (t0-1 i p q))
                                          (rneg (c p i (t0-1 i p q)))))
                             (rabs (rplus (c p i
                                             (rplus (t0-1 i p q) (delta2 q p i)))
                                          (rneg (c q i (t0-1 i p q))))))))
            (z (big-delta)))
```

```
          (rlessp-rleq-transitivity
           (x (rplus (rabs (rplus (c p i (t0-1 i p q))
                                  (rplus (delta2 q p i)
                                         (rneg (c p i
                                                  (rplus (t0-1 i p q)
                                                         (delta2 q p i)))))))
                     (rplus (rabs (rplus (c q i (t0-1 i p q))
                                         (rneg (c p i (t0-1 i p q)))))
                            (rabs (rplus (c p i
                                            (rplus (t0-1 i p q) (delta2 q p i)))
                                         (rneg (c q i (t0-1 i p q))))))))
           (y (rplus (delta)
                     (rplus (epsilon)
                            (rtimes (rational 1 2)
                                    (rtimes (rho) (s))))))
           (z (big-delta)))
          (rleq-transitive
           (x (rabs (rplus (c p i (t0-1 i p q))
                           (rplus (delta2 q p i)
                                  (rneg (c p i
                                           (rplus (t0-1 i p q)
                                                  (delta2 q p i))))))))
           (y (rtimes (rational 1 2)
                      (rtimes (rho)
                              (rabs (delta2 q p i)))))
           (z (rtimes (rational 1 2) (rtimes (rho) (s))))))
     (enable rabs-rdifference rleq-rtimes-hack rho-rleq0 rlessp-rleq nonfx
             inrs skew rdifference associativity-of-rplus
             parameters-intro rleq-rlessp-hack rabs-negate-lemma1-hack)
     (enable-theory reductions)))

(lemma lemma3 (rewrite)
       (implies (and (s1c p q i)
                     (s2 p i)
                     (nonfaulty p (add1 i))
                     (nonfaulty q (add1 i))
                     (in-s tm i))
                (rlessp (rabs (rplus (c p i (rplus tm (delta2 q p i)))
                                     (rneg (c q i tm))))
                        (rplus (epsilon)
                               (rtimes (rho) (s)))))
       ((use (a2)
             (some-ok-time-necc)
             (rearrange-alt
              (x (c p i (rplus tm (delta2 q p i))))
              (y (c q i tm))
              (u (c p i
                    (rplus (t0-1 i p q) (delta2 q p i))))
              (v (rplus tm (rneg (t0-1 i p q))))
              (w (c q i (t0-1 i p q))))
             (lemma2c
              (p q)
              (tm (t0-1 i p q))
              (pi (rplus tm (rneg (t0-1 i p q)))))
             (lemma2b
              (tm (t0-1 i p q))
              (phi (delta2 q p i))
              (pi (rplus tm (rneg (t0-1 i p q)))))
             (rlessp-rleq-transitivity2
              (x (rabs (rplus (c p i (rplus tm (delta2 q p i)))
                              (rneg (c q i tm)))))
              (y (rplus (rabs (rplus (c p i (rplus tm (delta2 q p i)))
                                     (rplus (rneg (c p i
                                                     (rplus (delta2 q p i) (t0-1 i p q))))
                                            (rplus (rneg tm) (t0-1 i p q)))))
                        (rplus (rabs (rplus (c p i
                                               (rplus (delta2 q p i) (t0-1 i p q)))
                                            (rneg (c q i (t0-1 i p q)))))
                               (rabs (rplus (c q i tm)
                                            (rplus (rneg (c q i (t0-1 i p q)))
                                                   (rplus (rneg tm) (t0-1 i p q))))))))
              (z (rplus (epsilon)
                        (rtimes (rho) (s)))))
```

```
            (rleq-transitive
             (x (rplus (rabs (rplus (c p i (rplus tm (delta2 q p i)))
                                    (rplus (rneg (c p i
                                                    (rplus (delta2 q p i) (t0-1 i p q))))
                                           (rplus (rneg tm) (t0-1 i p q)))))
                       (rabs (rplus (c q i tm)
                                    (rplus (rneg (c q i (t0-1 i p q)))
                                           (rplus (rneg tm) (t0-1 i p q)))))))
             (y (rtimes (rho)
                        (rabs (rplus tm (rneg (t0-1 i p q))))))
             (z (rtimes (rho) (s)))))
      (enable-theory reductions)
      (enable in-s-lemma2 rho-rleq0 rlessp-rplus-triple rplus-reduce
              rneg-rneg rneg-rplus in-s-lemma2 c-reduce nonfx rdifference
              c-commutativity rlessp-transitive associativity-of-rplus
              rho-rleq0 rlessp-rleq-hack rleq-rtimes-hack rleq-half-rplus)))

(lemma sublemma1 (rewrite)
      (implies (and (s1c p r i)
                    (s2 p i)
                    (nonfaulty p (add1 i))
                    (nonfaulty r (add1 i)))
               (equal (d2-bar r p i)
                      (delta2 r p i)))
     ((use (lemma1 (q r)))
      (enable d2-bar delta2-intro)))

(lemma lemma2x (rewrite)
       (implies (and (s1c p r i)
                     (s2 p i)
                     (nonfaulty p (add1 i))
                     (nonfaulty r (add1 i))
                     (in-s tm i))
                (rleq (rabs (rplus (c p i (rplus tm (delta2 r p i)))
                                   (rplus (rneg (c p i tm))
                                          (rneg (delta2 r p i)))))
                      (rtimes (rational 1 2) (rtimes (rho) (big-delta)))))
       ((use (lemma2c (pi (delta2 r p i)))
             (lemma1 (q r))
             (rleq-transitive
              (x (rabs (rplus (c p i (rplus tm (delta2 r p i)))
                              (rplus (rneg (c p i tm))
                                     (rneg (delta2 r p i))))))
              (y (rtimes (rational 1 2)
                         (rtimes (rho) (rabs (delta2 r p i)))))
              (z (rtimes (rational 1 2)
                         (rtimes (rho) (big-delta))))))
        (enable rleq-rtimes-hack rho-rleq0 rlessp-rleq nonfx a2
                rleq-rtimes-pos2 rleq-transitive)))

(lemma lemma4-version1 (rewrite)
      (implies (and (s1c q r i)
                    (s1c p q i)
                    (s1c p r i)
                    (s2 p i)
                    (s2 q i)
                    (s2 r i)
                    (nonfaulty p (add1 i))
                    (nonfaulty q (add1 i))
                    (nonfaulty r (add1 i))
                    (in-s tm i))
               (rlessp (rabs (rplus (rplus (c p i tm)
                                           (d2-bar r p i))
                                    (rplus (rneg (c q i tm))
                                           (rneg (d2-bar r q i)))))
                       (rplus (rtimes (rational 1 2) (rtimes (rho) (big-delta)))
                              (rplus (rtimes (rational 1 2) (rtimes (rho) (big-delta)))
                                     (rplus (rplus (epsilon) (rtimes (rho) (s)))
                                            (rplus (epsilon) (rtimes (rho) (s))))))))
       ((use (rearrange3
              (x (rplus (c p i tm) (d2-bar r p i)))
              (y (rplus (c q i tm) (d2-bar r q i)))
              (u (c q i (rplus tm (d2-bar r q i))))
              (v (c p i (rplus tm (d2-bar r p i))))
              (w (c r i tm))))
```

```
       (enable-theory reductions)
       (enable sublemma1 lemma3 lemma2x rdifference rlessp-rleq-transitivity2
               rlessp-rleq rlessp-rplus-pair rneg-rplus rleq-rlessp-rplus-pair)))

(lemma lemma4-hack (rewrite)
       (equal (rplus z (rplus z (rplus x (rplus y (rplus x y)))))
              (rtimes (rational 2 1) (rplus x (rplus y z))))
       ((enable rtimes-add1 rtimes-rzerop commutativity-of-rplus
                commutativity2-of-rplus)))

(lemma lemma4 (rewrite)
      (implies (and (s1c q r i)
                    (s1c p q i)
                    (s1c p r i)
                    (s2 p i)
                    (s2 q i)
                    (s2 r i)
                    (nonfaulty p (add1 i))
                    (nonfaulty q (add1 i))
                    (nonfaulty r (add1 i))
                    (in-s tm i))
               (rlessp (rabs (rplus (c p i tm)
                                    (rplus (d2-bar r p i)
                                           (rplus (rneg (c q i tm))
                                                  (rneg (d2-bar r q i))))))
                       (rtimes (rational 2 1)
                               (rplus (epsilon)
                                      (rplus (rtimes (rho) (s))
                                             (rtimes (rational 1 2)
                                                     (rtimes (rho) (big-delta))))))))
     ((use (lemma4-version1))
      (enable associativity-of-rplus lemma4-hack)))

(lemma lemma5 (rewrite)
       (implies (and (s1c p q i)
                     (nonfaulty p (add1 i))
                     (nonfaulty q (add1 i))
                     (in-s tm i))
                (rlessp (rabs (rplus (c p i tm)
                                     (rplus (d2-bar r p i)
                                            (rplus (rneg (c q i tm))
                                                   (rneg (d2-bar r q i))))))
                        (rplus (delta) (rtimes (rational 2 1) (big-delta)))))
       ((use (rearrange4 (a (c p i tm))
                         (b (c q i tm))
                         (x (d2-bar r p i))
                         (y (d2-bar r q i)))
             (s1c-necc))
        (enable rdifference associativity-of-rplus rneg-rplus
                rlessp-rleq-transitivity2 rleq-rlessp-rplus-pair skew nonfx inrs
                rlessp-times2 d2-bar-prop)))

(lemma rleq-rplus-hack3 (rewrite)
       (equal (rleq (rplus x (rplus y z)) (rplus y w))
              (rleq (rplus x z) (reduce w)))
      ((enable rleq requal)))

(lemma sublemma-a (rewrite)
       (implies (and (nonfaulty p i)
                     (nonfaulty q i)
                     (in-r tm i))
                (rleq (skew p q tm i)
                      (rplus (skew p q (ti i) i)
                             (rtimes (rho) (r)))))
      ((use (rearrange-alt (x (c p i tm))
                           (y (c q i tm))
                           (u (c p i (ti i)))
                           (v (rplus tm (rneg (ti i))))
                           (w (c q i (ti i))))
            (lemma2d (pi (rplus tm (rneg (ti i)))))
            (lemma2d (p q) (pi (rplus tm (rneg (ti i))))))
```

```
       (enable-theory reductions)
       (enable skew rdifference rneg-rplus c-reduce ti-next rabs-positive2
               rplus-rleq-rewrite rplus-rleq-rewrite2 rleq-rtimes-hack in-r
               in-interval rho-rleq0 rleq-transitive rleq-rplus-hack3 rleq-half-rplus
               rleq-rtimes-hack rplus-rleq-rewrite)))

(lemma delta1-rleq-s (rewrite)
       (rlessp (rabs (delta1 p i)) (s))
      ((use (theorem2))
       (enable s2 rdifference corr-add1 associativity-of-rplus
               rabs-reduce rlessp-transitive rleq-rlessp-relation)))

(lemma sublemma2 (rewrite)
       (equal (skew p q tm (add1 i))
              (rabs (rplus (c p i (rplus tm (delta1 p i)))
                           (rneg (c q i (rplus tm (delta1 q i)))))))
      ((enable skew rdifference clock-prop)))

(lemma lemma6 (rewrite)
       (implies (and (nonfaulty p (add1 i))
                     (nonfaulty q (add1 i))
                     (in-r tm (add1 i)))
                (rleq (skew p q tm (add1 i))
                      (rplus (rabs (rplus (c p i (ti (add1 i)))
                                          (rplus (delta1 p i)
                                                 (rplus (rneg (c q i (ti (add1 i))))
                                                        (rneg (delta1 q i))))))
                             (rplus (rtimes (rho) (r))
                                    (rtimes (rho) (big-sigma))))))
       ((use (sublemma-a (i (add1 i)))
             (rearrange
              (x (c p i
                    (rplus (ti (add1 i)) (delta1 p i))))
              (y (c q i
                    (rplus (ti (add1 i)) (delta1 q i))))
              (u (c p i (ti (add1 i))))
              (v (delta1 p i))
              (w (c q i (ti (add1 i))))
              (z (delta1 q i)))
             (lemma2c (tm (ti (add1 i))) (pi (delta1 p i)))
             (lemma2c (tm (ti (add1 i))) (pi (delta1 q i)) (p q)))
        (disable correctness-of-cancel-rplus-rleq)
        (enable rdifference rneg-rplus sublemma2 rplus-rleq-rlessp-cancel2 rleq-transitive
                rplus-rleq-rlessp-cancel delta1-rleq-s ti-in-s rleq-rtimes-pos2 rlessp-rleq
                delta1-rlessp rho-rleq0 rleq-rtimes-hack rleq-half-rplus)))


(defn l1-term-list (p q i r)
      ;; This generates the list of terms to which the mean is applied in lemma l1.
      ;; Notice that they don't have the absolute-value applied.
      (if (zerop r)
          nil
        (append (l1-term-list p q i (sub1 r))
                (list (rplus (c p i (ti (add1 i)))
                             (rplus (d2-bar r p i)
                                    (rplus (rneg (c q i (ti (add1 i))))
                                           (rneg (d2-bar r q i)))))))))

(lemma length-l1-term-list (rewrite)
       (equal (length (l1-term-list p q i r))
              (fix r))
      ((enable l1-term-list)))

(lemma l1-term-list-rewrite (rewrite)
       (equal (rsum (l1-term-list p q i r))
              (rplus (rtimes-nat r (c p i (ti (add1 i))))
                     (rplus (rsum (d2-bar-list r p i))
                            (rplus (rneg (rtimes-nat r (c q i (ti (add1 i)))))
                                   (rneg (rsum (d2-bar-list r q i)))))))
       ((enable l1-term-list d2-bar-list rsum rtimes-nat-zerop rtimes-nat-add1
                associativity-of-rplus rneg-rplus
                reduce-rplus rsum-append rplus-rzerop
                commutativity-of-rplus reduce-reduce rplus-reduce)))
```

```
(lemma l1 (rewrite)
       (rleq (rabs (rplus (c p i (ti (add1 i)))
                          (rplus (delta1 p i)
                                 (rplus (rneg (c q i (ti (add1 i))))
                                        (rneg (delta1 q i))))))
             (rmean (map-rabs (l1-term-list p q i (n)))))
       ((use (abs-mean (lst (l1-term-list p q i (n)))))
        (enable rleq-transitive rmean l1-term-list-rewrite length-l1-term-list
                rquotient-nat-rplus rquotient-nat-rtimes-nat rplus-reduce
                rneg-rtimes-nat delta1 d2-bar-mean rmean
                length-d2-bar-list rneg-rquotient-nat rleq-reflexive)))

(lemma l2 (rewrite)
       (rleq (rabs (rplus (c p i (ti (add1 i)))
                          (rplus (delta1 p i)
                                 (rplus (rneg (c q i (ti (add1 i))))
                                        (rneg (delta1 q i))))))
             (rquotient-nat (rplus (rsum (firstn (map-rabs (l1-term-list p q i (n))) (m)))
                                   (rsum (restn (map-rabs (l1-term-list p q i (n))) (m))))
                            (n)))
      ((use (l1))
       (enable rplus-rsum firstn-append-restn rmean
               length-map-rabs length-l1-term-list)))

(lemma bound-faulty (rewrite)
       (implies (and (S1A (add1 i))
                     (S1C p q i)
                     (not (zerop r))
                     (nonfaulty p (add1 i))
                     (nonfaulty q (add1 i)))
                (rlessp (rabs (rplus (c p i (ti (add1 i)))
                                     (rplus (d2-bar r p i)
                                            (rplus (rneg (c q i (ti (add1 i))))
                                                   (rneg (d2-bar r q i))))))
                        (rplus (delta) (rtimes (rational 2 1) (big-delta)))))
      ((enable lemma5 ti-in-s)))

(defn firstn-l1-induction (m n)
      (if (zerop n)
          t
        (if (zerop m)
            t
          (if (equal m n)
              t
            (firstn-l1-induction m (sub1 n))))))

(lemma firstn-l1-term-list (rewrite)
       (implies (leq m n)
                (equal (firstn (map-rabs (l1-term-list p q i n)) m)
                       (map-rabs (l1-term-list p q i m))))
      ((induct (firstn-l1-induction m n))
       (enable map-rabs-append firstn-n plist-map-rabs
                  length-map-rabs length-l1-term-list
                  firstn-append-lessp)))

(lemma l3-sublemma (rewrite)
      (implies (and (leq m (n))
                    (S1A (add1 i))
                    (S1C p q i)
                    (nonfaulty p (add1 i))
                    (nonfaulty q (add1 i))
                    (not (zerop m)))
               (all-rlessp (firstn (map-rabs (l1-term-list p q i (n))) m)
                           (rplus (delta)
                                  (rtimes (rational 2 1) (big-delta)))))
     ((induct (sub1-induction m))
      (enable all-rlessp firstn map-rabs l1-term-list bound-faulty
              map-rabs-append firstn-l1-term-list all-rlessp-append)))
```

```
(lemma l3 (rewrite)
       (implies (and (S1A (add1 i))
                     (S1C p q i)
                     (nonfaulty p (add1 i))
                     (nonfaulty q (add1 i))
                     (lessp m (n)))
                (rleq (rsum (firstn (map-rabs (l1-term-list p q i (n))) m))
                      (rtimes-nat2 (rplus (delta) (rtimes (rational 2 1) (big-delta)))
                                   m)))
       ((use (sum-bound (lst (map-rabs (l1-term-list p q i m)))
             (x (rplus (delta)
                       (rtimes (rational 2 1)
                               (big-delta)))))
             (l3-sublemma))
        (enable firstn rsum rtimes-nat-rtimes-nat2 rtimes-nat-zerop
                rleq-reflexive firstn-l1-term-list rtimes-nat-rtimes-nat2
                length-map-rabs length-l1-term-list rlessp-rleq listp-map-rabs)))

(defn-sk+ theorem1-one-step (i)
  (forall (p q)
       (implies (S1A i)
                (S1C p q i))))

(lemma bound-nonfaulty (rewrite)
       (implies (and (S1A (add1 i))
                     (S1C p q i)
                     (leq (add1 (m)) r)
                     (leq r (n))
                     (nonfaulty p (add1 i))
                     (nonfaulty q (add1 i))
                     (theorem1-one-step i))
                (rlessp (rabs (rplus (c p i (ti (add1 i)))
                                     (rplus (d2-bar r p i)
                                            (rplus (rneg (c q i (ti (add1 i))))
                                                   (rneg (d2-bar r q i))))))
                        (rtimes (rational 2 1)
                                (rplus (epsilon)
                                       (rplus (rtimes (rho) (s))
                                              (rtimes (rational 1 2)
                                                      (rtimes (rho) (big-delta))))))))
       ((use (S1A-necc (i (add1 i)))
             (theorem1-one-step-necc (p q) (q r))
             (theorem1-one-step-necc (q r)))
        (enable lemma4 theorem2 S1A-lemma ti-in-s)))

(defn l4-term-list (p q i m r)
      (if (leq m r)
          (cons (rabs (rplus (c p i (ti (add1 i)))
                             (rplus (d2-bar m p i)
                                    (rplus (rneg (c q i (ti (add1 i))))
                                           (rneg (d2-bar m q i))))))
                (l4-term-list p q i (add1 m) r))
        nil)
     ((lessp (difference (add1 r) m))))

(lemma l4-term-strip-last (rewrite)
       (implies (and (leq m r)
                     (not (zerop r)))
                (equal (l4-term-list p q i m r)
                       (append
                        (l4-term-list p q i m (sub1 r))
                        (list (rabs (rplus (c p i (ti (add1 i)))
                                           (rplus (d2-bar r p i)
                                                  (rplus (rneg (c q i (ti (add1 i))))
                                                         (rneg (d2-bar r q i)))))))))))

(lemma length-l4-term-list-from1 (rewrite)
       (equal (length (l4-term-list p q i 1 r)) (fix r))
       ((induct (sub1-induction r))
        (enable l4-term-strip-last)))
```

```
(lemma length-l4-term-list (rewrite)
       (equal (length (l4-term-list p q i (add1 m) r))
              (difference r m))
       ((enable length-l4-term-list-from1 l4-term-strip-last)))

(defn l1-l4-relation-induction (m n)
      (if (not (lessp m n))
          (if (equal m n)
              t
            (if (lessp n m) t f))
        (l1-l4-relation-induction m (sub1 n))))

(lemma l1-l4-term-lists-relation (rewrite)
       (implies (and (not (zerop n))
                     (leq m n))
                (equal (restn (map-rabs (l1-term-list p q i n)) m)
                       (l4-term-list p q i (add1 m) n)))
      ((induct (l1-l4-relation-induction m n))
       (enable restn-n plistp-map-rabs length-map-rabs length-l1-term-list
               restn-append map-rabs-append restn-1 l4-term-strip-last)))

(lemma listp-l4-term-list (rewrite)
       (implies (and (lessp m n)
                     (not (zerop n)))
                (listp (l4-term-list p q i (add1 m) n)))
      ((enable l4-term-strip-last)))

(lemma l4-sublemma (rewrite)
       (implies (and (leq r (n))
                     (leq (add1 (m)) r)
                     (S1A (add1 i))
                     (S1C p q i)
                     (nonfaulty p (add1 i))
                     (nonfaulty q (add1 i))
                     (theorem1-one-step i))
               (all-rlessp (l4-term-list p q i r (n))
                           (rtimes (rational 2 1)
                                   (rplus (epsilon)
                                          (rplus (rtimes (rho) (s))
                                                 (rtimes (rational 1 2)
                                                         (rtimes (rho) (big-delta))))))))
      ((enable bound-nonfaulty)))

(lemma l4-version1 (rewrite)
       (implies (and (lessp m (n))
                     (leq (m) m)
                     (S1A (add1 i))
                     (S1C p q i)
                     (nonfaulty p (add1 i))
                     (nonfaulty q (add1 i))
                     (theorem1-one-step i))
                (rleq (rsum (l4-term-list p q i (add1 m) (n)))
                      (rtimes-nat2 (rtimes (rational 2 1)
                                           (rplus (epsilon)
                                                  (rplus (rtimes (rho) (s))
                                                         (rtimes (rational 1 2)
                                                                 (rtimes (rho) (big-delta))))))
                                   (difference (n) m))))
      ((use (sum-bound (lst (l4-term-list p q i (add1 m) (n)))
                       (x (rtimes (rational 2 1)
                                  (rplus (epsilon)
                                         (rplus (rtimes (rho) (s))
                                                (rtimes (rational 1 2)
                                                        (rtimes (rho) (big-delta)))))))))
       (enable rtimes-nat-rtimes-nat2 rlessp-rleq
               listp-l4-term-list length-l4-term-list l4-sublemma)))

(lemma l4 (rewrite)
       (implies (and (S1A (add1 i))
                     (S1C p q i)
                     (nonfaulty p (add1 i))
                     (nonfaulty q (add1 i))
                     (theorem1-one-step i))
```

```
          (rleq (rsum (restn (map-rabs (l1-term-list p q i (n))) (m)))
                (rtimes-nat2 (rtimes (rational 2 1)
                                     (rplus (epsilon)
                                            (rplus (rtimes (rho) (s))
                                                   (rtimes (rational 1 2)
                                                           (rtimes (rho) (big-delta))))))
                             (difference (n) (m)))))
     ((enable l1-l4-term-lists-relation l4-version1)))

(lemma l5 (rewrite)
       (implies (and (S1A (add1 i))
                     (S1C p q i)
                     (nonfaulty p (add1 i))
                     (nonfaulty q (add1 i))
                     (theorem1-one-step i))
          (rleq (rabs (rplus (c p i (ti (add1 i)))
                             (rplus (delta1 p i)
                                    (rplus (rneg (c q i (ti (add1 i))))
                                           (rneg (delta1 q i))))))
                (rquotient-nat
                  (rplus (rtimes-nat2
                           (rplus (delta) (rtimes (rational 2 1) (big-delta))) (m))
                         (rtimes-nat2
                           (rtimes (rational 2 1)
                                   (rplus (epsilon)
                                          (rplus (rtimes (rho) (s))
                                                 (rtimes (rational 1 2)
                                                         (rtimes (rho) (big-delta))))))
                           (difference (n) (m))))
                  (n))))
       ((use (div-mon2
              (x (rsum (append (firstn (map-rabs (l1-term-list p q i (n))) (m))
                               (restn (map-rabs (l1-term-list p q i (n))) (m)))))
              (y (rplus
                  (rtimes-nat2 (rplus (delta) (rtimes (rational 2 1) (big-delta))) (m))
                  (rtimes-nat2 (rtimes (rational 2 1)
                                       (rplus (epsilon)
                                              (rplus (rtimes (rho) (s))
                                                     (rtimes (rational 1 2)
                                                             (rtimes (rho) (big-delta))))))
                               (difference (n) (m)))))
              (z (n)))
             (rleq-transitive
              (x (rabs (rplus (c p i (ti (add1 i)))
                               (rplus (delta1 p i)
                                      (rplus (rneg (c q i (ti (add1 i))))
                                             (rneg (delta1 q i)))))))
              (y (rquotient-nat
                  (rsum (append (firstn (map-rabs (l1-term-list p q i (n))) (m))
                                (restn (map-rabs (l1-term-list p q i (n))) (m))))
                  (n)))
              (z (rquotient-nat
                   (rplus
                    (rtimes-nat2 (rplus (delta) (rtimes (rational 2 1) (big-delta))) (m))
                    (rtimes-nat2 (rtimes (rational 2 1)
                                         (rplus (epsilon)
                                                (rplus (rtimes (rho) (s))
                                                       (rtimes (rational 1 2)
                                                               (rtimes (rho) (big-delta))))))
                                 (difference (n) (m))))
                   (n)))))
        (enable rleq-transitive rsum-append rleq-rplus-pair l2 l3 l4)))

(lemma culmination (rewrite)
       (implies (and (S1A (add1 i))
                     (S1C p q i)
                     (nonfaulty p (add1 i))
                     (nonfaulty q (add1 i))
                     (in-r tm (add1 i))
                     (theorem1-one-step i))
```

```
         (rleq (skew p q tm (add1 i))
               (rplus (rquotient-nat
                       (rplus
                        (rtimes-nat
                         (m) (rplus (delta)
                                    (rtimes (rational 2 1) (big-delta))))
                        (rtimes-nat
                          (difference (n) (m))
                          (rtimes (rational 2 1)
                                  (rplus (epsilon)
                                         (rplus (rtimes (rho) (s))
                                                (rtimes (rational 1 2)
                                                        (rtimes (rho) (big-delta)))))))
                       (n))
                      (rplus (rtimes (rho) (r))
                             (rtimes (rho) (big-sigma))))))
   ((use (l5)
         (rleq-transitive
          (x (skew p q tm (add1 i)))
          (y (rplus (rabs (rplus (c p i (ti (add1 i)))
                                 (rplus (delta1 p i)
                                        (rplus
                                         (rneg (c q i (ti (add1 i))))
                                         (rneg (delta1 q i))))))
                    (rplus (rtimes (rho) (r)) (rtimes (rho) (big-sigma)))))
          (z (rplus (rquotient-nat
                     (rplus (rtimes-nat
                              (m) (rplus (delta)
                                         (rtimes (rational 2 1)
                                                 (big-delta))))
                            (rtimes-nat
                              (difference (n) (m))
                              (rtimes (rational 2 1)
                                      (rplus (epsilon)
                                             (rplus (rtimes (rho) (s))
                                                    (rtimes (rational 1 2)
                                                            (rtimes (rho) (big-delta)))))))
                     (n))
                    (rplus (rtimes (rho) (r)) (rtimes (rho) (big-sigma)))))))
    (enable rlessp-rleq lemma6 rleq-rplus-pair rleq-reflexive rtimes-nat-rtimes-nat2)))

(lemma theorem1-basis (rewrite)
       (S1C p q 0)
      ((use (sublemma-a (i 0) (tm (tm 0 p q)))
            (rleq-transitive (x (skew p q (tm 0 p q) 0))
                             (y (rplus (skew p q (ti 0) 0) (rtimes (rho) (r))))
                             (z (delta)))
            (a0) (c5))
       (enable S1C-suff rlessp-rleq rlessp-rleq-transitive3)))

(lemma theorem1-ind-step0 (rewrite)
       (implies (and (S1A (add1 i))
                     (S1C p q i)
                     (theorem1-one-step i))
                (S1C p q (add1 i)))
      ((use (rearrange-delta (delta (delta))
                             (big-sigma (big-sigma))
                             (r (r))
                             (n (n))
                             (big-delta (big-delta))
                             (m (m))
                             (s (s))
                             (rho (rho))
                             (epsilon (epsilon)))
            (c6) (culmination (tm (tm (add1 i) p q))))
       (enable rleq-transitive)))

(lemma theorem1-ind-step (rewrite)
       (IMPLIES (AND (NOT (ZEROP I))
                     (THEOREM1-ONE-STEP (SUB1 I)))
                (THEOREM1-ONE-STEP I))
```

```
      ((use (theorem1-ind-step0 (i (sub1 i)) (p (p i)) (q (q i)))
            (theorem1-one-step-necc (i (sub1 i)) (p (p i)) (q (q i)))
            (theorem1-one-step-suff))
       (enable S1A-lemma)
       (disable theorem1-one-step-suff)))

(lemma theorem1-version1 (rewrite)
       (theorem1-one-step i)
      ((induct (sub1-induction i))
       (enable theorem1-basis not-numberp-S1C theorem1-ind-step)))

(lemma theorem1 (rewrite)
      (implies (S1A i)
               (S1C p q i))
     ((use (theorem1-version1)
           (theorem1-one-step-necc))))
```

Table of Contents

1. Introduction . . . 1
2. The Interactive Convergence Clock Synchronization Algorithm . . . 2
3. Specifying the ICCSA in the Boyer-Moore Logic . . . 2
   3.1. Computing with Rationals . . . 3
   3.2. Uses of `CONSTRAIN` . . . 5
   3.3. `DEFN-SK` . . . 7
   3.4. Avoiding Higher Order Functions . . . 9
4. Aspects of the Proof . . . 10
   4.1. Restraining the Prover . . . 10
   4.2. Order of Steps in the Proof . . . 10
   4.3. Proof Encapsulation . . . 12
   4.4. Syntax . . . 13
5. Conclusions . . . 13

List of Figures

Figure 1:	Some Quantities Required for Specifying the ICCSA	3
Figure 2:	`CONSTRAIN` Introducing ICCSA Parameters	6
Figure 3:	EHDM and Boyer-Moore Versions of the Same Lemma	7
Figure 4:	Theorems Generated for a `DEFN-SK+` Event	8
Figure 5:	Proof of `SUBLEMMA-A` Showing `USE` Hints	11