Appendix D

Raw Specifications

In order to save paper and expense, the raw specification listings are omitted from this version of the report. They are available on request from the authors.

main: **Module**

Using natinduction, algorithm, lemma6, summations, juggle

Proof

p, q, r: **Var** proc

i, j, k: **Var** period

T: **Var** clocktime

basis: **Lemma** S1(0)

basis_proof: **Prove** basis **from**
S1 $\{i \leftarrow 0\}$,
sublemma_A $\{i \leftarrow 0,\ T \leftarrow T@p3,\ p \leftarrow p@p1,\ q \leftarrow q@p1\}$,
S1C $\{i \leftarrow 0,\ p \leftarrow p@p1,\ q \leftarrow q@p1\}$,
A0 $\{p \leftarrow p@p1,\ q \leftarrow q@p1\}$,
C5

ind_step: **Lemma** S1(i) $\supset$ S1($i+1$)

ind_proof: **Prove** ind_step **from**
S1 $\{p \leftarrow p@p2,\ q \leftarrow q@p2\}$,
S1 $\{i \leftarrow i+1\}$,
culmination $\{T \leftarrow T@p5,\ p \leftarrow p@p2,\ q \leftarrow q@p2\}$,
rearrange_delta,
S1C $\{i \leftarrow i+1,\ p \leftarrow p@p2,\ q \leftarrow q@p2\}$,
C6,
S1A_lemma

Theorem_1_proof: **Prove** Theorem_1 **from**
basis, ind_step $\{i \leftarrow i@p3\}$, induction $\{n \leftarrow i,\ \text{prop} \leftarrow \text{S1}\}$

End main

top: **Module**

Proof

Using main, numeric_types, numeric_types_tcc_proofs, natinduction_tcc, sigmaprops_tcc, natprops_tcc, algorithm_tcc, arithmetics_tcc, absolutes_tcc, sums_tcc, summations_tcc, juggle_tcc

rearrange_delta_TCC1_PROOF: **Prove** rearrange_delta_TCC1 **from** C0

rearrange_delta_TCC2_PROOF: **Prove** rearrange_delta_TCC2 **from** npos

step2_TCC1_PROOF: **Prove** step2_TCC1 **from** npos

culmination_TCC1_PROOF: **Prove** culmination_TCC1 **from** npos

l2_TCC1_PROOF: **Prove** l2_TCC1 **from** npos

l5_TCC1_PROOF: **Prove** l5_TCC1 **from** npos

abs_recip_TCC2_PROOF: **Prove** abs_recip_TCC2 **from** $|a|$ $\{a \leftarrow b\}$

C6_TCC1_PROOF: **Prove** C6_TCC1 **from** C0

End top

step1_proof: **Prove** step1 **from**
mult_mon2
$\{x \leftarrow 2 * (\epsilon + \rho * S) + 2 * m * \Delta/(n-m) + n * \rho * R/(n-m)$
$+ \rho * \Delta$
$+ n * \rho * \Sigma/(n-m),$
$y \leftarrow \delta,$
$z \leftarrow n - m\},$
mult_div $\{x \leftarrow 2 * m * \Delta,\ y \leftarrow n - m\}$,
mult_div $\{x \leftarrow n * \rho * R,\ y \leftarrow n - m\}$,
mult_div $\{x \leftarrow n * \rho * \Sigma,\ y \leftarrow n - m\}$,
C0

step2: **Lemma**
$\delta * n \geq \delta * m + 2 * (\epsilon + \rho * S) * (n-m) + 2 * m * \Delta + n * \rho * R$
$+ \rho * \Delta * (n-m)$
$+ n * \rho * \Sigma$
$\supset \delta \geq \delta * m/n + 2 * (\epsilon + \rho * S) * (n-m)/n + 2 * m * \Delta/n$
$+ \rho * R$
$+ \rho * \Delta * (n-m)/n$
$+ \rho * \Sigma$

step2_proof: **Prove** step2 **from**
div_mon2
$\{z \leftarrow n,$
$y \leftarrow \delta * n,$
x
$\leftarrow \delta * m + 2 * (\epsilon + \rho * S) * (n-m) + 2 * m * \Delta + n * \rho * R$
$+ \rho * \Delta * (n-m)$
$+ n * \rho * \Sigma\},$
npos

final: **Prove** rearrange_delta **from** step1, step2

End juggle

juggle_tcc: **Module**

Using juggle

Exporting all with juggle

Theory

y: **Var** number

rearrange_delta_TCC1: **Formula** $((n-m) \neq 0)$

rearrange_delta_TCC2: **Formula**
$(\delta \geq 2 * (\epsilon + \rho * S) + 2 * m * \Delta/(n-m) + n * \rho * R/(n-m)$
$+ \rho * \Delta$
$+ n * \rho * \Sigma/(n-m))$
$\supset (n \neq 0)$

mult_div_TCC1: **Formula** $(y \neq 0) \supset (y \neq 0)$

step2_TCC1: **Formula**
$(\delta * n \geq \delta * m + 2 * (\epsilon + \rho * S) * (n-m) + 2 * m * \Delta + n * \rho * R$
$+ \rho * \Delta * (n-m)$
$+ n * \rho * \Sigma)$
$\supset (n \neq 0)$

Proof

rearrange_delta_TCC1_PROOF: **Prove** rearrange_delta_TCC1

rearrange_delta_TCC2_PROOF: **Prove** rearrange_delta_TCC2

mult_div_TCC1_PROOF: **Prove** mult_div_TCC1

step2_TCC1_PROOF: **Prove** step2_TCC1

End juggle_tcc

End summations_tcc

juggle: **Module**

Using algorithm

Exporting with algorithm

Theory

rearrange_delta: **Lemma**

$$\begin{aligned}&\delta \geq 2 * (\epsilon + \rho * S) + 2 * m * \Delta/(n-m) + n * \rho * R/(n-m)\\ &\qquad\qquad + \rho * \Delta\\ &\qquad\quad + n * \rho * \Sigma/(n-m)\\ &\quad \supset \delta \geq ((\delta + 2 * \Delta) * m + 2 * (\epsilon + \rho * S + \tfrac{\rho}{2} * \Delta) * (n-m))/n\\ &\qquad\qquad + \rho * R\\ &\qquad\quad + \rho * \Sigma\end{aligned}$$

Proof

x, y, z: **Var** number

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

mult_div_proof: **Prove** mult_div

step1: **Lemma**

$$\begin{aligned}&\delta \geq 2 * (\epsilon + \rho * S) + 2 * m * \Delta/(n-m) + n * \rho * R/(n-m)\\ &\qquad\qquad + \rho * \Delta\\ &\qquad\quad + n * \rho * \Sigma/(n-m)\\ &\quad \supset \delta * (n-m)\\ &\qquad \geq 2 * (\epsilon + \rho * S) * (n-m) + 2 * m * \Delta + n * \rho * R\\ &\qquad\qquad + \rho * \Delta * (n-m)\\ &\qquad\quad + n * \rho * \Sigma\end{aligned}$$

l5_proof: **Prove** l5 **from**
l2,
l3,
l4,
div_mon2
$\{x \leftarrow \sum_1^m((\lambda\ r \rightarrow \text{number} : |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)})|))$
$+ \sum_{m+1}^n((\lambda\ r \rightarrow \text{number} : |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)})|)),$
$y \leftarrow (\delta + 2 * \Delta) * m + 2 * (\rho * S + \epsilon + \frac{\rho}{2} * \Delta) * (n - m),$
$z \leftarrow n\},$
npos

culm_proof: **Prove** culmination **from** lemma6def, l5, S1A $\{i \leftarrow i + 1\}$

End summations

summations_tcc: **Module**

Using summations

Exporting all with summations

Theory

p: **Var** naturalnumber

q: **Var** naturalnumber

T: **Var** number

i: **Var** naturalnumber

pp: **Var** naturalnumber

y: **Var** number

x: **Var** number

culmination_TCC1: **Formula**
$(\text{nonfaulty}(p, i+1) \wedge \text{nonfaulty}(q, i+1) \wedge T \in R^{(i+1)})$
$\wedge (\text{S1}(i) \wedge \text{S1A}(i+1) \wedge \text{S1C}(p, q, i))$
$\supset (n \neq 0)$

l2_TCC1: **Formula** $(n \neq 0)$

l5_TCC1: **Formula**
$(\text{S1}(i) \wedge \text{S1A}(i+1)$
$\wedge \text{S1C}(p, q, i) \wedge \text{nonfaulty}(p, i+1) \wedge \text{nonfaulty}(q, i+1))$
$\supset (n \neq 0)$

Proof

culmination_TCC1_PROOF: **Prove** culmination_TCC1

l2_TCC1_PROOF: **Prove** l2_TCC1

l5_TCC1_PROOF: **Prove** l5_TCC1

l2_proof: **Prove** l2 **from**
l1,
split_mean
$\{i \leftarrow 1,$
$j \leftarrow n,$
$k \leftarrow m,$
F
$\leftarrow (\lambda\, r \rightarrow \text{number}:$
$|c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)})|)\},$
C0

bound_faulty: **Lemma**
$\text{S1A}(i+1)$
$\wedge\, \text{S1C}(p,q,i)$
$\wedge\, 1 \leq r \wedge r \leq m \wedge \text{nonfaulty}(p, i+1) \wedge \text{nonfaulty}(q, i+1)$
$\supset |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)})| < \delta + 2 * \Delta$

bound_faulty_proof: **Prove** bound_faulty **from**
lemma5def $\{T \leftarrow T^{(i+1)}\}$, Ti_in_S

l3_proof: **Prove** l3 **from**
sum_bound
$\{F \leftarrow (\lambda\, r \rightarrow \text{number}:$
$|c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)})|),$
$x \leftarrow \delta + 2 * \Delta,$
$i \leftarrow 1,$
$j \leftarrow m\},$
bound_faulty $\{r \leftarrow \text{pp@}p1\}$

S2_pqr: **Lemma** $\text{S2}(p,i) \wedge \text{S2}(q,i) \wedge \text{S2}(r,i)$

S2_pqr_proof: **Prove** S2_pqr **from**
Theorem_2, Theorem_2 $\{p \leftarrow q\}$, Theorem_2 $\{p \leftarrow r\}$

bound_nonfaulty: **Lemma**
$\text{S1}(i) \wedge \text{S1A}(i+1)$
$\wedge\, \text{S1C}(p,q,i)$
$\wedge\, m+1 \leq r$
$\wedge\, r \leq n \wedge \text{nonfaulty}(p, i+1) \wedge \text{nonfaulty}(q, i+1)$
$\supset |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)})|$
$< 2 * (\rho * S + \epsilon + \frac{\rho}{2} * \Delta)$

bound_nonfaulty_proof: **Prove** bound_nonfaulty **from**
S1A $\{i \leftarrow i+1\}$,
S1A_lemma,
S1A,
nonfx,
nonfx $\{p \leftarrow q\}$,
S1 $\{q \leftarrow r\}$,
S1 $\{p \leftarrow q,\ q \leftarrow r\}$,
S2_pqr,
lemma4def $\{T \leftarrow T^{(i+1)}\}$,
Ti_in_S

l4_proof: **Prove** l4 **from**
sum_bound
$\{F \leftarrow (\lambda\, r \rightarrow \text{number}:$
$|c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)})|),$
$x \leftarrow 2 * (\rho * S + \epsilon + \frac{\rho}{2} * \Delta),$
$i \leftarrow m+1,$
$j \leftarrow n\},$
bound_nonfaulty $\{r \leftarrow \text{pp@}p1\}$,
C0

summations: **Module**

Using algorithm, sums, lemma4, lemma5, lemma6

Exporting with algorithm

Theory

p, q, r: **Var** proc

T: **Var** clocktime

i: **Var** period

culmination: **Lemma**

$$\begin{aligned}
&\mathrm{S1}(i) \wedge \mathrm{S1A}(i+1) \wedge \mathrm{S1C}(p,q,i) \\
&\quad \supset (\mathrm{nonfaulty}(p, i+1) \wedge \mathrm{nonfaulty}(q, i+1) \wedge T \in R^{(i+1)} \\
&\qquad \supset \mathrm{skew}(p, q, T, i+1) \\
&\qquad\quad \leq ((\delta + 2 * \Delta) * m + 2 * (\rho * S + \epsilon + \tfrac{\rho}{2} * \Delta) * (n-m))/n \\
&\qquad\qquad + \rho * R \\
&\qquad\quad + \rho * \Sigma)
\end{aligned}$$

Proof

l1: **Lemma** $|c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)})|$

$$\leq \bigoplus_1^n ((\lambda\, r \rightarrow \mathrm{number} : |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)})|))$$

l2: **Lemma** $|c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)})|$

$$\begin{aligned}
&\leq (\textstyle\sum_1^m ((\lambda\, r \rightarrow \mathrm{number} : \\
&\qquad |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)})|)) \\
&\qquad + \textstyle\sum_{m+1}^n ((\lambda\, r \rightarrow \mathrm{number} : \\
&\qquad\quad |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)})|))) \\
&\quad /n
\end{aligned}$$

l3: **Lemma** $\mathrm{S1A}(i+1)$

$$\begin{aligned}
&\wedge \mathrm{S1C}(p,q,i) \wedge \mathrm{nonfaulty}(p, i+1) \wedge \mathrm{nonfaulty}(q, i+1) \\
&\supset \textstyle\sum_1^m ((\lambda\, r \rightarrow \mathrm{number} : \\
&\qquad |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)})|)) \\
&\quad \leq (\delta + 2 * \Delta) * m
\end{aligned}$$

l4: **Lemma** $\mathrm{S1}(i)$

$$\begin{aligned}
&\wedge \mathrm{S1A}(i+1) \\
&\quad \wedge \mathrm{S1C}(p,q,i) \wedge \mathrm{nonfaulty}(p, i+1) \wedge \mathrm{nonfaulty}(q, i+1) \\
&\supset \textstyle\sum_{m+1}^n ((\lambda\, r \rightarrow \mathrm{number} : \\
&\qquad |c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)})|)) \\
&\quad \leq 2 * (\rho * S + \epsilon + \tfrac{\rho}{2} * \Delta) * (n-m)
\end{aligned}$$

l5: **Lemma** $\mathrm{S1}(i)$

$$\begin{aligned}
&\wedge \mathrm{S1A}(i+1) \\
&\quad \wedge \mathrm{S1C}(p,q,i) \wedge \mathrm{nonfaulty}(p, i+1) \wedge \mathrm{nonfaulty}(q, i+1) \\
&\supset |c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)})| \\
&\quad \leq ((\delta + 2 * \Delta) * m + 2 * (\rho * S + \epsilon + \tfrac{\rho}{2} * \Delta) * (n-m))/n
\end{aligned}$$

l1_proof: **Prove** l1 **from**

$\Delta_p^{(i)}$,

$\Delta_p^{(i)}$ $\{p \leftarrow q\}$,

rearrange_sum

$\{x \leftarrow c_p^{(i)}(T^{(i+1)}),$

$y \leftarrow c_q^{(i)}(T^{(i+1)}),$

$F \leftarrow (\lambda\, r \rightarrow \mathrm{number} : \bar{\Delta}_{r\,p}^{(i)}),$

$G \leftarrow (\lambda\, r \rightarrow \mathrm{number} : \bar{\Delta}_{r\,q}^{(i)}),$

$i \leftarrow 1,$

$j \leftarrow n\},$

abs_mean

$\{i \leftarrow 1,$

$j \leftarrow n,$

$F \leftarrow (\lambda\, r \rightarrow \mathrm{number} : x@p3 + \bar{\Delta}_{r\,p}^{(i)} - (y@p3 + \bar{\Delta}_{r\,q}^{(i)}))\},$

npos

sub_A_proof: **Prove** sublemma_A **from**
$T \in R^{(i)}$,
rearrange_alt
$\{x \leftarrow c_p^{(i)}(T),$
$y \leftarrow c_q^{(i)}(T),$
$u \leftarrow c_p^{(i)}(T^{(i)}),$
$v \leftarrow \Pi$@p1,
$w \leftarrow c_q^{(i)}(T^{(i)})\},$
lemma2d $\{\Pi \leftarrow \Pi$@p1$\}$,
lemma2d $\{p \leftarrow q,\ \Pi \leftarrow \Pi$@p1$\}$,
sublemma1 $\{\Pi \leftarrow \Pi$@p1$\}$

sublemma2: **Lemma** skew$(p, q, T, i+1) = |c_p^{(i)}(T + \Delta_p^{(i)}) - c_q^{(i)}(T + \Delta_q^{(i)})|$

sub2_proof: **Prove** sublemma2 **from** clock_prop, clock_prop $\{p \leftarrow q\}$

lemma6_proof: **Prove** lemma6def **from**
sublemma_A $\{i \leftarrow i+1\}$,
sublemma2 $\{T \leftarrow T^{(i+1)}\}$,
rearrange
$\{x \leftarrow c_p^{(i)}(T^{(i+1)} + \Delta_p^{(i)}),$
$y \leftarrow c_q^{(i)}(T^{(i+1)} + \Delta_q^{(i)}),$
$u \leftarrow c_p^{(i)}(T^{(i+1)}),$
$v \leftarrow \Delta_p^{(i)},$
$w \leftarrow c_q^{(i)}(T^{(i+1)}),$
$z \leftarrow \Delta_q^{(i)}\},$
lemma2c $\{T \leftarrow T^{(i+1)},\ \Pi \leftarrow \Delta_p^{(i)}\}$,
lemma2c $\{T \leftarrow T^{(i+1)},\ \Pi \leftarrow \Delta_q^{(i)},\ p \leftarrow q\}$,
$C_p^{(i)}$ $\{i \leftarrow i+1\}$,
$C_p^{(i)}$ $\{i \leftarrow i+1,\ p \leftarrow q\}$,
S2,
S2 $\{p \leftarrow q\}$,
Theorem_2,
Theorem_2 $\{p \leftarrow q\}$,
mult_mon2 $\{x \leftarrow |\Delta_p^{(i)}|,\ y \leftarrow \Sigma,\ z \leftarrow \frac{\rho}{2}\}$,
mult_mon2 $\{x \leftarrow |\Delta_q^{(i)}|,\ y \leftarrow \Sigma,\ z \leftarrow \frac{\rho}{2}\}$,
rho_pos,
Ti_in_S,
C2

End lemma6

lemma5proof: **Prove** lemma5def **from**
rearrange2
$\{a \leftarrow c_p^{(i)}(T),$
$b \leftarrow c_q^{(i)}(T),$
$x \leftarrow \bar{\Delta}_{r\,p}^{(i)},$
$y \leftarrow \bar{\Delta}_{r\,q}^{(i)}\},$
D2bar_prop $\{p \leftarrow r,\ q \leftarrow p\}$,
D2bar_prop $\{p \leftarrow r,\ q \leftarrow q\}$,
inRS,
S1C,
nonfx,
nonfx $\{p \leftarrow q\}$

End lemma5

lemma6: **Module**

Using algorithm, clockprops, lemma2

Theory

p, q: **Var** proc

i: **Var** period

T, Π: **Var** clocktime

sublemma_A: **Lemma**
nonfaulty$(p, i) \wedge$ nonfaulty$(q, i) \wedge T \in R^{(i)}$
$\supset$ skew$(p, q, T, i) \leq$ skew$(p, q, T^{(i)}, i) + \rho * R$

lemma6def: **Lemma**
nonfaulty$(p, i+1) \wedge$ nonfaulty$(q, i+1) \wedge T \in R^{(i+1)}$
$\supset$ skew$(p, q, T, i+1)$
$\leq |c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - (c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)})| + \rho * R + \rho * \Sigma$

Proof

sublemma1: **Lemma** $0 \leq \Pi \wedge \Pi \leq R \supset 2 * \frac{\rho}{2} * \Pi \leq \rho * R$

sub1_proof: **Prove** sublemma1 **from**
mult_mon2 $\{x \leftarrow \Pi,\ y \leftarrow R,\ z \leftarrow \frac{\rho}{2}\}$, rho_pos

lemma4_proof: **Prove** lemma4def **from**
rearrange3
$\{x \leftarrow c_p^{(i)}(T) + \bar{\Delta}_{r\,p}^{(i)},$
$y \leftarrow c_q^{(i)}(T) + \bar{\Delta}_{r\,q}^{(i)},$
$u \leftarrow c_q^{(i)}(T + \Delta_{r\,q}^{(i)}),$
$v \leftarrow c_p^{(i)}(T + \Delta_{r\,p}^{(i)}),$
$w \leftarrow c_r^{(i)}(T)\},$
sublemma1,
sublemma1 $\{p \leftarrow q\}$,
lemma2x,
lemma2x $\{p \leftarrow q\}$,
lemma3def $\{q \leftarrow r\}$,
lemma3def $\{p \leftarrow q,\ q \leftarrow r\}$,
S1C_lemma

End lemma4

lemma5: **Module**

Using algorithm, clockprops

Theory

p, q, r: **Var** proc

T: **Var** clocktime

i, j: **Var** period

lemma5def: **Lemma**
S1C$(p, q, i) \wedge$ nonfaulty$(p, i+1) \wedge$ nonfaulty$(q, i+1) \wedge T \in S^{(i)}$
$\supset |c_p^{(i)}(T) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T) + \bar{\Delta}_{r\,q}^{(i)})| < \delta + 2 * \Delta$

Proof

a, b, x, y: **Var** clocktime

rearrange1: **Lemma** $(a + x) - (b + y) = (a - b) + x - y$

rearrange1_proof: **Prove** rearrange1

rearrange2: **Lemma** $|(a + x) - (b + y)| \leq |a - b| + |x| + |y|$

rearrange2_proof: **Prove** rearrange2 **from**
rearrange1, abs_ax8, abs_ax2 $\{x \leftarrow (a - b),\ y \leftarrow (x - y)\}$

lemma4: **Module**

Using algorithm, lemma1, lemma2, lemma3

Theory

p, q, r: **Var** proc

i: **Var** period

T: **Var** clocktime

lemma4def: **Lemma**
$$\mathrm{S1C}(q,r,i) \land \mathrm{S1C}(p,q,i) \land \mathrm{S1C}(p,r,i) \land \mathrm{S2}(p,i) \land \mathrm{S2}(q,i) \land \mathrm{S2}(r,i) \land \mathrm{nonfaulty}(p,i+1) \land \mathrm{nonfaulty}(q,i+1) \land \mathrm{nonfaulty}(r,i+1) \land T \in S^{(i)}$$
$$\supset |c_p^{(i)}(T) + \bar{\Delta}_{r\,p}^{(i)} - (c_q^{(i)}(T) + \bar{\Delta}_{r\,q}^{(i)})| < 2 * (\epsilon + \rho * S + \tfrac{\rho}{2} * \Delta)$$

Proof

T_0, T_1, T_2: **Var** clocktime

Π: **Var** realtime

u, v, w, x, y, z: **Var** number

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

rearrange1_proof: **Prove** rearrange1

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

rearrange2_proof: **Prove** rearrange2 **from**
abs_ax2c
$\{w \leftarrow (u-y),$
$x \leftarrow (x-v),$
$y \leftarrow (v-w),$
$z \leftarrow (w-u)\}$,
abs_ax3 $\{x \leftarrow (v-x)\}$,
abs_ax3 $\{x \leftarrow (u-w)\}$

rearrange3: **Lemma** $|x-y| \le |u-y| + |v-x| + |v-w| + |u-w|$

rearrange3_proof: **Prove** rearrange3 **from** rearrange1, rearrange2

sublemma1: **Lemma**
$$\mathrm{S1C}(p,r,i) \land \mathrm{S2}(p,i) \land \mathrm{nonfaulty}(p,i+1) \land \mathrm{nonfaulty}(r,i+1) \supset \bar{\Delta}_{r\,p}^{(i)} = \Delta_{r\,p}^{(i)}$$

sublemma1_proof: **Prove** sublemma1 **from**
lemma1def $\{q \leftarrow r\}$, $\bar{\Delta}_{r,p}^{(i)}$, A2_aux

lemma2x: **Lemma**
$$\mathrm{S1C}(p,r,i) \land \mathrm{S2}(p,i) \land \mathrm{nonfaulty}(p,i+1) \land \mathrm{nonfaulty}(r,i+1) \land T \in S^{(i)}$$
$$\supset |c_p^{(i)}(T + \Delta_{r\,p}^{(i)}) - (c_p^{(i)}(T) + \Delta_{r\,p}^{(i)})| \le \tfrac{\rho}{2} * \Delta$$

lemma2x_proof: **Prove** lemma2x **from**
lemma2c $\{\Pi \leftarrow \Delta_{r\,p}^{(i)}\}$,
lemma1def $\{q \leftarrow r\}$,
C2and3,
mult_mon2 $\{z \leftarrow \frac{\rho}{2},\ x \leftarrow |\Delta_{r\,p}^{(i)}|,\ y \leftarrow \Delta\}$,
rho_pos

lemma2d_proof: **Prove** lemma2d **from**
A1,
goodclock
$\{T_0 \leftarrow A_p^{(0)}(T^{(0)}),$
$T_N \leftarrow A_p^{(i)}(T^{(i+1)}),$
$T_1 \leftarrow A_p^{(i)}(T^{(i)} + \Pi),$
$T_2 \leftarrow A_p^{(i)}(T^{(i)})\},$
$c_p^{(i)}(T)\ \{T \leftarrow T^{(i)}\},$
$c_p^{(i)}(T)\ \{T \leftarrow T^{(i)} + \Pi\},$
posR,
pos_abs $\{x \leftarrow \Pi\}$,
lower_bound,
lower_bound $\{\Pi \leftarrow 0\}$,
T_next

End lemma2

lemma3: **Module**

Using algorithm, lemma2

Theory

p, q: **Var** proc

i: **Var** period

T, T_0, T_1, T_2: **Var** clocktime

Π: **Var** realtime

lemma3def: **Lemma**
S1C(p, q, i)
$\wedge$ S2$(p, i) \wedge$ nonfaulty$(p, i+1) \wedge$ nonfaulty$(q, i+1) \wedge T \in S^{(i)}$
$\supset |c_p^{(i)}(T + \Delta_{qp}^{(i)}) - c_q^{(i)}(T)| < \epsilon + \rho * S$

Proof

lemma3_proof: **Prove** lemma3def **from**
A2,
rearrange_alt
$\{x \leftarrow c_p^{(i)}(T + \Delta_{qp}^{(i)}),$
$y \leftarrow c_q^{(i)}(T),$
$u \leftarrow c_p^{(i)}(T_0@p1 + \Delta_{qp}^{(i)}),$
$v \leftarrow T - T_0@p1,$
$w \leftarrow c_q^{(i)}(T_0@p1)\},$
lemma2b $\{T \leftarrow T_0@p1,\ \Phi \leftarrow \Delta_{qp}^{(i)},\ \Pi \leftarrow T - T_0@p1\}$,
lemma2c $\{p \leftarrow q,\ T \leftarrow T_0@p1,\ \Pi \leftarrow T - T_0@p1\}$,
nonfx,
nonfx $\{p \leftarrow q\}$,
mult_mon2 $\{x \leftarrow |T - T_0@p1|,\ y \leftarrow S,\ z \leftarrow \frac{\rho}{2}\}$,
rho_pos,
in_S_lemma $\{T_1 \leftarrow T,\ T_2 \leftarrow T_0@p1\}$

End lemma3

lemma2: **Module**

Using algorithm, clockprops

Theory

p, q, r: **Var** proc

i: **Var** period

T: **Var** clocktime

Π, Φ: **Var** realtime

lemma2def: **Lemma**
nonfaulty$(p, i+1)$
$\wedge A_p^{(i)}(T) \leq A_p^{(i+1)}(T^{(i+2)})$
$\wedge A_p^{(0)}(T^{(0)}) \leq A_p^{(i)}(T)$
$\wedge A_p^{(i)}(T+\Pi) \leq A_p^{(i+1)}(T^{(i+2)}) \wedge A_p^{(0)}(T^{(0)}) \leq A_p^{(i)}(T+\Pi)$
$\supset |c_p^{(i)}(T+\Pi) - (c_p^{(i)}(T)+\Pi)| \leq \frac{\rho}{2} * |\Pi|$

lemma2a: **Lemma**
nonfaulty$(p, i+1) \wedge |\Pi + \Phi| \leq R - S \wedge |\Phi| \leq R - S \wedge T \in S^{(i)}$
$\supset |c_p^{(i)}(T+\Phi+\Pi) - (c_p^{(i)}(T+\Phi)+\Pi)| \leq \frac{\rho}{2} * |\Pi|$

lemma2b: **Lemma**
nonfaulty$(p, i+1) \wedge |\Phi| \leq S \wedge |\Pi| \leq S \wedge T \in S^{(i)}$
$\supset |c_p^{(i)}(T+\Phi+\Pi) - (c_p^{(i)}(T+\Phi)+\Pi)| \leq \frac{\rho}{2} * |\Pi|$

lemma2c: **Lemma**
nonfaulty$(p, i+1) \wedge |\Pi| \leq S \wedge T \in S^{(i)}$
$\supset |c_p^{(i)}(T+\Pi) - (c_p^{(i)}(T)+\Pi)| \leq \frac{\rho}{2} * |\Pi|$

lemma2d: **Lemma**
nonfaulty$(p, i) \wedge 0 \leq \Pi \wedge \Pi \leq R$
$\supset |c_p^{(i)}(T^{(i)}+\Pi) - (c_p^{(i)}(T^{(i)})+\Pi)| \leq \frac{\rho}{2} * \Pi$

Proof

lemma2_proof: **Prove** lemma2def **from**
A1 $\{i \leftarrow i+1\}$,
goodclock
$\{T_0 \leftarrow A_p^{(0)}(T^{(0)})$,
$T_N \leftarrow A_p^{(i+1)}(T^{(i+2)})$,
$T_2 \leftarrow A_p^{(i)}(T)$,
$T_1 \leftarrow A_p^{(i)}(T+\Pi)\}$,
$c_p^{(i)}(T)$,
$c_p^{(i)}(T)$ $\{T \leftarrow T+\Pi\}$

lemma2a_proof: **Prove** lemma2a **from**
lemma2def $\{T \leftarrow T+\Phi\}$,
upper_bound $\{\Pi \leftarrow \Phi+\Pi\}$,
lower_bound2 $\{\Pi \leftarrow \Phi+\Pi\}$,
upper_bound $\{\Pi \leftarrow \Phi\}$,
lower_bound2 $\{\Pi \leftarrow \Phi\}$

lemma2b_proof: **Prove** lemma2b **from**
lemma2a,
abs_ax1 $\{x \leftarrow \Pi\}$,
abs_ax2 $\{x \leftarrow \Phi,\ y \leftarrow \Pi\}$,
C1,
posS,
posR

lemma2c_proof: **Prove** lemma2c **from** lemma2b $\{\Phi \leftarrow 0\}$, abs_ax0, posS

bounds_proof: **Prove** bounds **from**
  upper_bound $\{\Pi \leftarrow 0,\ T \leftarrow T^{(i+1)}\}$,
  lower_bound2 $\{\Pi \leftarrow 0,\ T \leftarrow T^{(i+1)}\}$,
  abs_ax0,
  SinR,
  Ti_in_S

rmproof: **Prove** nonfx **from**
  A1 $\{i \leftarrow i+1\}$,
  A1,
  gc_prop
    $\{T_0 \leftarrow A_p^{(0)}(T^{(0)})$,
     $T_N \leftarrow A_p^{(i+1)}(T^{(i+2)})$,
     $T \leftarrow A_p^{(i)}(T^{(i+1)})\}$,
  bounds

S1A_lemma_proof: **Prove** S1A_lemma **from**
  S1A $\{i \leftarrow i,\ r \leftarrow r@p1\}$,
  S1A $\{i \leftarrow i+1,\ r \leftarrow r@p1\}$,
  nonfx $\{p \leftarrow r@p1\}$

End clockprops

lemma1: **Module**

Using algorithm, lemma2

Theory

p, q: **Var** proc

i: **Var** period

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

Proof

lemma1_proof: **Prove** lemma1def **from**
  A2,
  lemma2c $\{\Pi \leftarrow \Delta_{q\,p}^{(i)},\ T \leftarrow T_0@p1\}$,
  S1C $\{T \leftarrow T_0@p1\}$,
  abs_ax4 $\{x \leftarrow c_p^{(i)}(T_0@p1),\ y \leftarrow c_q^{(i)}(T_0@p1)\}$,
  abs_ax4 $\{x \leftarrow c_p^{(i)}(T_0@p1 + \Pi@\mathrm{p}2),\ y \leftarrow c_p^{(i)}(T_0@p1) + \Pi@\mathrm{p}2\}$,
  abs_ax2b
    $\{x \leftarrow y@p5 - x@p5$,
     $y \leftarrow y@p4 - x@p4$,
     $z \leftarrow x@p5 - y@p4\}$,
  nonfx,
  nonfx $\{p \leftarrow q\}$,
  inRS $\{T \leftarrow T_0@p1\}$,
  mult_mon2 $\{x \leftarrow |\Delta_{q\,p}^{(i)}|,\ y \leftarrow S,\ z \leftarrow \frac{\rho}{2}\}$,
  rho_pos,
  C4

End lemma1

clockprops: **Module**

Using clocks, algorithm, natinduction

Theory

$T, T_0, T_1, T_2, T_N, \Pi$: **Var** clocktime

p, q: **Var** proc

i: **Var** period

upper_bound: **Lemma**
$T \in S^{(i)} \wedge |\Pi| \leq R - S \supset A_p^{(i)}(T + \Pi) \leq A_p^{(i+1)}(T^{(i+2)})$

lower_bound: **Lemma** $0 \leq \Pi \supset A_p^{(0)}(T^{(0)}) \leq A_p^{(i)}(T^{(i)} + \Pi)$

lower_bound2: **Lemma** $T \in S^{(i)} \wedge |\Pi| \leq R - S \supset A_p^{(0)}(T^{(0)}) \leq A_p^{(i)}(T + \Pi)$

adj_always_pos: **Lemma** $A_p^{(i)}(T^{(i)}) \geq T^0 + C_p^{(0)}$

nonfx: **Lemma** nonfaulty$(p, i + 1) \supset$ nonfaulty(p, i)

S1A_lemma: **Lemma** S1A$(i + 1) \supset$ S1A(i)

Proof

i2R: **Lemma** $T^{(i+2)} = T^{(i)} + 2 * R$

i2R_proof: **Prove** i2R **from** $T^{(i)}$ $\{i \leftarrow i + 2\}$, $T^{(i)}$

upper_bound_proof: **Prove** upper_bound **from**
$T \in S^{(i)}$,
i2R,
abs_ax6 $\{x \leftarrow \Pi,\ y \leftarrow R - S\}$,
S2,
Theorem_2,
abs_ax6 $\{x \leftarrow C_p^{(i+1)} - C_p^{(i)},\ y \leftarrow \Sigma\}$,
C2

basis: **Lemma** $A_p^{(0)}(T^{(0)}) \geq T^0 + C_p^{(0)}$

basis_proof: **Prove** basis **from** $C_p^{(i)}$ $\{i \leftarrow i + 1\}$, $T^{(i)}$ $\{i \leftarrow 0\}$

small_shift: **Lemma** $C_p^{(i+1)} - C_p^{(i)} \geq -R$

small_shift_proof: **Prove** small_shift **from**
S2, Theorem_2, $|a|$ $\{a \leftarrow C_p^{(i+1)} - C_p^{(i)}\}$, C2, SinR

inductive_step: **Lemma** $A_p^{(i)}(T^{(i)}) \geq T^0 + C_p^{(0)} \supset A_p^{(i+1)}(T^{(i+1)}) \geq T^0 + C_p^{(0)}$

ind_proof: **Prove** inductive_step **from** small_shift, T_next

adj_pos_proof: **Prove** adj_always_pos **from**
induction
$\{n \leftarrow i,$
prop $\leftarrow (\lambda\ i \rightarrow \text{bool} : A_p^{(i)}(T^{(i)}) \geq T^0 + C_p^{(0)})\}$,
basis,
inductive_step $\{i \leftarrow i@p1\}$

lower_bound_proof: **Prove** lower_bound **from**
adj_always_pos, $T^{(i)}$ $\{i \leftarrow 0\}$, $C_p^{(i)}$ $\{i \leftarrow i + 1\}$

lower_bound2_proof: **Prove** lower_bound2 **from**
lower_bound $\{\Pi \leftarrow T - T^{(i)} + \Pi@\text{c}\}$, $T \in S^{(i)}$, $|a|$ $\{a \leftarrow \Pi\}$, SinR

gc_prop: **Lemma**
goodclock$(p, T_0, T_N) \wedge T_0 \leq T \wedge T \leq T_N \supset$ goodclock(p, T_0, T)

gc_proof: **Prove** gc_prop **from**
goodclock $\{T_1 \leftarrow T_1@p2,\ T_2 \leftarrow T_2@p2\}$, goodclock $\{T_N \leftarrow T\}$

bounds: **Lemma** $A_p^{(0)}(T^{(0)}) \leq A_p^{(i)}(T^{(i+1)}) \wedge A_p^{(i)}(T^{(i+1)}) \leq A_p^{(i+1)}(T^{(i+2)})$

clock_proof: **Prove** clock_prop **from**
$c_p^{(i)}(T)$,
$c_p^{(i)}(T)$ $\{T \leftarrow T + \Delta_p^{(i)}\}$,
$c_p^{(i)}(T)$ $\{i \leftarrow i+1\}$,
$C_p^{(i)}$ $\{i \leftarrow i+1\}$,
nat_invariant {nat_var $\leftarrow i$}

D2bar_prop_proof: **Prove** D2bar_prop **from**
$\bar{\Delta}_{r,p}^{(i)}$ $\{r \leftarrow p,\ p \leftarrow q\}$, abs_ax0

S1C_lemma_proof: **Prove** S1C_lemma **from**
S1C $\{T \leftarrow T@p2\}$,
S1C $\{p \leftarrow q,\ q \leftarrow p\}$,
abs_ax4 $\{x \leftarrow c_q^{(i)}(T@p2),\ y \leftarrow c_p^{(i)}(T@p2)\}$

Theorem_2_proof: **Prove** Theorem_2 **from**
S2,
$C_p^{(i)}$ $\{i \leftarrow i+1\}$,
D2bar_prop $\{p \leftarrow \mathrm{pp}@p6,\ q \leftarrow p\}$,
$\Delta_p^{(i)}$,
C0,
mean_bound
$\{i \leftarrow 1,$
$j \leftarrow n,$
$x \leftarrow \Delta,$
$F \leftarrow (\lambda\, r \rightarrow \text{number} : |\bar{\Delta}_{r\,p}^{(i)}|)\}$,
abs_mean $\{i \leftarrow 1,\ j \leftarrow n,\ F \leftarrow (\lambda\, r \rightarrow \text{number} : \bar{\Delta}_{r\,p}^{(i)})\}$,
C3

End algorithm

algorithm_tcc: **Module**

Using algorithm

Exporting all with algorithm

Theory

i: **Var** naturalnumber

p: **Var** naturalnumber

pp: **Var** naturalnumber

T: **Var** number

Corr_TCC1: **Formula** $(i > 0) \supset \text{second\_arg}(p, i) > \text{second\_arg}(p, \text{pred}(i))$

C6_TCC1: **Formula** $((n - m) \neq 0)$

Proof

Corr_TCC1_PROOF: **Prove** Corr_TCC1

C6_TCC1_PROOF: **Prove** C6_TCC1

End algorithm_tcc

$A_p^{(i)}(T)$: function[proc, period, clocktime → clocktime] ==
  $(\lambda\, p, i, T : T + C_p^{(i)})$

$c_p^{(i)}(T)$: function[proc, period, clocktime → realtime] =
  $(\lambda\, p, i, T : c_p(A_p^{(i)}(T)))$

skew: function[proc, proc, clocktime, period → clocktime] ==
  $(\lambda\, p, q, T, i \rightarrow \text{clocktime} : |c_p^{(i)}(T) - c_q^{(i)}(T)|)$

nonfaulty: function[proc, period → boolean] =
  $(\lambda\, p, i : \text{goodclock}(p, A_p^{(0)}(T^{(0)}), A_p^{(i)}(T^{(i+1)})))$

S1A: function[period → bool] =
  $(\lambda\, i : (\forall\, r : (m + 1 \leq r \wedge r \leq n) \supset \text{nonfaulty}(r, i)))$

S1C: function[proc, proc, period → bool] =
  $(\lambda\, p, q, i :$
    $(\forall\, T :$
      $\text{nonfaulty}(p, i) \wedge \text{nonfaulty}(q, i) \wedge T \in R^{(i)}$
        $\supset \text{skew}(p, q, T, i) \leq \delta))$

S1C_lemma: **Lemma** $\text{S1C}(p, q, i) \supset \text{S1C}(q, p, i)$

S1: function[period → bool] =
  $(\lambda\, i : \text{S1A}(i) \supset (\forall\, p, q : \text{S1C}(p, q, i)))$

S2: function[proc, period → bool] = $(\lambda\, p, i : (|C_p^{(i+1)} - C_p^{(i)}| < \Sigma))$

A0: **Axiom** $\text{skew}(p, q, T^{(0)}, 0) < \delta_0$

A1: **Lemma** $\text{nonfaulty}(p, i) = \text{goodclock}(p, A_p^{(0)}(T^{(0)}), A_p^{(i)}(T^{(i+1)}))$

A2: **Axiom** $\text{nonfaulty}(p, i) \wedge \text{nonfaulty}(q, i) \wedge \text{S1C}(p, q, i) \wedge \text{S2}(p, i)$
  $\supset |\Delta_{q\,p}^{(i)}| \leq S$
  $\wedge\, (\exists\, T_0 : T_0 \in S^{(i)} \wedge |c_p^{(i)}(T_0 + \Delta_{q\,p}^{(i)}) - c_q^{(i)}(T_0)| < \epsilon)$

A2_aux: **Axiom** $\Delta_{p\,p}^{(i)} = 0$

C0: **Axiom** $m < n$

C2: **Axiom** $S \geq \Sigma$

C3: **Axiom** $\Sigma \geq \Delta$

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

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

C6: **Axiom** δ
  $\geq 2 * (\epsilon + \rho * S) + 2 * m * \Delta/(n - m) + n * \rho * R/(n - m)$
    $+ \rho * \Delta$
   $+ n * \rho * \Sigma/(n - m)$

C2and3: **Lemma** $\Delta \leq S$

npos: **Lemma** $n > 0$

clock_prop: **Lemma** $c_p^{(i+1)}(T) = c_p^{(i)}(T + \Delta_p^{(i)})$

D2bar_prop: **Lemma** $|\bar{\Delta}_{p\,q}^{(i)}| < \Delta$

Theorem_1: **Theorem** $\text{S1}(i)$

Theorem_2: **Theorem** $\text{S2}(p, i)$

Proof

A1_proof: **Prove** A1 **from** nonfaulty

C2and3_proof: **Prove** C2and3 **from** C2, C3

npos_proof: **Prove** npos **from** C0

rho_small_proof: **Prove** rho_small **from**
  fraction_invariant $\{$fraction_var $\leftarrow \rho\}$

x, y: **Var** number

diminish: **Lemma** $x > 0 \supset \frac{\rho}{2} * x \leq x$

diminish_proof: **Prove** diminish **from**
  mult_mon2 $\{x \leftarrow \frac{\rho}{2},\ y \leftarrow 1,\ z \leftarrow x\}$, rho_small

monoproof: **Prove** monotonicity **from**
  goodclock,
  diminish $\{x \leftarrow |T_1 - T_2|\}$,
  $|a|$ $\{a \leftarrow c_p(T_1) - c_p(T_2) - (T_1 - T_2)\}$,
  $|a|$ $\{a \leftarrow T_1 - T_2\}$

End clocks

algorithm: **Module**

Using clocks, sums

Exporting second_arg, $C_p^{(i)}, A_p^{(i)}(T), c_p^{(i)}(T)$, nonfaulty, $\Sigma, \Delta, \Delta_p^{(i)}, \Delta_{\star 1,\star 2}^{(\star 3)}, \bar{\Delta}_{r,p}^{(i)}$, skew, S1, S1A, S1C, S2, $\delta, \epsilon, \delta_0, n, m$ **with** clocks

Theory

T, T_0, T_1, X, Π: **Var** clocktime

i: **Var** period

p, q, r: **Var** proc

m, n: proc

$\epsilon, \delta_0, \delta$: posrealtime

Σ, Δ: posclocktime

$\Delta_{\star 1,\star 2}^{(\star 3)}$: function[proc, proc, period $\rightarrow$ clocktime]

$\bar{\Delta}_{r,p}^{(i)}$: function[proc, proc, period $\rightarrow$ clocktime] =
  $(\lambda\, r, p, i :$ **if** $r \neq p \wedge |\Delta_{r\,p}^{(i)}| < \Delta$ **then** $\Delta_{r\,p}^{(i)}$ **else** 0 **end if**$)$

$\Delta_p^{(i)}$: function[proc, period $\rightarrow$ clocktime] =
  $(\lambda\, p, i : \bigoplus_1^n((\lambda\, r \rightarrow \text{number} : \bar{\Delta}_{r\,p}^{(i)})))$

$C_{\star 1}^0$: function[proc $\rightarrow$ clocktime]

second_arg: function[proc, period $\rightarrow$ nat] == $(\lambda\, p, i : i)$

$C_p^{(i)}$: **Recursive** function[proc, period $\rightarrow$ clocktime] =
  $(\lambda\, p, i :$ **if** $i > 0$ **then** $C_p^{(\text{pred}(i))} + \Delta_p^{(\text{pred}(i))}$ **else** C_p^0 **end if**$)$
  by second_arg

Ti_in_R: **Lemma** $T^{(i)} \in R^{(i)}$

$T \in S^{(i)}$: function[clocktime, period $\rightarrow$ boolean] =
$(\lambda\, T, i : (\exists\, \Pi : 0 \leq \Pi \wedge \Pi \leq S \wedge T = T^{(i)} + R - S + \Pi))$

inRS: **Lemma** $T \in S^{(i)} \supset T \in R^{(i)}$

Ti_in_S: **Lemma** $T^{(i+1)} \in S^{(i)}$

in_S_lemma: **Lemma** $(\exists\, i : T_1 \in S^{(i)} \wedge T_2 \in S^{(i)}) \supset |T_1 - T_2| \leq S$

Proof

posS_proof: **Prove** posS **from** posnum_invariant $\{$posnum_var $\leftarrow S\}$

posR_proof: **Prove** posR **from** posnum_invariant $\{$posnum_var $\leftarrow R\}$

SinR_proof: **Prove** SinR **from** C1, posS, posR

Ti_proof: **Prove** Ti_in_R **from**
$T \in R^{(i)}$ $\{T \leftarrow T^{(i)},\ \Pi \leftarrow 0\}$, abs_ax0, posR

inRS_proof: **Prove** inRS **from**
$T \in S^{(i)}$, $T \in R^{(i)}$ $\{\Pi \leftarrow R - S + \Pi@\text{p1}\}$, SinR

T_next_proof: **Prove** T_next **from** $T^{(i)}$, $T^{(i)}$ $\{i \leftarrow i + 1\}$

Ti_in_S_proof: **Prove** Ti_in_S **from**
$T \in S^{(i)}$ $\{\Pi \leftarrow S,\ T \leftarrow T^{(i+1)}\}$, posS, T_next

in_S_proof: **Prove** in_S_lemma **from**
$T \in S^{(i)}$ $\{T \leftarrow T_1\}$,
$T \in S^{(i)}$ $\{T \leftarrow T_2\}$,
abs_ax5 $\{x \leftarrow \Pi@\text{p1},\ y \leftarrow \Pi@\text{p2},\ z \leftarrow S\}$

End time

clocks: **Module**

Using time

Exporting proc, $c_{\star 1}(\star 2)$, ρ, goodclock **with** time

Theory

proc: **Type is** nat

p: **Var** proc

$c_{\star 1}(\star 2)$: function[proc, clocktime $\rightarrow$ realtime]

i: **Var** period

T, T_0, T_1, T_2, T_N: **Var** clocktime

ρ: fraction

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

rho_small: **Lemma** $\frac{\rho}{2} < 1$

goodclock: function[proc, clocktime, clocktime $\rightarrow$ bool] =
$(\lambda\, p, T_0, T_N :$
$(\forall\, T_1, T_2 :$
$T_0 \leq T_1 \wedge T_0 \leq T_2 \wedge T_1 \leq T_N \wedge T_2 \leq T_N$
$\supset |c_p(T_1) - c_p(T_2) - (T_1 - T_2)| \leq \frac{\rho}{2} * |T_1 - T_2|))$

monotonicity: **Theorem**
$(\exists\, T_0, T_N :$
$\text{goodclock}(p, T_0, T_N) \wedge T_0 \leq T_1 \wedge T_0 \leq T_2 \wedge T_1 \leq T_N \wedge T_2 \leq T_N)$
$\supset (T_1 > T_2 \supset c_p(T_1) \geq c_p(T_2))$

Proof

rho_pos_proof: **Prove** rho_pos **from**
fraction_invariant $\{$fraction_var $\leftarrow \rho\}$

sigmaprops_tcc: **Module**

Using sigmaprops

Exporting all with sigmaprops

Theory

F: **Var** function[naturalnumber → number]

n: **Var** naturalnumber

k: **Var** naturalnumber

j: **Var** naturalnumber

i_2: **Var** naturalnumber

i_0: **Var** naturalnumber

i_1: **Var** naturalnumber

i: **Var** naturalnumber

x: **Var** number

sigma_TCC1: **Formula**
$(\neg(n = 0)) \supset \text{middle}(i, n, F) > \text{middle}(i, \text{pred}(n), F)$

revsigma_TCC1: **Formula**
$(\neg(n = 0)) \supset \text{middle}(i, n, F) > \text{middle}(i + 1, \text{pred}(n), F)$

Proof

sigma_TCC1_PROOF: **Prove** sigma_TCC1

revsigma_TCC1_PROOF: **Prove** revsigma_TCC1

End sigmaprops_tcc

time: **Module**

Using absolutes, numeric_types

Exporting clocktime, realtime, posclocktime, posrealtime, period,
$R, S, T^0, T^{(i)}, T \in R^{(i)}, T \in S^{(i)}$ **with** absolutes, numeric_types

Theory

realtime: **Type is** number

clocktime: **Type is** number

posclocktime: **Type is** posnum

posrealtime: **Type is** posnum

period: **Type is** nat

R, S: posclocktime

posR: **Lemma** $0 < R$

posS: **Lemma** $0 < S$

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

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

T^0: clocktime

i: **Var** period

$T^{(i)}$: function[period → clocktime] $= (\lambda\, i : T^0 + i * R)$

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

T, T_1, T_2, Π: **Var** clocktime

$T \in R^{(i)}$: function[clocktime, period → boolean] =
$(\lambda\, T, i : (\exists\, \Pi : 0 \leq \Pi \wedge \Pi \leq R \wedge T = T^{(i)} + \Pi))$

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

bounded: function[nat, nat, function[nat $\rightarrow$ number], number $\rightarrow$ bool] =
$(\lambda\, i, n, F, x : (\forall\, k : i \leq k \wedge k \leq i + \mathrm{pred}(n) \supset F(k) < x))$

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

bounded_proof: **Prove** bounded_lemma **from**
bounded $\{k \leftarrow k@p1\}$, bounded $\{n \leftarrow n+1,\ k \leftarrow k@p1\}$

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

sb_basis_proof: **Prove** sigma_bound_basis **from**
bounded $\{n \leftarrow 1,\ k \leftarrow i\}$,
$\sigma(i, n, F)\ \{n \leftarrow 0\}$,
$\sigma(i, n, F)\ \{n \leftarrow 1\}$

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

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

alt_sigma_bound_one_step_proof: **Prove** alt_sigma_bound_one_step **from**
bounded $\{n \leftarrow n+1,\ k \leftarrow i+n\}$, $\sigma(i, n, F)\ \{n \leftarrow n+1\}$

sigma_split: **Lemma**
$\sigma(i, n+1, F) < x + \sigma(i, n, F) \wedge \sigma(i, n, F) < y$
$\supset \sigma(i, n+1, F) < x + y$

sigma_split_proof: **Prove** sigma_split

alt_sb_step_proof: **Prove** alt_sigma_bound_step **from**
alt_sigma_bound_one_step, sigma_split $\{y \leftarrow n * x\}$

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

sb_step_proof: **Prove** sigma_bound_step **from** alt_sigma_bound_step

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

sb_proof: **Prove** sb **from**
mod_induction1
$\{A \leftarrow (\lambda\, \mathrm{nn} \rightarrow \mathrm{bool} : \mathrm{bounded}(i, \mathrm{nn}, F, x))$,
$B \leftarrow (\lambda\, \mathrm{mm} \rightarrow \mathrm{bool} : \sigma(i, \mathrm{mm}, F) < \mathrm{mm} * x)\}$,
bounded_lemma $\{n \leftarrow j@p1\}$,
sigma_bound_basis,
sigma_bound_step $\{n \leftarrow i@p1\}$

sigma_bound_proof: **Prove** sigma_bound $\{k \leftarrow k@p2\}$ **from** sb, bounded

End sigmaprops

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

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

srb_proof: **Prove** sigma_rev_basis **from**
$\sigma(i,n,F)$ $\{n \leftarrow 0\}$, revsigma $\{n \leftarrow 0\}$

sigma_rev_step: **Lemma**
$(\forall\, i_1 : \sigma(i_1,n,F) = \text{revsigma}(i_1,n,F))$
$\supset (\forall\, i_2 : \sigma(i_2,n+1,F) = \text{revsigma}(i_2,n+1,F))$

srp_proof: **Prove** sigma_rev_step $\{i_1 \leftarrow i_2+1\}$ **from**
revsigma $\{i \leftarrow i_2,\ n \leftarrow n+1\}$,
sigma1 $\{i \leftarrow i_2\}$,
nat_invariant {nat_var $\leftarrow n$}

sigma_rev_proof: **Prove** sigma_rev **from**
induction2
$\{i_1 \leftarrow i_1@p3$,
$i_3 \leftarrow i$,
prop2
$\leftarrow (\lambda\, i, \text{nn} \rightarrow \text{bool} : \sigma(i,\text{nn},F) = \text{revsigma}(i,\text{nn},F))\}$,
sigma_rev_basis $\{i \leftarrow i_0@p1\}$,
sigma_rev_step $\{i_2 \leftarrow i_2@p1,\ n \leftarrow j@p1\}$

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

split_basis_proof: **Prove** split_sigma_basis **from**
$\sigma(i,n,F)$,
$\sigma(i,n,F)$ $\{n \leftarrow 0\}$,
diff $\{m \leftarrow 0\}$,
nat_invariant {nat_var $\leftarrow n$}

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

split_step_proof: **Prove** split_sigma_step **from**
$\sigma(i,n,F)$ $\{n \leftarrow m+1\}$,
sigma_rev $\{i \leftarrow i+m+1,\ n \leftarrow \text{diff}(n,m+1)\}$,
revsigma $\{i \leftarrow i+m,\ n \leftarrow \text{diff}(n,m)\}$,
sigma_rev $\{i \leftarrow i+m,\ n \leftarrow \text{diff}(n,m)\}$,
pred_diff,
diff,
nat_invariant {nat_var $\leftarrow m$}

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

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

sa_basis_proof: **Prove** sigma_abs_basis **from**
$\sigma(i,n,F)$ $\{n \leftarrow 0\}$,
$\sigma(i,n,F)$ $\{n \leftarrow 0,\ F \leftarrow (\lambda\, \text{qq} \rightarrow \text{number} : |F(\text{qq})|)\}$,
abs_ax0

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

sa_step_proof: **Prove** sigma_abs_step **from**
$\sigma(i,n,F)$ $\{n \leftarrow n+1\}$,
$\sigma(i,n,F)$ $\{n \leftarrow n+1,\ F \leftarrow (\lambda\, \text{qq} \rightarrow \text{number} : |F(\text{qq})|)\}$,
abs_ax2 $\{x \leftarrow F(i+n),\ y \leftarrow \sigma(i,n,F)\}$,
nat_invariant {nat_var $\leftarrow n$}

sm_step_proof: **Prove** sigma_mult_step **from**
  $\sigma(i,n,F)$ $\{n \leftarrow n+1,\ F \leftarrow (\lambda\,\text{qq} \rightarrow \text{number} : x * F(\text{qq}))\}$,
  $\sigma(i,n,F)$ $\{n \leftarrow n+1\}$,
  nat_invariant {nat_var $\leftarrow n$}

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

mod_sigma_mult_proof: **Prove** mod_sigma_mult **from**
  sigma_mult,
  extensionality
    $\{F \leftarrow (\lambda\,\text{qq} \rightarrow \text{number} : x * F(\text{qq})),$
     $G \leftarrow (\lambda\,\text{qq} \rightarrow \text{number} : F(\text{qq}) * x)\}$

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

ss_basis_proof: **Prove** sigma_sum_basis **from**
  $\sigma(i,n,F)$ $\{n \leftarrow 0,\ F \leftarrow (\lambda\,\text{qq} \rightarrow \text{number} : F(\text{qq}) + G(\text{qq}))\}$,
  $\sigma(i,n,F)$ $\{n \leftarrow 0,\ F \leftarrow G\}$,
  $\sigma(i,n,F)$ $\{n \leftarrow 0\}$

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

ss_step_proof: **Prove** sigma_sum_step **from**
  $\sigma(i,n,F)$ $\{n \leftarrow n+1,\ F \leftarrow (\lambda\,\text{qq} \rightarrow \text{number} : F(\text{qq}) + G(\text{qq}))\}$,
  $\sigma(i,n,F)$ $\{n \leftarrow n+1,\ F \leftarrow G\}$,
  $\sigma(i,n,F)$ $\{n \leftarrow n+1\}$,
  nat_invariant {nat_var $\leftarrow n$}

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

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

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

s1b_proof: **Prove** sigma1_basis **from**
  $\sigma(i,n,F)$ $\{n \leftarrow 0\}$,
  $\sigma(i,n,F)$ $\{i \leftarrow i+1,\ n \leftarrow 0\}$,
  $\sigma(i,n,F)$ $\{n \leftarrow 1\}$

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

s1s_proof: **Prove** sigma1_step **from**
  $\sigma(i,n,F)$ $\{i \leftarrow i+1,\ n \leftarrow n+1\}$,
  $\sigma(i,n,F)$ $\{n \leftarrow n+2\}$,
  nat_invariant {nat_var $\leftarrow n$}

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

revsigma: **Recursive** function[nat, nat, function[nat $\rightarrow$ number]
          $\rightarrow$ number] =
  $(\lambda\, i,n,F :$
    if $n = 0$ **then** 0 **else** $F(i) + \text{revsigma}(i+1,\text{pred}(n),F)$ **end if**)
  by middle

sigmaprops: **Module**

Using absolutes, natprops, functionprops[nat, number], natinduction

Exporting $\sigma(i, n, F)$, middle

Theory

i, i_1, i_2, j, k, l: **Var** nat

F, G: **Var** function[nat → number]

n, m, mm, nn, qq: **Var** nat

x, y: **Var** number

middle: function[nat, nat, function[nat → number] → nat] ==
($\lambda\, i, n, F : n$)

$\sigma(i, n, F)$:
Recursive function[nat, nat, function[nat → number] → number] =
($\lambda\, i, n, F$:
if $n = 0$ **then** 0 **else** $\sigma(i, \mathrm{pred}(n), F) + F(i + \mathrm{pred}(n))$ **end if**)
by middle

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

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

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

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

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

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

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

Proof

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

sc_basis_proof: **Prove** sigma_const_basis **from**
$\sigma(i, n, F)$ $\{n \leftarrow 0,\ F \leftarrow (\lambda\, \mathrm{qq} \to \mathrm{number} : x@c)\}$

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

sc_step_proof: **Prove** sigma_const_step **from**
$\sigma(i, n, F)$ $\{n \leftarrow n + 1,\ F \leftarrow (\lambda\, \mathrm{qq} \to \mathrm{number} : x@c)\}$,
nat_invariant {nat_var ← n}

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

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

sm_basis_proof: **Prove** sigma_mult_basis **from**
$\sigma(i, n, F)$ $\{n \leftarrow 0\}$,
$\sigma(i, n, F)$ $\{n \leftarrow 0,\ F \leftarrow (\lambda\, \mathrm{qq} \to \mathrm{number} : x * F(\mathrm{qq}))\}$

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

rearrange_sum_proof: **Prove** rearrange_sum **from**
rearrange_sub,
rearrange_sub $\{x \leftarrow y,\ F \leftarrow G\}$,
mean_diff
$\{F \leftarrow (\lambda\, \mathrm{pp} \rightarrow \mathrm{number} : x + \mathrm{F@c(pp)}),$
$G \leftarrow (\lambda\, \mathrm{pp} \rightarrow \mathrm{number} : y + \mathrm{G@c(pp)})\}$

End sums

sums_tcc: **Module**

Using sums

Exporting all with sums

Theory

i: **Var** naturalnumber

j: **Var** naturalnumber

F: **Var** function[naturalnumber $\rightarrow$ number]

pp: **Var** naturalnumber

k: **Var** naturalnumber

mean_TCC1: **Formula** $(i \leq j) \supset ((j + 1 - i) \neq 0)$

split_mean_TCC1: **Formula**
$(i \leq j \wedge i \leq k + 1 \wedge k \leq j) \supset ((j - i + 1) \neq 0)$

Proof

mean_TCC1_PROOF: **Prove** mean_TCC1

split_mean_TCC1_PROOF: **Prove** split_mean_TCC1

End sums_tcc

sum_bound_mod_proof: **Prove** sum_bound_mod $\{\text{pp} \leftarrow k@p2\}$ **from**
$\sum_i^j(F)$,
sigma_bound $\{n \leftarrow \text{diff}(j+1, i),\ i \leftarrow i\}$,
diff $\{n \leftarrow j+1,\ m \leftarrow i\}$,
diff $\{n \leftarrow j+1,\ m \leftarrow i+1\}$

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

sum_bound0_proof: **Prove** sum_bound0 **from**
$\sum_i^j(F)\ \{i \leftarrow j+1\}$,
diff $\{n \leftarrow j+1,\ m \leftarrow j+1\}$,
$\sigma(i, n, F)\ \{i \leftarrow j+1,\ n \leftarrow 0\}$

sum_bound_proof: **Prove** sum_bound $\{\text{pp} \leftarrow \text{pp}@p1\}$ **from**
sum_bound_mod, sum_bound0

mean_bound_proof: **Prove** mean_bound $\{\text{pp} \leftarrow \text{pp}@p1\}$ **from**
sum_bound_mod, $\bigoplus_i^j(F)$, div_mult $\{z \leftarrow \sum_i^j(F),\ y \leftarrow j-i+1\}$

mean_const_proof: **Prove** mean_const **from**
mean_lemma $\{F \leftarrow (\lambda\, \text{qq} \rightarrow \text{number} : x)\}$,
sigma_const $\{n \leftarrow \text{diff}(j+1, i),\ i \leftarrow i\}$,
diff $\{n \leftarrow j+1,\ m \leftarrow i\}$

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

sum_mult_proof: **Prove** sum_mult **from**
$\sum_i^j(F)$,
$\sum_i^j(F)\ \{F \leftarrow (\lambda\, \text{qq} \rightarrow \text{number} : F(\text{qq}) * x)\}$,
mod_sigma_mult $\{i \leftarrow i,\ n \leftarrow \text{diff}(j+1, i)\}$

mean_mult_proof: **Prove** mean_mult **from**
$\bigoplus_i^j(F)$, $\bigoplus_i^j(F)\ \{F \leftarrow (\lambda\, \text{qq} \rightarrow \text{number} : F(\text{qq}) * x)\}$, sum_mult

mean_sum_proof: **Prove** mean_sum **from**
mean_lemma $\{F \leftarrow (\lambda\, \text{qq} \rightarrow \text{number} : F(\text{qq}) + G(\text{qq}))\}$,
mean_lemma,
mean_lemma $\{F \leftarrow G\}$,
sigma_sum $\{n \leftarrow \text{diff}(j+1, i),\ i \leftarrow i\}$

mean_diff_proof: **Prove** mean_diff **from**
mean_mult $\{F \leftarrow G,\ x \leftarrow -1\}$,
mean_sum $\{G \leftarrow (\lambda\, \text{qq} \rightarrow \text{number} : G(\text{qq}) * -1)\}$

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

abs_sum_proof: **Prove** abs_sum **from**
$\sum_i^j(F)$,
$\sum_i^j(F)\ \{F \leftarrow (\lambda\, \text{qq} \rightarrow \text{number} : |F(\text{qq})|)\}$,
sigma_abs $\{n \leftarrow \text{diff}(j+1, i),\ i \leftarrow i\}$,
abs_ax0

abs_mean_proof: **Prove** abs_mean **from**
$\bigoplus_i^j(F)$,
$\bigoplus_i^j(F)\ \{F \leftarrow (\lambda\, \text{qq} \rightarrow \text{number} : |F(\text{qq})|)\}$,
abs_sum,
abs_div2 $\{x \leftarrow \sum_i^j(F),\ y \leftarrow j+1-i\}$,
div_mon2 $\{x \leftarrow |\sum_i^j(F)|,\ y \leftarrow \sum_i^j(F@p2),\ z \leftarrow j+1-i\}$,
abs_ax0

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

rearrange_sub_proof: **Prove** rearrange_sub **from**
mean_const, mean_sum $\{G \leftarrow (\lambda\, \text{qq} \rightarrow \text{number} : x)\}$

sums: **Module**

Using absolutes, natprops, sigmaprops

Exporting $\sum_i^j(F), \bigoplus_i^j(F)$

Theory

i, j, k, n, pp, qq, rr: **Var** nat

x, y, z: **Var** number

F, G: **Var** function[nat $\rightarrow$ number]

$\sum_i^j(F)$: function[nat, nat, function[nat $\rightarrow$ number] $\rightarrow$ number] =
$(\lambda\, i, j, F:$ **if** $i \leq j+1$ **then** $\sigma(i, \mathrm{diff}(j+1, i), F)$ **else** 0 **end if**)

$\bigoplus_i^j(F)$: function[nat, nat, function[nat $\rightarrow$ number] $\rightarrow$ number] =
$(\lambda\, i, j, F:$ **if** $i \leq j$ **then** $\sum_i^j(F)/(j+1-i)$ **else** 0 **end if**)

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

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

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

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

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

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

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

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

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

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

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

Proof

mean_lemma_proof: **Prove** mean_lemma **from** $\bigoplus_i^j(F)$, $\sum_i^j(F)$

split_sum_proof: **Prove** split_sum **from**
$\sum_i^j(F)$,
$\sum_i^j(F)\ \{j \leftarrow k\}$,
$\sum_i^j(F)\ \{i \leftarrow k+1\}$,
split_sigma
$\{n \leftarrow \mathrm{diff}(j+1, i)$,
$m \leftarrow \mathrm{diff}(k+1, i)$,
$i \leftarrow i\}$,
diff_diff $\{n \leftarrow j+1,\ m \leftarrow k+1\}$,
diff_plus $\{n \leftarrow k+1,\ m \leftarrow i\}$,
diff_ineq $\{n \leftarrow j+1,\ m \leftarrow k+1\}$

split_mean_proof: **Prove** split_mean **from** split_sum, $\bigoplus_i^j(F)$

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

curry: function[function[nat, nat → bool] → function[nat → bool]] =
  $(\lambda\, \text{prop2} : (\lambda\, i : (\forall\, q : \text{prop2}(q, i))))$

induction2_proof: **Prove**
  induction2 $\{i_0 \leftarrow q@p2,\ i_2 \leftarrow q@p4,\ j \leftarrow i@p1\}$ **from**
  induction $\{\text{prop} \leftarrow \text{curry}(\text{prop2})\}$,
  curry $\{i \leftarrow 0\}$,
  curry $\{i \leftarrow i@p1,\ q \leftarrow i_1\}$,
  curry $\{i \leftarrow i@p1 + 1\}$,
  curry $\{i \leftarrow n,\ q \leftarrow i_3\}$

End natinduction

natinduction_tcc: **Module**

Using natinduction

Exporting all with natinduction

Theory

m: **Var** naturalnumber

n: **Var** naturalnumber

q: **Var** naturalnumber

j: **Var** naturalnumber

i: **Var** naturalnumber

d_1: **Var** naturalnumber

ind_m_proof_TCC1: **Formula** (**if** $n \geq m$ **then** $n - m$ **else** 0 **end if** ≥ 0)

Proof

ind_m_proof_TCC1_PROOF: **Prove** ind_m_proof_TCC1

End natinduction_tcc

natinduction: **Module**

Using natprops

Theory

$q, i, i_0, i_1, i_2, i_3, j, m, n$: **Var** nat

prop, A, B: **Var** function[nat $\rightarrow$ bool]

prop2: **Var** function[nat, nat $\rightarrow$ bool]

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

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

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

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

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

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

induction2: **Lemma**
$(\forall\, i_0 : \text{prop2}(i_0, 0))$
$\wedge (\forall\, j : (\forall\, i_1 : \text{prop2}(i_1, j)) \supset (\forall\, i_2 : \text{prop2}(i_2, j+1)))$
$\supset (\forall\, i_3, n : \text{prop2}(i_3, n))$

Proof

Using noetherian

prev: function[nat, nat $\rightarrow$ bool] == $(\lambda\, m, n : m+1 = n)$

instance: **Module is** noetherian[nat, prev]

x: **Var** nat

identity: function[nat $\rightarrow$ nat] == $(\lambda\, n : n)$

discharge: **Prove** well_founded {measure $\leftarrow$ identity}

ind_proof: **Prove** induction $\{i \leftarrow \text{pred}(d_1@p1)\}$ **from**
general_induction $\{d \leftarrow n,\ d_2 \leftarrow i,\ p \leftarrow \text{prop}\}$

ind_m_proof: **Prove** induction_m $\{i \leftarrow i@p1 + m\}$ **from**
induction
{prop $\leftarrow (\lambda\, x : \text{prop@c}(x+m))$,
$n \leftarrow$ **if** $n \geq m$ **then** $n - m$ **else** 0 **end if**}

mod_m_proof: **Prove** mod_induction_m $\{i \leftarrow i@p1,\ j \leftarrow i\}$ **from**
induction_m {prop $\leftarrow (\lambda\, i \rightarrow \text{bool} : A(i) \supset B(i))$}

mod_induction_proof: **Prove** mod_induction $\{i \leftarrow i@p1,\ j \leftarrow j@p1\}$
from mod_induction_m $\{m \leftarrow 0\}$, nat_invariant {nat_var $\leftarrow n$}

induction1_proof: **Prove** induction1 $\{i \leftarrow i@p1\}$ **from**
induction_m $\{m \leftarrow 1\}$

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

functionprops: **Module** $[A, B$: **Type**$]$

Theory

F, G: **Var** function$[A \to B]$

x: **Var** A

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

End functionprops

noetherian: **Module** [dom: **Type**, $<$: function[dom, dom $\to$ bool]]

Assuming

measure: **Var** function[dom $\to$ nat]

a, b: **Var** dom

well_founded: **Formula** $(\exists\, \text{measure} : a < b \supset \text{measure}(a) < \text{measure}(b))$

Theory

p, A, B: **Var** function[dom $\to$ bool]

d, d_1, d_2: **Var** dom

general_induction: **Axiom**
$(\forall\, d_1 : (\forall\, d_2 : d_2 < d_1 \supset p(d_2)) \supset p(d_1)) \supset (\forall\, d : p(d))$

End noetherian

diff_ineq_proof: **Prove** diff_ineq **from**
diff $\{m \leftarrow i\}$, diff $\{n \leftarrow m,\ m \leftarrow i\}$

End natprops

natprops_tcc: **Module**

Using natprops

Exporting all with natprops

Theory

m: **Var** naturalnumber

n: **Var** naturalnumber

diff_TCC1: **Formula** (**if** $n \geq m$ **then** $n - m$ **else** 0 **end if** ≥ 0)

Proof

diff_TCC1_PROOF: **Prove** diff_TCC1

End natprops_tcc

absolutes_tcc: **Module**

Using absolutes

Exporting all with absolutes

Theory

b: **Var** number

y: **Var** number

abs_recip_TCC1: **Formula** $(b \neq 0) \supset (b \neq 0)$

abs_recip_TCC2: **Formula** $(b \neq 0) \supset (|b| \neq 0)$

abs_div2_TCC1: **Formula** $(y > 0) \supset (y \neq 0)$

abs_recip_proof_TCC1: **Formula** (**if** $b = 0$ **then** 1 **else** b **end if** $\neq 0$)

Proof

abs_recip_TCC1_PROOF: **Prove** abs_recip_TCC1

abs_recip_TCC2_PROOF: **Prove** abs_recip_TCC2

abs_div2_TCC1_PROOF: **Prove** abs_div2_TCC1

abs_recip_proof_TCC1_PROOF: **Prove** abs_recip_proof_TCC1

End absolutes_tcc

natprops: **Module**

Exporting diff

Theory

i, m, n: **Var** nat

diff: function[nat, nat → nat] =
  ($\lambda\, n, m \rightarrow$ nat : **if** $n \geq m$ **then** $n - m$ **else** 0 **end if**)

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

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

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

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

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

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

Proof

diff_zero_proof: **Prove** diff_zero **from** diff

pred_diff_proof: **Prove** pred_diff **from** diff, diff $\{m \leftarrow m + 1\}$

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

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

diff_plus_proof: **Prove** diff_plus **from** diff pos_abs: **Lemma** $0 \le x \supset |x| = x$

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

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

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

Proof

abs_times_proof: **Prove** abs_times **from**
mult_pos $\{x \leftarrow a,\ y \leftarrow b\}$,
$|a|$,
$|a|$ $\{a \leftarrow b\}$,
$|a|$ $\{a \leftarrow a * b\}$

abs_recip_proof: **Prove** abs_recip **from**
quotient_pos $\{z \leftarrow -b\}$,
quotient_pos $\{z \leftarrow b\}$,
$|a|$ $\{a \leftarrow 1/$ **if** $b = 0$ **then** 1 **else** b **end if**$\}$,
$|a|$ $\{a \leftarrow b\}$

abs_div_proof: **Prove** abs_div **from**
abs_times $\{b \leftarrow 1/$ **if** $b = 0$ **then** 1 **else** b **end if**$\}$, abs_recip

abs_proof0: **Prove** abs_ax0 **from** $|a|$ $\{a \leftarrow 0\}$

abs_proof1: **Prove** abs_ax1 **from** $|a|$ $\{a \leftarrow x\}$

abs_proof2: **Prove** abs_ax2 **from**
$|a|$ $\{a \leftarrow x + y\}$, $|a|$ $\{a \leftarrow x\}$, $|a|$ $\{a \leftarrow y\}$

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

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

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

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

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

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

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

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

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

abs_div2_proof: **Prove** abs_div2 **from**
abs_div $\{a \leftarrow x,\ b \leftarrow y\}$, pos_abs $\{x \leftarrow y\}$

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

rearrange1_proof: **Prove** rearrange1

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

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

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

rearrange_alt_proof: **Prove** rearrange_alt **from** rearrange $\{z \leftarrow v\}$

End absolutes

arithmetics_tcc: **Module**

Using arithmetics

Exporting all with arithmetics

Theory

x: **Var** number

y: **Var** number

z: **Var** number

half_TCC1: **Formula** $(2 \neq 0)$

quotient_pos_TCC1: **Formula** $(z > 0) \supset (z \neq 0)$

div_mult_TCC1: **Formula** $(y > 0 \wedge z < x * y) \supset (y \neq 0)$

div_mon_TCC1: **Formula** $(x < y \wedge z > 0) \supset (z \neq 0)$

div_mon2_TCC1: **Formula** $(x \leq y \wedge z > 0) \supset (z \neq 0)$

quotient_pos_proof_TCC1: **Formula** (**if** $z = 0$ **then** 1 **else** z **end if** $\neq 0$)

Proof

half_TCC1_PROOF: **Prove** half_TCC1

quotient_pos_TCC1_PROOF: **Prove** quotient_pos_TCC1

div_mult_TCC1_PROOF: **Prove** div_mult_TCC1

div_mon_TCC1_PROOF: **Prove** div_mon_TCC1

div_mon2_TCC1_PROOF: **Prove** div_mon2_TCC1

quotient_pos_proof_TCC1_PROOF: **Prove** quotient_pos_proof_TCC1

End arithmetics_tcc

absolutes: **Module**

Using arithmetics

Exporting $|a|$ **with** arithmetics

Theory

a, b, u, v, w, x, y, z: **Var** number

$|a|$: function[number → number] =
($\lambda\, a$: **if** $a < 0$ **then** $-a$ **else** a **end if**)

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

abs_recip: **Lemma** $b \neq 0 \supset |1/b| = 1/|b|$

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

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

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

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

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

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

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

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

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

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

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

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

arithmetics: **Module**

Exporting $\frac{x}{2}$

Theory

x, y, z: **Var** number

$\frac{x}{2}$: function[number $\rightarrow$ number] == $(\lambda\, x : x/2)$

mult_pos: **Axiom** $x * y > 0 \Leftrightarrow (x > 0 \wedge y > 0) \vee (x < 0 \wedge y < 0)$

quotient_pos: **Lemma** $z > 0 \supset 1/z > 0$

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

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

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

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

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

Proof

quotient_pos_proof: **Prove** quotient_pos **from**
mult_pos $\{x \leftarrow z,\ y \leftarrow 1/$ **if** $z = 0$ **then** 1 **else** z **end if**$\}$

div_mult_proof: **Prove** div_mult **from**
div_mon $\{z \leftarrow y,\ x \leftarrow z,\ y \leftarrow x * y\}$

mult_mon_proof: **Prove** mult_mon **from** mult_pos $\{x \leftarrow y - x,\ y \leftarrow z\}$

mult_pos_alt: **Lemma** $x \geq 0 \wedge y \geq 0 \supset x * y \geq 0$

mult_pos_alt_proof: **Prove** mult_pos_alt **from** mult_pos

mult_mon2_proof: **Prove** mult_mon2 **from**
mult_pos_alt $\{y \leftarrow y - x,\ x \leftarrow z\}$

div_mon_proof: **Prove** div_mon **from**
mult_mon $\{z \leftarrow 1/$ **if** $z = 0$ **then** 1 **else** z **end if**$\}$, quotient_pos

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

End arithmetics

End numeric_types_tcc

numeric_types_tcc_proofs: **Module**

Using numeric_types, numeric_types_tcc

Proof

posint_TCC1_PROOF: **Prove** posint_TCC1 $\{n \leftarrow 1\}$

posnum_TCC1_PROOF: **Prove** posnum_TCC1 $\{x \leftarrow 1\}$

fraction_TCC1_PROOF: **Prove** fraction_TCC1 $\{x \leftarrow 0\}$

End numeric_types_tcc_proofs

Appendix C

Specifications

numeric_types: **Module**

Exporting all

Theory

n: **Var** int

x: **Var** number

posint: **Type from** int **with** $(\lambda\, n : n > 0)$

posnum: **Type from** number **with** $(\lambda\, x : x > 0)$

fraction: **Type from** number **with** $(\lambda\, x : x \geq 0 \wedge x < 1)$

End numeric_types

numeric_types_tcc: **Module**

Using numeric_types

Exporting all with numeric_types

Theory

n: **Var** integer

x: **Var** number

posint_TCC1: **Formula** $(\exists\, n : n > 0)$

posnum_TCC1: **Formula** $(\exists\, x : x > 0)$

fraction_TCC1: **Formula** $(\exists\, x : x \geq 0 \wedge x < 1)$

Proof

posint_TCC1_PROOF: **Prove** posint_TCC1

posnum_TCC1_PROOF: **Prove** posnum_TCC1

fraction_TCC1_PROOF: **Prove** fraction_TCC1

```
  natinduction_tcc.ind_m_proof_TCC1_PROOF
  natprops_tcc.diff_TCC1_PROOF
  sigmaprops.alt_sb_step_proof
  sigmaprops.alt_sigma_bound_one_step_proof
  sigmaprops.bounded_proof
  sigmaprops.sa_basis_proof
  sigmaprops.sa_proof
  sigmaprops.sa_step_proof
  sigmaprops.sb_basis_proof
  sigmaprops.sb_proof
  sigmaprops.sb_step_proof
  sigmaprops.sigma_bound_proof
  sigmaprops.sigma_split_proof
  sigmaprops_tcc.revsigma_TCC1_PROOF
  sigmaprops_tcc.sigma_TCC1_PROOF
  sums.abs_mean_proof
  sums.abs_sum_proof
  sums.mean_bound_proof
  sums.sum_bound_mod_proof
  sums_tcc.mean_TCC1_PROOF
  sums_tcc.split_mean_TCC1_PROOF
  top.C6_TCC1_PROOF
  top.abs_recip_TCC2_PROOF
Total: 52
```

```
  algorithm.Theorem_2
  algorithm_tcc.C6_TCC1
  algorithm_tcc.Corr_TCC1
  arithmetics.div_mon
  arithmetics.div_mon2
  arithmetics.div_mult
  arithmetics.mult_mon
  arithmetics.quotient_pos
  arithmetics_tcc.div_mon2_TCC1
  arithmetics_tcc.div_mon_TCC1
  arithmetics_tcc.div_mult_TCC1
  arithmetics_tcc.half_TCC1
  arithmetics_tcc.quotient_pos_TCC1
  arithmetics_tcc.quotient_pos_proof_TCC1
  natinduction.induction
  natinduction.induction_m
  natinduction.mod_induction1
  natinduction.mod_induction_m
  natinduction_tcc.ind_m_proof_TCC1
  natprops_tcc.diff_TCC1
  noetherian[naturalnumber, natinduction.prev].well_founded
  sigmaprops.alt_sigma_bound_one_step
  sigmaprops.alt_sigma_bound_step
  sigmaprops.bounded_lemma
  sigmaprops.sb
  sigmaprops.sigma_abs
  sigmaprops.sigma_abs_basis
  sigmaprops.sigma_abs_step
  sigmaprops.sigma_bound
  sigmaprops.sigma_bound_basis
  sigmaprops.sigma_bound_step
  sigmaprops.sigma_split
  sigmaprops_tcc.revsigma_TCC1
  sigmaprops_tcc.sigma_TCC1
  sums.abs_mean
  sums.abs_sum
  sums.mean_bound
  sums.sum_bound_mod
  sums_tcc.mean_TCC1
  sums_tcc.split_mean_TCC1
Total: 52

The completed proofs are:
  absolutes.abs_div2_proof
  absolutes.abs_div_proof
  absolutes.abs_proof0
  absolutes.abs_proof2
  absolutes.abs_recip_proof
  absolutes.abs_times_proof
  absolutes.pos_abs_proof
  absolutes_tcc.abs_div2_TCC1_PROOF
  absolutes_tcc.abs_recip_TCC1_PROOF
  absolutes_tcc.abs_recip_proof_TCC1_PROOF
  algorithm.D2bar_prop_proof
  algorithm.Theorem_2_proof
  algorithm_tcc.Corr_TCC1_PROOF
  arithmetics.div_mon2_proof
  arithmetics.div_mon_proof
  arithmetics.div_mult_proof
  arithmetics.mult_mon_proof
  arithmetics.quotient_pos_proof
  arithmetics_tcc.div_mon2_TCC1_PROOF
  arithmetics_tcc.div_mon_TCC1_PROOF
  arithmetics_tcc.div_mult_TCC1_PROOF
  arithmetics_tcc.half_TCC1_PROOF
  arithmetics_tcc.quotient_pos_TCC1_PROOF
  arithmetics_tcc.quotient_pos_proof_TCC1_PROOF
  natinduction.discharge
  natinduction.ind_m_proof
  natinduction.ind_proof
  natinduction.mod_induction1_proof
  natinduction.mod_m_proof
```

```
  sigmaprops.sigma

Formula sigmaprops_tcc.revsigma_TCC1 is a termination TCC for
  sigmaprops.revsigma
Proof of
  sigmaprops_tcc.revsigma_TCC1
must not use
  sigmaprops.revsigma

Use of the formula
  natinduction.mod_induction1
requires the following TCCs to be proven
  natinduction_tcc.ind_m_proof_TCC1

Use of the formula
  noetherian[naturalnumber, natinduction.prev].general_induction
requires the following assumptions to be discharged
  noetherian[naturalnumber, natinduction.prev].well_founded

Use of the formula
  natprops.diff
requires the following TCCs to be proven
  natprops_tcc.diff_TCC1

Use of the formula
  arithmetics.div_mult
requires the following TCCs to be proven
  arithmetics_tcc.half_TCC1
  arithmetics_tcc.quotient_pos_TCC1
  arithmetics_tcc.div_mult_TCC1
  arithmetics_tcc.div_mon_TCC1
  arithmetics_tcc.div_mon2_TCC1
  arithmetics_tcc.quotient_pos_proof_TCC1

   ================== SUMMARY ==================

The proof chain is complete

The axioms and assumptions at the base are:
  algorithm.C0
  algorithm.C3
  arithmetics.mult_pos
  noetherian[EXPR, EXPR].general_induction
Total: 4

The definitions and type-constraints are:
  absolutes.abs
  algorithm.Corr
  algorithm.D2bar
  algorithm.Delta1
  algorithm.S2
  natprops.diff
  naturalnumbers.nat_invariant
  sigmaprops.bounded
  sigmaprops.sigma
  sums.mean
  sums.sum
Total: 11

The formulae used are:
  absolutes.abs_ax0
  absolutes.abs_ax2
  absolutes.abs_div
  absolutes.abs_div2
  absolutes.abs_recip
  absolutes.abs_times
  absolutes.pos_abs
  absolutes_tcc.abs_div2_TCC1
  absolutes_tcc.abs_recip_TCC1
  absolutes_tcc.abs_recip_TCC2
  absolutes_tcc.abs_recip_proof_TCC1
  algorithm.D2bar_prop
```

```
  sums.split_mean_proof
  sums.split_sum_proof
  sums.sum_bound0_proof
  sums.sum_bound_mod_proof
  sums.sum_bound_proof
  sums.sum_mult_proof
  sums_tcc.mean_TCC1_PROOF
  sums_tcc.split_mean_TCC1_PROOF
  time.SinR_proof
  time.T_next_proof
  time.Ti_in_S_proof
  time.inRS_proof
  time.in_S_proof
  time.posR_proof
  time.posS_proof
  top.C6_TCC1_PROOF
  top.abs_recip_TCC2_PROOF
  top.culmination_TCC1_PROOF
  top.l2_TCC1_PROOF
  top.l5_TCC1_PROOF
  top.rearrange_delta_TCC1_PROOF
  top.rearrange_delta_TCC2_PROOF
  top.step2_TCC1_PROOF
Total: 173
```

B.2 Clock Synchronization Condition S2

The terse proof chain for `Theorem_2` in the specification is given below in full. It can be seen that the proof chain is complete.

```
Terse proof chain for formula Theorem_2 in module algorithm

Use of the formula
  algorithm.Theorem_2
requires the following TCCs to be proven
  algorithm_tcc.Corr_TCC1
  algorithm_tcc.C6_TCC1

Formula algorithm_tcc.Corr_TCC1 is a termination TCC for
  algorithm.Corr
Proof of
  algorithm_tcc.Corr_TCC1
must not use
  algorithm.Corr

Use of the formula
  absolutes.abs_ax0
requires the following TCCs to be proven
  absolutes_tcc.abs_recip_TCC1
  absolutes_tcc.abs_recip_TCC2
  absolutes_tcc.abs_div2_TCC1
  absolutes_tcc.abs_recip_proof_TCC1

Use of the formula
  sums.mean_bound
requires the following TCCs to be proven
  sums_tcc.mean_TCC1
  sums_tcc.split_mean_TCC1

Use of the formula
  sigmaprops.sigma_bound
requires the following TCCs to be proven
  sigmaprops_tcc.sigma_TCC1
  sigmaprops_tcc.revsigma_TCC1

Formula sigmaprops_tcc.sigma_TCC1 is a termination TCC for
  sigmaprops.sigma
Proof of
  sigmaprops_tcc.sigma_TCC1
must not use
```

```
lemma6.sub1_proof
lemma6.sub2_proof
lemma6.sub_A_proof
main.Theorem_1_proof
main.basis_proof
main.ind_proof
natinduction.discharge
natinduction.ind_m_proof
natinduction.ind_proof
natinduction.induction2_proof
natinduction.mod_induction1_proof
natinduction.mod_m_proof
natinduction_tcc.ind_m_proof_TCC1_PROOF
natprops.diff_diff_proof
natprops.diff_ineq_proof
natprops.diff_plus_proof
natprops.pred_diff_proof
natprops_tcc.diff_TCC1_PROOF
numeric_types_tcc_proofs.fraction_TCC1_PROOF
numeric_types_tcc_proofs.posint_TCC1_PROOF
numeric_types_tcc_proofs.posnum_TCC1_PROOF
sigmaprops.alt_sb_step_proof
sigmaprops.alt_sigma_bound_one_step_proof
sigmaprops.bounded_proof
sigmaprops.mod_sigma_mult_proof
sigmaprops.s1b_proof
sigmaprops.s1s_proof
sigmaprops.sa_basis_proof
sigmaprops.sa_proof
sigmaprops.sa_step_proof
sigmaprops.sb_basis_proof
sigmaprops.sb_proof
sigmaprops.sb_step_proof
sigmaprops.sc_basis_proof
sigmaprops.sc_proof
sigmaprops.sc_step_proof
sigmaprops.sigma1_proof
sigmaprops.sigma_bound_proof
sigmaprops.sigma_rev_proof
sigmaprops.sigma_split_proof
sigmaprops.sm_basis_proof
sigmaprops.sm_proof
sigmaprops.sm_step_proof
sigmaprops.split_basis_proof
sigmaprops.split_proof
sigmaprops.split_step_proof
sigmaprops.srb_proof
sigmaprops.srp_proof
sigmaprops.ss_basis_proof
sigmaprops.ss_proof
sigmaprops.ss_step_proof
sigmaprops_tcc.revsigma_TCC1_PROOF
sigmaprops_tcc.sigma_TCC1_PROOF
summations.S2_pqr_proof
summations.bound_faulty_proof
summations.bound_nonfaulty_proof
summations.culm_proof
summations.l1_proof
summations.l2_proof
summations.l3_proof
summations.l4_proof
summations.l5_proof
sums.abs_mean_proof
sums.abs_sum_proof
sums.mean_bound_proof
sums.mean_const_proof
sums.mean_diff_proof
sums.mean_lemma_proof
sums.mean_mult_proof
sums.mean_sum_proof
sums.rearrange_sub_proof
sums.rearrange_sum_proof
```

```
absolutes.abs_proof2c
absolutes.abs_proof3
absolutes.abs_proof4
absolutes.abs_proof5
absolutes.abs_proof6
absolutes.abs_proof8
absolutes.abs_recip_proof
absolutes.abs_times_proof
absolutes.pos_abs_proof
absolutes.rearrange1_proof
absolutes.rearrange2_proof
absolutes.rearrange_alt_proof
absolutes.rearrange_proof
absolutes_tcc.abs_div2_TCC1_PROOF
absolutes_tcc.abs_recip_TCC1_PROOF
absolutes_tcc.abs_recip_proof_TCC1_PROOF
algorithm.A1_proof
algorithm.C2and3_proof
algorithm.D2bar_prop_proof
algorithm.S1C_lemma_proof
algorithm.Theorem_2_proof
algorithm.clock_proof
algorithm.npos_proof
algorithm_tcc.Corr_TCC1_PROOF
arithmetics.div_mon2_proof
arithmetics.div_mon_proof
arithmetics.div_mult_proof
arithmetics.mult_mon2_proof
arithmetics.mult_mon_proof
arithmetics.mult_pos_alt_proof
arithmetics.quotient_pos_proof
arithmetics_tcc.div_mon2_TCC1_PROOF
arithmetics_tcc.div_mon_TCC1_PROOF
arithmetics_tcc.div_mult_TCC1_PROOF
arithmetics_tcc.half_TCC1_PROOF
arithmetics_tcc.quotient_pos_TCC1_PROOF
arithmetics_tcc.quotient_pos_proof_TCC1_PROOF
clockprops.S1A_lemma_proof
clockprops.adj_pos_proof
clockprops.basis_proof
clockprops.bounds_proof
clockprops.gc_proof
clockprops.i2R_proof
clockprops.ind_proof
clockprops.lower_bound2_proof
clockprops.lower_bound_proof
clockprops.rmproof
clockprops.small_shift_proof
clockprops.upper_bound_proof
clocks.rho_pos_proof
juggle.final
juggle.mult_div_proof
juggle.step1_proof
juggle.step2_proof
juggle_tcc.mult_div_TCC1_PROOF
lemma1.lemma1_proof
lemma2.lemma2_proof
lemma2.lemma2a_proof
lemma2.lemma2b_proof
lemma2.lemma2c_proof
lemma2.lemma2d_proof
lemma3.lemma3_proof
lemma4.lemma2x_proof
lemma4.lemma4_proof
lemma4.rearrange1_proof
lemma4.rearrange2_proof
lemma4.rearrange3_proof
lemma4.sublemma1_proof
lemma5.lemma5proof
lemma5.rearrange1_proof
lemma5.rearrange2_proof
lemma6.lemma6_proof
```

```
  sigmaprops.sigma1_basis
  sigmaprops.sigma1_step
  sigmaprops.sigma_abs
  sigmaprops.sigma_abs_basis
  sigmaprops.sigma_abs_step
  sigmaprops.sigma_bound
  sigmaprops.sigma_bound_basis
  sigmaprops.sigma_bound_step
  sigmaprops.sigma_const
  sigmaprops.sigma_const_basis
  sigmaprops.sigma_const_step
  sigmaprops.sigma_mult
  sigmaprops.sigma_mult_basis
  sigmaprops.sigma_mult_step
  sigmaprops.sigma_rev
  sigmaprops.sigma_rev_basis
  sigmaprops.sigma_rev_step
  sigmaprops.sigma_split
  sigmaprops.sigma_sum
  sigmaprops.sigma_sum_basis
  sigmaprops.sigma_sum_step
  sigmaprops.split_sigma
  sigmaprops.split_sigma_basis
  sigmaprops.split_sigma_step
  sigmaprops_tcc.revsigma_TCC1
  sigmaprops_tcc.sigma_TCC1
  summations.S2_pqr
  summations.bound_faulty
  summations.bound_nonfaulty
  summations.culmination
  summations.l1
  summations.l2
  summations.l3
  summations.l4
  summations.l5
  summations_tcc.culmination_TCC1
  summations_tcc.l2_TCC1
  summations_tcc.l5_TCC1
  sums.abs_mean
  sums.abs_sum
  sums.mean_bound
  sums.mean_const
  sums.mean_diff
  sums.mean_lemma
  sums.mean_mult
  sums.mean_sum
  sums.rearrange_sub
  sums.rearrange_sum
  sums.split_mean
  sums.split_sum
  sums.sum_bound
  sums.sum_bound0
  sums.sum_bound_mod
  sums.sum_mult
  sums_tcc.mean_TCC1
  sums_tcc.split_mean_TCC1
  time.SinR
  time.T_next
  time.Ti_in_S
  time.inRS
  time.in_S_lemma
  time.posR
  time.posS
Total: 173

The completed proofs are:
  absolutes.abs_div2_proof
  absolutes.abs_div_proof
  absolutes.abs_proof0
  absolutes.abs_proof1
  absolutes.abs_proof2
  absolutes.abs_proof2b
```

```
arithmetics.mult_pos_alt
arithmetics.quotient_pos
arithmetics_tcc.div_mon2_TCC1
arithmetics_tcc.div_mon_TCC1
arithmetics_tcc.div_mult_TCC1
arithmetics_tcc.half_TCC1
arithmetics_tcc.quotient_pos_TCC1
arithmetics_tcc.quotient_pos_proof_TCC1
clockprops.S1A_lemma
clockprops.adj_always_pos
clockprops.basis
clockprops.bounds
clockprops.gc_prop
clockprops.i2R
clockprops.inductive_step
clockprops.lower_bound
clockprops.lower_bound2
clockprops.nonfx
clockprops.small_shift
clockprops.upper_bound
clocks.rho_pos
juggle.mult_div
juggle.rearrange_delta
juggle.step1
juggle.step2
juggle_tcc.mult_div_TCC1
juggle_tcc.rearrange_delta_TCC1
juggle_tcc.rearrange_delta_TCC2
juggle_tcc.step2_TCC1
lemma1.lemma1def
lemma2.lemma2a
lemma2.lemma2b
lemma2.lemma2c
lemma2.lemma2d
lemma2.lemma2def
lemma3.lemma3def
lemma4.lemma2x
lemma4.lemma4def
lemma4.rearrange1
lemma4.rearrange2
lemma4.rearrange3
lemma4.sublemma1
lemma5.lemma5def
lemma5.rearrange1
lemma5.rearrange2
lemma6.lemma6def
lemma6.sublemma1
lemma6.sublemma2
lemma6.sublemma_A
main.basis
main.ind_step
natinduction.induction
natinduction.induction2
natinduction.induction_m
natinduction.mod_induction1
natinduction.mod_induction_m
natinduction_tcc.ind_m_proof_TCC1
natprops.diff_diff
natprops.diff_ineq
natprops.diff_plus
natprops.pred_diff
natprops_tcc.diff_TCC1
noetherian[naturalnumber, natinduction.prev].well_founded
numeric_types_tcc.fraction_TCC1
numeric_types_tcc.posint_TCC1
numeric_types_tcc.posnum_TCC1
sigmaprops.alt_sigma_bound_one_step
sigmaprops.alt_sigma_bound_step
sigmaprops.bounded_lemma
sigmaprops.mod_sigma_mult
sigmaprops.sb
sigmaprops.sigma1
```

```
  arithmetics.mult_pos
  functionprops[EXPR, EXPR].extensionality
  noetherian[EXPR, EXPR].general_induction
  time.C1
Total: 13

The definitions and type-constraints are:
  absolutes.abs
  algorithm.Corr
  algorithm.D2bar
  algorithm.Delta1
  algorithm.S1
  algorithm.S1A
  algorithm.S1C
  algorithm.S2
  algorithm.nonfaulty
  algorithm.rt
  clocks.goodclock
  natinduction.curry
  natprops.diff
  naturalnumbers.nat_invariant
  numeric_types.fraction_invariant
  numeric_types.posnum_invariant
  sigmaprops.bounded
  sigmaprops.revsigma
  sigmaprops.sigma
  sums.mean
  sums.sum
  time.T_sup
  time.in_R_interval
  time.in_S_interval
Total: 24

The formulae used are:
  absolutes.abs_ax0
  absolutes.abs_ax1
  absolutes.abs_ax2
  absolutes.abs_ax2b
  absolutes.abs_ax2c
  absolutes.abs_ax3
  absolutes.abs_ax4
  absolutes.abs_ax5
  absolutes.abs_ax6
  absolutes.abs_ax8
  absolutes.abs_div
  absolutes.abs_div2
  absolutes.abs_recip
  absolutes.abs_times
  absolutes.pos_abs
  absolutes.rearrange
  absolutes.rearrange1
  absolutes.rearrange2
  absolutes.rearrange_alt
  absolutes_tcc.abs_div2_TCC1
  absolutes_tcc.abs_recip_TCC1
  absolutes_tcc.abs_recip_TCC2
  absolutes_tcc.abs_recip_proof_TCC1
  algorithm.A1
  algorithm.C2and3
  algorithm.D2bar_prop
  algorithm.S1C_lemma
  algorithm.Theorem_1
  algorithm.Theorem_2
  algorithm.clock_prop
  algorithm.npos
  algorithm_tcc.C6_TCC1
  algorithm_tcc.Corr_TCC1
  arithmetics.div_mon
  arithmetics.div_mon2
  arithmetics.div_mult
  arithmetics.mult_mon
  arithmetics.mult_mon2
```

```
  noetherian[naturalnumber, natinduction.prev].well_founded

Use of the formula
  sums.mean_bound
requires the following TCCs to be proven
  sums_tcc.mean_TCC1
  sums_tcc.split_mean_TCC1

Use of the formula
  sigmaprops.sigma_bound
requires the following TCCs to be proven
  sigmaprops_tcc.sigma_TCC1
  sigmaprops_tcc.revsigma_TCC1

Formula sigmaprops_tcc.sigma_TCC1 is a termination TCC for
  sigmaprops.sigma
Proof of
  sigmaprops_tcc.sigma_TCC1
must not use
  sigmaprops.sigma

Formula sigmaprops_tcc.revsigma_TCC1 is a termination TCC for
  sigmaprops.revsigma
Proof of
  sigmaprops_tcc.revsigma_TCC1
must not use
  sigmaprops.revsigma

Use of the formula
  natprops.diff
requires the following TCCs to be proven
  natprops_tcc.diff_TCC1

Use of the formula
  arithmetics.div_mult
requires the following TCCs to be proven
  arithmetics_tcc.half_TCC1
  arithmetics_tcc.quotient_pos_TCC1
  arithmetics_tcc.div_mult_TCC1
  arithmetics_tcc.div_mon_TCC1
  arithmetics_tcc.div_mon2_TCC1
  arithmetics_tcc.quotient_pos_proof_TCC1

Use of the formula
  summations.culmination
requires the following TCCs to be proven
  summations_tcc.culmination_TCC1
  summations_tcc.l2_TCC1
  summations_tcc.l5_TCC1

Use of the formula
  juggle.rearrange_delta
requires the following TCCs to be proven
  juggle_tcc.rearrange_delta_TCC1
  juggle_tcc.rearrange_delta_TCC2
  juggle_tcc.mult_div_TCC1
  juggle_tcc.step2_TCC1

   ================== SUMMARY ==================

The proof chain is complete

The axioms and assumptions at the base are:
  algorithm.A0
  algorithm.A2
  algorithm.A2_aux
  algorithm.C0
  algorithm.C2
  algorithm.C3
  algorithm.C4
  algorithm.C5
  algorithm.C6
```

Appendix B

Proof-Chain Analysis

This Appendix reproduces the output from the EHDM Proof-Chain Analyzer for the two Theorems proved in the specification.

B.1 Clock Synchronization Condition S1

The terse proof chain for `Theorem_1` in the specification is given below in full. It can be seen that the proof chain is complete.

```
Terse proof chain for formula Theorem_1 in module algorithm

Use of the formula
  algorithm.Theorem_1
requires the following TCCs to be proven
  algorithm_tcc.Corr_TCC1
  algorithm_tcc.C6_TCC1

Formula algorithm_tcc.Corr_TCC1 is a termination TCC for
  algorithm.Corr
Proof of
  algorithm_tcc.Corr_TCC1
must not use
  algorithm.Corr
```

```
Use of the formula
  absolutes.rearrange_alt
requires the following TCCs to be proven
  absolutes_tcc.abs_recip_TCC1
  absolutes_tcc.abs_recip_TCC2
  absolutes_tcc.abs_div2_TCC1
  absolutes_tcc.abs_recip_proof_TCC1

Use of the formula
  numeric_types.posnum_invariant
requires the following TCCs to be proven
  numeric_types_tcc.posint_TCC1
  numeric_types_tcc.posnum_TCC1
  numeric_types_tcc.fraction_TCC1

Use of the formula
  natinduction.induction
requires the following TCCs to be proven
  natinduction_tcc.ind_m_proof_TCC1

Use of the formula
  noetherian[naturalnumber, natinduction.prev].general_induction
requires the following assumptions to be discharged
```

Identifier	Declaration	Module
split_step_proof	prove	sigmaprops
split_sum	formula	sums
split_sum_proof	prove	sums
srb_proof	prove	sigmaprops
srp_proof	prove	sigmaprops
ss_basis_proof	prove	sigmaprops
ss_proof	prove	sigmaprops
ss_step_proof	prove	sigmaprops
step1	formula	juggle
step1_proof	prove	juggle
step2	formula	juggle
step2_proof	prove	juggle
step2_TCC1	formula	juggle_tcc
step2_TCC1_PROOF	prove	juggle_tcc
step2_TCC1_PROOF	prove	top
sub1_proof	prove	lemma6
sub2_proof	prove	lemma6
sub_A_proof	prove	lemma6
sublemma1	formula	lemma4
sublemma1	formula	lemma6
sublemma1_proof	prove	lemma4
sublemma2	formula	lemma6
sublemma_A	formula	lemma6
sum	defined-fn	sums
sum_bound	formula	sums
sum_bound0	formula	sums
sum_bound0_proof	prove	sums
sum_bound_mod	formula	sums
sum_bound_mod_proof	prove	sums
sum_bound_proof	prove	sums
summations	module	summations
summations_tcc	module	summations_tcc
sum_mult	formula	sums
sum_mult_proof	prove	sums
sums	module	sums

Identifier	Declaration	Module
sums_tcc	module	sums_tcc
Theorem_1	formula	algorithm
Theorem_1_proof	prove	main
Theorem_2	formula	algorithm
Theorem_2_proof	prove	algorithm
Ti_in_R	formula	time
Ti_in_S	formula	time
Ti_in_S_proof	prove	time
time	module	time
Ti_proof	prove	time
T_next	formula	time
T_next_proof	prove	time
top	module	top
T_sup	defined-fn	time
T_ZERO	const	time
upper_bound	formula	clockprops
upper_bound_proof	prove	clockprops
well_founded	formula	noetherian

Table A.2: EHDM Identifers used in the Specification

Identifier	Declaration	Module
S1A_lemma	formula	clockprops
S1A_lemma_proof	prove	clockprops
s1b_proof	prove	sigmaprops
S1C	defined-fn	algorithm
S1C_lemma	formula	algorithm
S1C_lemma_proof	prove	algorithm
s1s_proof	prove	sigmaprops
S2	defined-fn	algorithm
S2_pqr	formula	summations
S2_pqr_proof	prove	summations
sa_basis_proof	prove	sigmaprops
sa_proof	prove	sigmaprops
sa_step_proof	prove	sigmaprops
sb	formula	sigmaprops
sb_basis_proof	prove	sigmaprops
sb_proof	prove	sigmaprops
sb_step_proof	prove	sigmaprops
sc_basis_proof	prove	sigmaprops
sc_proof	prove	sigmaprops
sc_step_proof	prove	sigmaprops
second_arg	literal-fn	algorithm
Sigma	const	algorithm
sigma	recursive-fn	sigmaprops
sigma1	formula	sigmaprops
sigma1_basis	formula	sigmaprops
sigma1_proof	prove	sigmaprops
sigma1_step	formula	sigmaprops
sigma_abs	formula	sigmaprops
sigma_abs_basis	formula	sigmaprops
sigma_abs_step	formula	sigmaprops
sigma_bound	formula	sigmaprops
sigma_bound_basis	formula	sigmaprops
sigma_bound_proof	prove	sigmaprops
sigma_bound_step	formula	sigmaprops
sigma_const	formula	sigmaprops

Identifier	Declaration	Module
sigma_const_basis	formula	sigmaprops
sigma_const_step	formula	sigmaprops
sigma_mult	formula	sigmaprops
sigma_mult_basis	formula	sigmaprops
sigma_mult_step	formula	sigmaprops
sigmaprops	module	sigmaprops
sigmaprops_tcc	module	sigmaprops_tcc
sigma_rev	formula	sigmaprops
sigma_rev_basis	formula	sigmaprops
sigma_rev_proof	prove	sigmaprops
sigma_rev_step	formula	sigmaprops
sigma_split	formula	sigmaprops
sigma_split_proof	prove	sigmaprops
sigma_sum	formula	sigmaprops
sigma_sum_basis	formula	sigmaprops
sigma_sum_step	formula	sigmaprops
sigma_TCC1	formula	sigmaprops_tcc
sigma_TCC1_PROOF	prove	sigmaprops_tcc
SinR	formula	time
SinR_proof	prove	time
skew	literal-fn	algorithm
small_shift	formula	clockprops
small_shift_proof	prove	clockprops
sm_basis_proof	prove	sigmaprops
sm_proof	prove	sigmaprops
sm_step_proof	prove	sigmaprops
split_basis_proof	prove	sigmaprops
split_mean	formula	sums
split_mean_proof	prove	sums
split_mean_TCC1	formula	sums_tcc
split_mean_TCC1_PROOF	prove	sums_tcc
split_proof	prove	sigmaprops
split_sigma	formula	sigmaprops
split_sigma_basis	formula	sigmaprops
split_sigma_step	formula	sigmaprops

Identifier	Declaration	Module
numeric_types_tcc_proofs	module	numeric_types_tcc_proofs
period	type	time
pos_abs	formula	absolutes
pos_abs_proof	prove	absolutes
posclocktime	type	time
posint	subtype-with	numeric_types
posint_TCC1	formula	numeric_types_tcc
posint_TCC1_PROOF	prove	numeric_types_tcc
posint_TCC1_PROOF	prove	numeric_types_tcc_proofs
posnum	subtype-with	numeric_types
posnum_TCC1	formula	numeric_types_tcc
posnum_TCC1_PROOF	prove	numeric_types_tcc
posnum_TCC1_PROOF	prove	numeric_types_tcc_proofs
posR	formula	time
posrealtime	type	time
posR_proof	prove	time
posS	formula	time
posS_proof	prove	time
pred_diff	formula	natprops
pred_diff_proof	prove	natprops
prev	literal-fn	natinduction
proc	type	clocks
quotient_pos	formula	arithmetics
quotient_pos_proof	prove	arithmetics
quotient_pos_proof_TCC1	formula	arithmetics_tcc
quotient_pos_proof_TCC1_PROOF	prove	arithmetics_tcc
quotient_pos_TCC1	formula	arithmetics_tcc
quotient_pos_TCC1_PROOF	prove	arithmetics_tcc
R	const	time
realtime	type	time
rearrange	formula	absolutes
rearrange1	formula	absolutes
rearrange1	formula	lemma4
rearrange1	formula	lemma5
rearrange1_proof	prove	absolutes

Identifier	Declaration	Module
rearrange1_proof	prove	lemma4
rearrange1_proof	prove	lemma5
rearrange2	formula	absolutes
rearrange2	formula	lemma4
rearrange2	formula	lemma5
rearrange2_proof	prove	absolutes
rearrange2_proof	prove	lemma4
rearrange2_proof	prove	lemma5
rearrange3	formula	lemma4
rearrange3_proof	prove	lemma4
rearrange_alt	formula	absolutes
rearrange_alt_proof	prove	absolutes
rearrange_delta	formula	juggle
rearrange_delta_TCC1	formula	juggle_tcc
rearrange_delta_TCC1_PROOF	prove	juggle_tcc
rearrange_delta_TCC1_PROOF	prove	top
rearrange_delta_TCC2	formula	juggle_tcc
rearrange_delta_TCC2_PROOF	prove	juggle_tcc
rearrange_delta_TCC2_PROOF	prove	top
rearrange_proof	prove	absolutes
rearrange_sub	formula	sums
rearrange_sub_proof	prove	sums
rearrange_sum	formula	sums
rearrange_sum_proof	prove	sums
revsigma	recursive-fn	sigmaprops
revsigma_TCC1	formula	sigmaprops_tcc
revsigma_TCC1_PROOF	prove	sigmaprops_tcc
rho	const	clocks
rho_pos	formula	clocks
rho_small	formula	clocks
rmproof	prove	clockprops
rt	defined-fn	algorithm
S	const	time
S1	defined-fn	algorithm
S1A	defined-fn	algorithm

Identifier	Declaration	Module
lemma2_proof	prove	lemma2
lemma2x	formula	lemma4
lemma2x_proof	prove	lemma4
lemma3	module	lemma3
lemma3def	formula	lemma3
lemma3_proof	prove	lemma3
lemma4	module	lemma4
lemma4def	formula	lemma4
lemma4_proof	prove	lemma4
lemma5	module	lemma5
lemma5def	formula	lemma5
lemma5proof	prove	lemma5
lemma6	module	lemma6
lemma6def	formula	lemma6
lemma6_proof	prove	lemma6
lower_bound	formula	clockprops
lower_bound2	formula	clockprops
lower_bound2_proof	prove	clockprops
lower_bound_proof	prove	clockprops
m	const	algorithm
main	module	main
mean	defined-fn	sums
mean_bound	formula	sums
mean_bound_proof	prove	sums
mean_const	formula	sums
mean_const_proof	prove	sums
mean_diff	formula	sums
mean_diff_proof	prove	sums
mean_lemma	formula	sums
mean_lemma_proof	prove	sums
mean_mult	formula	sums
mean_mult_proof	prove	sums
mean_sum	formula	sums
mean_sum_proof	prove	sums
mean_TCC1	formula	sums_tcc

Identifier	Declaration	Module
mean_TCC1_PROOF	prove	sums_tcc
middle	literal-fn	sigmaprops
mod_induction	formula	natinduction
mod_induction1	formula	natinduction
mod_induction1_proof	prove	natinduction
mod_induction_m	formula	natinduction
mod_induction_proof	prove	natinduction
mod_m_proof	prove	natinduction
mod_sigma_mult	formula	sigmaprops
mod_sigma_mult_proof	prove	sigmaprops
monoproof	prove	clocks
monotonicity	formula	clocks
mult_div	formula	juggle
mult_div_proof	prove	juggle
mult_div_TCC1	formula	juggle_tcc
mult_div_TCC1_PROOF	prove	juggle_tcc
mult_mon	formula	arithmetics
mult_mon2	formula	arithmetics
mult_mon2_proof	prove	arithmetics
mult_mon_proof	prove	arithmetics
mult_pos	axiom	arithmetics
mult_pos_alt	formula	arithmetics
mult_pos_alt_proof	prove	arithmetics
n	const	algorithm
natinduction	module	natinduction
natinduction_tcc	module	natinduction_tcc
natprops	module	natprops
natprops_tcc	module	natprops_tcc
noetherian	module	noetherian
nonfaulty	defined-fn	algorithm
nonfx	formula	clockprops
npos	formula	algorithm
npos_proof	prove	algorithm
numeric_types	module	numeric_types
numeric_types_tcc	module	numeric_types_tcc

Identifier	Declaration	Module
extensionality	axiom	functionprops
final	prove	juggle
fraction	subtype-with	numeric_types
fraction_TCC1	formula	numeric_types_tcc
fraction_TCC1_PROOF	prove	numeric_types_tcc
fraction_TCC1_PROOF	prove	numeric_types_tcc_proofs
functionprops	module	functionprops
gc_proof	prove	clockprops
gc_prop	formula	clockprops
general_induction	axiom	noetherian
goodclock	defined-fn	clocks
half	literal-fn	arithmetics
half_TCC1	formula	arithmetics_tcc
half_TCC1_PROOF	prove	arithmetics_tcc
i2R	formula	clockprops
i2R_proof	prove	clockprops
identity	literal-fn	natinduction
ind_m_proof	prove	natinduction
ind_m_proof_TCC1	formula	natinduction_tcc
ind_m_proof_TCC1_PROOF	prove	natinduction_tcc
ind_proof	prove	clockprops
ind_proof	prove	main
ind_proof	prove	natinduction
ind_step	formula	main
induction	formula	natinduction
induction1	formula	natinduction
induction1_proof	prove	natinduction
induction2	formula	natinduction
induction2_proof	prove	natinduction
induction_m	formula	natinduction
inductive_step	formula	clockprops
initial_Corr	function	algorithm
in_R_interval	defined-fn	time
inRS	formula	time
inRS_proof	prove	time

Identifier	Declaration	Module
in_S_interval	defined-fn	time
in_S_lemma	formula	time
in_S_proof	prove	time
instance	module	natinduction
juggle	module	juggle
juggle_tcc	module	juggle_tcc
l1	formula	summations
l1_proof	prove	summations
l2	formula	summations
l2_proof	prove	summations
l2_TCC1	formula	summations_tcc
l2_TCC1_PROOF	prove	summations_tcc
l2_TCC1_PROOF	prove	top
l3	formula	summations
l3_proof	prove	summations
l4	formula	summations
l4_proof	prove	summations
l5	formula	summations
l5_proof	prove	summations
l5_TCC1	formula	summations_tcc
l5_TCC1_PROOF	prove	summations_tcc
l5_TCC1_PROOF	prove	top
lemma1	module	lemma1
lemma1def	formula	lemma1
lemma1_proof	prove	lemma1
lemma2	module	lemma2
lemma2a	formula	lemma2
lemma2a_proof	prove	lemma2
lemma2b	formula	lemma2
lemma2b_proof	prove	lemma2
lemma2c	formula	lemma2
lemma2c_proof	prove	lemma2
lemma2d	formula	lemma2
lemma2def	formula	lemma2
lemma2d_proof	prove	lemma2

Identifier	Declaration	Module
bound_faulty	formula	summations
bound_faulty_proof	prove	summations
bound_nonfaulty	formula	summations
bound_nonfaulty_proof	prove	summations
bounds	formula	clockprops
bounds_proof	prove	clockprops
C0	axiom	algorithm
C1	axiom	time
C2	axiom	algorithm
C2and3	formula	algorithm
C2and3_proof	prove	algorithm
C3	axiom	algorithm
C4	axiom	algorithm
C5	axiom	algorithm
C6	axiom	algorithm
C6_TCC1	formula	algorithm_tcc
C6_TCC1_PROOF	prove	algorithm_tcc
C6_TCC1_PROOF	prove	top
clock	function	clocks
clock_proof	prove	algorithm
clock_prop	formula	algorithm
clockprops	module	clockprops
clocks	module	clocks
clocktime	type	time
Corr	recursive-fn	algorithm
Corr_TCC1	formula	algorithm_tcc
Corr_TCC1_PROOF	prove	algorithm_tcc
culmination	formula	summations
culmination_TCC1	formula	summations_tcc
culmination_TCC1_PROOF	prove	summations_tcc
culmination_TCC1_PROOF	prove	top
culm_proof	prove	summations
curry	defined-fn	natinduction
D2bar	defined-fn	algorithm
D2bar_prop	formula	algorithm

Identifier	Declaration	Module
D2bar_prop_proof	prove	algorithm
Delta	const	algorithm
delta	const	algorithm
delta0	const	algorithm
Delta1	defined-fn	algorithm
Delta2	function	algorithm
diff	defined-fn	natprops
diff1	formula	natprops
diff1_proof	prove	natprops
diff_diff	formula	natprops
diff_diff_proof	prove	natprops
diff_ineq	formula	natprops
diff_ineq_proof	prove	natprops
diff_plus	formula	natprops
diff_plus_proof	prove	natprops
diff_TCC1	formula	natprops_tcc
diff_TCC1_PROOF	prove	natprops_tcc
diff_zero	formula	natprops
diff_zero_proof	prove	natprops
diminish	formula	clocks
diminish_proof	prove	clocks
discharge	prove	natinduction
div_mon	formula	arithmetics
div_mon2	formula	arithmetics
div_mon2_proof	prove	arithmetics
div_mon2_TCC1	formula	arithmetics_tcc
div_mon2_TCC1_PROOF	prove	arithmetics_tcc
div_mon_proof	prove	arithmetics
div_mon_TCC1	formula	arithmetics_tcc
div_mon_TCC1_PROOF	prove	arithmetics_tcc
div_mult	formula	arithmetics
div_mult_proof	prove	arithmetics
div_mult_TCC1	formula	arithmetics_tcc
div_mult_TCC1_PROOF	prove	arithmetics_tcc
eps	const	algorithm

Identifier	Declaration	Module
A0	axiom	algorithm
A1	formula	algorithm
A1_proof	prove	algorithm
A2	axiom	algorithm
A2_aux	axiom	algorithm
abs	defined-fn	absolutes
abs_ax0	formula	absolutes
abs_ax1	formula	absolutes
abs_ax2	formula	absolutes
abs_ax2b	formula	absolutes
abs_ax2c	formula	absolutes
abs_ax3	formula	absolutes
abs_ax4	formula	absolutes
abs_ax5	formula	absolutes
abs_ax6	formula	absolutes
abs_ax7	formula	absolutes
abs_ax8	formula	absolutes
abs_div	formula	absolutes
abs_div2	formula	absolutes
abs_div2_proof	prove	absolutes
abs_div2_TCC1	formula	absolutes_tcc
abs_div2_TCC1_PROOF	prove	absolutes_tcc
abs_div_proof	prove	absolutes
abs_mean	formula	sums
abs_mean_proof	prove	sums
absolutes	module	absolutes
absolutes_tcc	module	absolutes_tcc
abs_proof0	prove	absolutes
abs_proof1	prove	absolutes
abs_proof2	prove	absolutes
abs_proof2b	prove	absolutes
abs_proof2c	prove	absolutes
abs_proof3	prove	absolutes
abs_proof4	prove	absolutes

Identifier	Declaration	Module
abs_proof5	prove	absolutes
abs_proof6	prove	absolutes
abs_proof7	prove	absolutes
abs_proof8	prove	absolutes
abs_recip	formula	absolutes
abs_recip_proof	prove	absolutes
abs_recip_proof_TCC1	formula	absolutes_tcc
abs_recip_proof_TCC1_PROOF	prove	absolutes_tcc
abs_recip_TCC1	formula	absolutes_tcc
abs_recip_TCC1_PROOF	prove	absolutes_tcc
abs_recip_TCC2	formula	absolutes_tcc
abs_recip_TCC2_PROOF	prove	absolutes_tcc
abs_recip_TCC2_PROOF	prove	top
abs_sum	formula	sums
abs_sum_proof	prove	sums
abs_times	formula	absolutes
abs_times_proof	prove	absolutes
adj_always_pos	formula	clockprops
adj_pos_proof	prove	clockprops
adjusted	literal-fn	algorithm
algorithm	module	algorithm
algorithm_tcc	module	algorithm_tcc
alt_sb_step_proof	prove	sigmaprops
alt_sigma_bound_one_step	formula	sigmaprops
alt_sigma_bound_one_step_proof	prove	sigmaprops
alt_sigma_bound_step	formula	sigmaprops
arithmetics	module	arithmetics
arithmetics_tcc	module	arithmetics_tcc
basis	formula	clockprops
basis	formula	main
basis_proof	prove	clockprops
basis_proof	prove	main
bounded	defined-fn	sigmaprops
bounded_lemma	formula	sigmaprops
bounded_proof	prove	sigmaprops

Appendix A

Cross-Reference Listing

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

Identifier	Translation
`abs`	$\|a\|$
`adjusted`	$A_p^{(i)}(T)$
`clock`	$c_{\star 1}(\star 2)$
`Corr`	$C_p^{(i)}$
`D2bar`	$\bar{\Delta}_{r,p}^{(i)}$
`delta0`	δ_0
`Delta1`	$\Delta_p^{(i)}$
`Delta2`	$\Delta_{\star 1,\star 2}^{(\star 3)}$
`eps`	ϵ
`half`	$\frac{x}{2}$
`initial_Corr`	$C_{\star 1}^{0}$
`in_R_interval`	$T \in R^{(i)}$
`in_S_interval`	$T \in S^{(i)}$
`mean`	$\bigoplus_i^j(F)$
`rt`	$c_p^{(i)}(T)$
`sigma`	$\sigma(i, n, F)$
`sum`	$\sum_i^j(F)$
`TN`	T_N
`T_sup`	$T^{(i)}$
`T_ZERO`	T^0

Table A.1: Translations for Identifiers Used in the Specification

[24] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[25] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, August 1987.

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

[27] Natarajan Shankar. Mechanical verification of a schematic Byzantine fault-tolerant clock synchronization algorithm. Technical Report SRI-CSL-91-4, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also NASA Contractor Report 4386.

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

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

[30] William D. Young. Verifying the Interactive Convergence clock-synchronization algorithm using the Boyer-Moore prover. Internal Note 199, Computational Logic Incorporated, Austin, TX, January 1991.

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

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

[13] Matt Kaufmann. DEFN-SK: An extension of the Boyer-Moore theorem prover to handle first-order quantifiers. Technical Report 43, Computational Logic Incorporated, Austin, TX, 1989.

[14] Herman Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, C-36(8):933–940, August 1987.

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

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

[17] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley Publishing Company, Reading, MA, 1985.

[18] Leslie Lamport. Synchronizing time servers. Technical Report 18, DEC Systems Research Center, Palo Alto, CA, June 1987.

[19] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming*, volume 1: Deductive Reasoning. Addison-Wesley, 1985.

[20] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming*, volume 2: Deductive Systems. Addison-Wesley, 1990.

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

[22] Parameswaran Ramanathan, Kang G. Shin, and Ricky W. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer*, 23(10):33–42, October 1990.

[23] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. Technical Report SRI-CSL-91-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also NASA Contractor Report 4384.

Bibliography

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

[2] Jon Barwise. Mathematical proofs of computer system correctness. *Notices of the AMS*, 36:844–851, September 1989.

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

[4] R.S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[5] Ricky W. Butler. A survey of provably correct fault-tolerant clock synchronization techniques. NASA Technical Memorandum 100553, NASA Langley Research Center, February 1988.

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

[7] EHDM *Specification and Verification System Version 4.1—User's Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, November 1988. See [8] for the updates to Version 5.2.

[8] EHDM *Specification and Verification System Version 5.2—Supplement to User's and Language Manuals.* Computer Science Laboratory, SRI International, Menlo Park, CA, March 1991. Current version number is 5.2.0.

[9] Flaviu Cristian. Probabilistic clock synchronization. Technical Report RJ 6432, IBM Almaden Research Center, San Jose, CA, September 1988.

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

A further challenge is to formalize and verify an *implementation* of the Interactive Convergence Clock Synchronization Algorithm—so far, we have simply verified properties of the algorithm itself. Our current work is addressing these challenges; we expect to report our results towards the end of 1991.

for doing those proofs. When reflected back into a traditional mathematical presentation (given in Chapter 2), we consider the result to be an analysis that is not only more precise, but simpler and easier to follow than the original.

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

Our formal verification does not usurp the social process in which De Millo, Lipton and Perlis place their faith, but should serve to shift its focus from details to fundamentals—i.e., to scrutiny of the definitions employed. We note that the social process apparently failed to discover the flaws that we have noted in the main theorem concerning the Interactive Convergence Clock Synchronization Algorithm, and in four of its five lemmas. This is not surprising: the standards of rigor and formality in what Barwise [2] calls the "proof sketches" of the normal "mathematical demonstration" are simply inadequate to the intricacy and detail required for the analysis of many algorithms and programs. Mechanically checked verification provides valuable supplementary scrutiny and evidence in these cases.

Mechanical verification does not, however, provide automatic scrutiny of the adequacy of the definitions and axioms employed, nor of the utility of the theorems proved: one can prove true, but useless theorems—useless either because they do not address the issues of real concern, or because they are predicated on unsatisfiable assumptions. The only way to resolve these concerns is through human scrutiny—i.e., the social process. Mechanical verification can provide evidence for consideration by the social process by supporting testing and experimentation with definitions and axioms. For example, models can be exhibited to show that axiomatizations are satisfiable, and test cases can be examined to ensure that definitions are "reasonable." We are considering ways in which a formal verification system such as EHDM can provide improved support for these activities.

In addition to these logical properties of our definitions, axioms, and theorems, we also need to consider their realism—i.e., the extent to which they adequately model real-world behavior. The aspect of the representation of the clock synchronization problem that causes us most concern is the basic definition of a clock. Real clocks increment in discrete "ticks" whose magnitude may be quite large compared with some of the other parameters in the system. Using the rationals as the interpretation of clock time is therefore unrealistic, as is the requirement that a good clock should be a strict monotonic function. Schneider [25] presents a formulation of clock synchronization which treats these aspects more realistically. Our colleague Shankar has conducted a formal specification and verification of Schneider's formulation [27].

This is parody. In a paper published several years earlier [29], von Henke and Luckham indicated the true nature of the scenario envisioned by the proponents of verification when they wrote:

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

The thrust of De Millo, Lipton and Perlis' argument is that formal verification moves responsibility away from the "social process" that involves human scrutiny, towards a mechanical process with little human participation. In reality, a verification system assists the human user to develop a convincing argument for the correctness of his program by acting as an implacable skeptic which demands that all assumptions be stated and all claims justified. The requirement to explicate and formalize what would otherwise be unexamined assumptions is especially valuable. Shankar [26], for example, observes:

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

This perspective on mechanical theorem proving is very similar to that developed by Lakatos [15] for the role of proof (not just mechanical theorem proving) in mathematics. Crudely, this view is that successful completion is among the least interesting and useful outcomes of a proof attempt; the real benefit comes from failed proof attempts, since these challenge us to revise our hypotheses, sharpen our statements, and achieve a deeper understanding of our problem.

Our own experience with the verification of the Interactive Convergence Clock Synchronization Algorithm supports this view. Most of our time was spent in trying to prove theorems and lemmas that turned out to be false, in coming to understand why they were false, and in revising their statements, or those of supporting lemmas and assumptions. The difficulties we encountered were consequences of genuine technical flaws in the previously published analysis of the Algorithm [16], and we consider the main benefit of this exercise to be the identification and correction of those flaws. The corrections led us to eliminate the use of approximations, thereby allowing precise statements of the constraints on the values of the parameters to the Algorithm, and led us to modify one of the assumptions (A2) underlying the Algorithm, thereby changing its external specification slightly. Our corrections to the statements and proofs of some of the lemmas led us to a more uniform method cant challenges, but they are at least more sharply posed than the problems with which we began.

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

The opening paragraphs of the introductory document to EHDM [24] make our own attitude clear:

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

The opponents to formal verification [10, 12] ignore caveats such as those expressed above (which are similar to those expressed by all serious proponents of formal verification) and perform a straw man attack in which verification is set up as an unequivocal demonstration of correctness, and in which intelligent human participation is minimized in favor of an omniscient mechanical verifier. For example, De Millo, Lipton and Perlis [10] claim that:

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

Chapter 5

Conclusions

> *"The virtue of a logical proof is not that it compels belief but that it suggests doubts."* [15, page 48]

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

In practice, all three of these limitations on "correctness" pose significant challenges. The behavior of the actual system will depend on physical processes that may not admit completely accurate descriptions, or that may be subject to random effects, while the properties required of the system may not be fully understood, let alone fully recorded in its specification. And demonstration of consistency between the two descriptions of the system will be subject to the errors attendant upon any human enterprise. *Formal* specification and verification attempts to control and delimit some of the difficulties associated with verification; the use of formal *specifications* can at least provide precise and unambiguous descriptions of the intended behavior of the system—the questions remain whether these descriptions correctly capture what is really required, or what the behavior of the system really is, but at least the doubt about what the descriptions themselves *mean* is removed. Formal *verification* attempts to put the demonstration of consistency between two system descriptions onto a more reliable basis by making it a mathematical—indeed, calculational—activity that can be checked by a mechanical theorem prover. Of course, the validity of this approach depends on the extent to which the semantics of the specification language are correctly implemented by its support environment, and on the correctness of the mechanical theorem prover. These represent signifidevelopment of verification environments. Consequently, we invite the developers and users of other verification systems to repeat the experiment described here. We suggest that the Interactive Convergence Clock Synchronization Algorithm is a paradigmatic example of a problem where formal verification can show its value and a verification system can demonstrate its capabilities: it is a "real" rather than an 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.

Since this "challenge" appeared in the first edition of this report, Bill Young of Computational Logic Inc. has repeated our verification using the Boyer-Moore theorem prover [30]. Young used extensions [13] to the standard Boyer-Moore prover [3, 4] that permitted him to use quantifiers and to introduce uninterpreted functions whose properties are stated axiomatically. He inhibited much of the automatic rewriting and associated proof-search normally performed by the Boyer-Moore prover and followed our verification pretty much lemma for lemma. Young drew our attention to one of the deficiencies in our original specification of a good clock, and found a better way to organize the main induction. A collaborative paper comparing the two verifications, and the systems in which they were undertaken, is planned for the future.

transliterating from the notation of Lamport and Melliar-Smith into the specification language of EHDM was straightforward. Automation of the reverse translation (by the LaTeX-printer) enabled us to do most of our work and thinking using compact and familiar notation and thereby contributed greatly to our productivity. The higher-order capabilities of EHDM allowed us to define the summation and averaging operators very straightforwardly and also enabled us to tailor induction schemes appropriately.

The arithmetic decision procedures of EHDM were of immense value in the formal verification. We doubt that verification environments lacking such decision procedures could accomplish the work described here without unreasonable effort. We found the basic theorem-proving paradigm of EHDM straightforward and adequate for its purpose (though others, especially novices, might not agree). The correspondence between the information in an EHDM "prove" declaration and that required for a journal-level proof description is quite close. Naturally, increased automation of details (for example, use of term rewriting to mechanize equational theories, and automatic discovery of substitution instances) would be welcome,[15] but we did not find theorem proving to be a bottleneck. (Discovering the correct theorems to prove was the bottleneck.)

The module structure supported by the EHDM specification language and its support environment simplified the task of managing and comprehending a formal development that eventually became quite large, and enabled us to keep track of undischarged proof obligations. The latter service was particularly valuable, due to the way in which our formal specification and verification were developed. Our approach was very much top-down: we introduced lemmas whenever it was convenient to do so, and worried about proving them later. We may have carried this approach a little too far in the early stages (i.e., we did not examine the content of our lemmas with sufficient care), but we did not know at that period whether our attempt to mechanically verify the algorithm would be successful[16] and we were anxious to explore the more obviously difficult parts first.

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

We found that EHDM served us reasonably well; we do not know whether other specification and verification environments would have fared as well or better. Understanding the practical benefits and limitations of different approaches to formal specification and mechanical theorem proving is necessary for sensible further

[15] And will be available in EHDM Version 6.

[16] The algorithm (or rather an implementation of it) had been asserted to be "probably beyond the ability of any current mechanical verifier" [21, page 9].

> Convergence Clock Synchronization Algorithm, and believe that they provide a simple and near-minimal foundation on which to construct the definition and analysis of this algorithm. Similarly, we believe that the 3 supporting axioms correspond to conventional interpretations of the concepts concerned.
>
> It is generally possible to establish the soundness of axiomatizations in EHDM by exhibiting a model.[13] We have not done so for this specification because, as noted, the current implementation of EHDM has some deficiencies when mapping modules that contain both interpreted and uninterpreted functions. We intend to demonstrate the soundness of the 10 axioms used in the specification of the algorithm when EHDM Version 6 becomes available.

It is difficult to answer the question whether the flaws we found in the published analysis of the Interactive Convergence Clock Synchronization Algorithm could have been discovered more easily by other methods. Once the flaws are known, they are easy to describe and their presence in the published proof is almost painfully obvious. Nonetheless, as far as we know, these flaws were not discovered previously. The reputation of the journal in which the paper was published, and of its authors, may have caused some to assume that the proof "must be right" without further scrutiny, and may have stilled any doubts in the minds of those who examined the proof in sufficient detail to become concerned by some of its details. Some who scrutinized the proof with great care decided that it would be easier to develop their own analysis than to persuade themselves of the veracity of the original.[14]

The root difficulty, we believe, lies in the fact that the proof in [16], though neither mathematically deep nor intrinsically interesting, is astonishingly intricate in its details. The analysis of many algorithms, computer programs, and similar artifacts shares this characteristic—and renders the standard "mathematical demonstration" (which forms the basis for the consensus model of classical mathematics) unreliable in these contexts.

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

We found EHDM very satisfactory from this perspective: because EHDM uses a standard logic (predicate calculus) with all the usual quantifiers and connectives,

[13] Actually, a theory interpretation and the soundness is relative to that of the second theory. See [24, Chapter 3].

[14] Fred Schneider has told us that this was one of the motivations behind [25].

The verification described in the first edition of this report depended on 47 axioms. Of these, 29 (6 in module `time`, 6 in `clocks` and 17 in `algorithm`) defined the concepts, constraints, and algorithm of direct interest. The other 18 introduced supporting concepts (e.g., summation) or properties of arithmetic beyond those built into the system (i.e., some of the properties of division and multiplication).

The current verification exploits improved capabilities for conservative extension (i.e., constant and subtype definitions) introduced in EHDM Version 5 and reduces the number of axioms required to 13. Of these, 7 are the constraints `C0` to `C6` and three specify the assumptions `A0`, `A2`, and `A2_aux`. Thus, 10 of the axioms are fundamental to the specification of the problem of interest. The remaining three axioms specify Noetherian induction, function extensionality, and the condition under which the product of two numbers is strictly positive. All other concepts of the specification (including all the supporting theories concerned with summation) are introduced by means of definitions and thus cannot introduce inconsistencies.

However, as noted earlier, the previous version of the verification used a definition for the predicate *good clock* that is unsatisfiable (i.e., there can be no clocks satisfying the predicate). This is not inconsistent (an inconsistency results if we add an axiom asserting that there is a good clock), but it is almost equally damaging, since our theorems have the form "if at least a certain number of clocks are good, then the good clocks stay synchronized." Since there can be no good clocks, the antecedent is false and the theorem is true, but useless. (It has the same structure as the true theorem "if 1=2, then 2=3.") Definitions need to be scrutinized no less carefully than axioms in order to ensure that they capture the intent of the specification. A formal verification system can support this by allowing experimentation and "reasonableness tests." For example, it is reasonable to expect that a defined predicate is not independent of its arguments. The flaw in our previous definition of a *good clock* would have become readily apparent if we had attempted to prove the test case that a perfect clock is a good clock (i.e., that the definition of a good clock is satisfiable when the clock is the identity function). It is possible to perform these tests in EHDM using mapping modules (the clock function is a constant, not a variable, so the test cannot be performed within the specification; it is necessary to consider a model or interpretation). We have not done so for this specification because the current implementation of EHDM has some technical deficiencies in this regard. We are considering ways in which EHDM Version 6 can provide improved support for these activities.

We spent a great deal of effort reducing the number and simplifying the content of the 10 remaining axioms underlying our specification of the Interactive the module `juggle`, which took a man-day to create[11]), and continually revised the modules of the supporting theories in order to simplify and systematize the axiomatic basis on which the whole verification depends. This process proceeded in parallel with the preparation of this report—both activities together consumed about two man-months.

In the period following publication of the first edition of this report in February 1989, we have undertaken additional verifications and have made significant improvements to the EHDM language and system. As we gained experience and as new capabilities became available, we modified the specification and verification accordingly. The total time expended on these adjustments to the verification was probably less than a man-week.

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

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

- A substantial portion of the time devoted to the mechanical verification was expended on the supporting theories. As formal verification becomes more widely practiced, we would expect libraries of such theories to become established, so that later efforts can concentrate their efforts on the problem of real interest.[12] If we neglect the effort spent on the supporting theories, then the time required to perform the mechanical verification was of a similar order to that required to prepare an adequately detailed "journal-level" description and proof for human consumption (i.e., the first 3 Chapters of this report).

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

[11] This was using an earlier version of EHDM that had poor support for division and nonlinear multiplication. The proof of `rearrange_delta` required 13 intermediate lemmas in that version.

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

our time to the effort. One of us broke the published proof of Lamport and Melliar-Smith down into elementary steps, while the other encoded these in EHDM and persuaded the theorem prover to accept the proofs. At this point we had caught the typographical errors in Lemmas 2 and 4, and had proofs of Lemmas 1, 3, 4, and 5—but Lemma 2 was essentially taken as an axiom. Approximate equality and inequalities were used freely at this stage, although several of the formulas needed were mentally flagged as suspicious.

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

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

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

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

Finally, we began to put the whole verification together and to prepare this document. We developed the module `sigmaprops` and used it to prove the previously unproved formulas in module `sums`. We discovered several minor flaws in the statements of those formulas while performing their proofs. As we began to describe and document our specifications and proofs, we filled in missing fragments (e.g.,

numeric_types		no proofs		
numeric_types_tcc	0	successful proofs	3	failures
numeric_types_tcc_proofs	3	successful proofs	0	failures
arithmetics	7	successful proofs	0	failures
arithmetics_tcc	6	successful proofs	0	failures
absolutes	20	successful proofs	0	failures
absolutes_tcc	3	successful proofs	1	failure
natprops	6	successful proofs	0	failures
natprops_tcc	1	successful proof	0	failures
functionprops		no proofs		
noetherian		no proofs		
natinduction	8	successful proofs	0	failures
natinduction_tcc	1	successful proof	0	failures
sums	16	successful proofs	0	failures
sums_tcc	2	successful proofs	0	failures
sigmaprops	30	successful proofs	0	failures
sigmaprops_tcc	2	successful proofs	0	failures
time	8	successful proofs	0	failures
clocks	4	successful proofs	0	failures
algorithm	7	successful proofs	0	failures
algorithm_tcc	1	successful proof	1	failure
clockprops	12	successful proofs	0	failures
lemma1	1	successful proof	0	failures
lemma2	5	successful proofs	0	failures
lemma3	1	successful proof	0	failures
lemma4	6	successful proofs	0	failures
lemma5	3	successful proofs	0	failures
lemma6	4	successful proofs	0	failures
summations	9	successful proofs	0	failures
summations_tcc	0	successful proofs	3	failures
juggle	4	successful proofs	0	failures
juggle_tcc	1	successful proof	3	failures
main	3	successful proofs	0	failures
top	8	successful proofs	0	failures
Totals	182	successful proofs	11	failures

Table 4.1: Proof Summaries for EHDM Modules

This lemma is proved directly (i.e., without premises) by the EHDM ground prover since the cancellation law is built in. The lemma is not superfluous, however, since the instances of the cancellation law required in the main proof are more complex than those that can be recognized automatically. By citing appropriate instances of the lemma `mult_div`, we can guide the prover along the right path.

4.2.3.5 Main and Top

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

This module `top` simply ties all the other specification modules together and provides proofs for 8 TCCs whose automatically-generated proof declarations are inadequate.

4.3 Statistics and Observations

The specification and verification described here was performed using EHDM Version 5.2.0 running on a Sun SPARCstation 2.

The specifications described here occupy 34 modules, comprising about 1,465 (nonblank) lines of EHDM (10 of the modules are system-generated TCC modules; the 24 non-system-generated modules comprise just under 1,300 lines). There are 193 proofs in the full specification, of which 11 are unsuccessful; all of these are trivial automatically-generated proofs for TCC formulas and are supplanted by manually-constructed proofs among the successful 182. It takes about 10 minutes to process all 193 proofs (a little over 3 seconds each, on average). It is hard to obtain accurate timings for individual proofs, since garbage collection and system load introduces considerable variability—however, the worst case seems to be about 45 seconds (for `sb_proof` in `sigmaprops`.) The proofs in each module are summarized in Table 4.1 below, which reproduces part of the output from the EHDM "proveall" command.

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

Unfortunately, we did not accurately record the human effort expended on this exercise, so the following account relies on memory. Our first attempt to perform the verification occupied a week, with both of us devoting about three-quarters of in turn is used to establish `lower_bound2`. The proof of `adj_always_pos` requires an induction. The proof of `upper_bound`, on the other hand, is straightforward.

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

4.2.3.2 Lemmas 1 to 6

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

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

4.2.3.3 Summations

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

The lemma `l1` connects the main term in the conclusion of Lemma 6 with the averaging step performed by the Algorithm (in the definition of $\Delta_p^{(i)}$). Lemma `l2` splits the summation implicitly involved in `l1` into two smaller summations—one over the faulty processors and one over the nonfaulty ones. Lemma `l3` uses Lemma 5 to obtain a bound on the sum of the errors introduced by the faulty processors; a subsidiary lemma called `bound_faulty` is used in the process. Lemma `l4` uses Lemma 4 to obtain a bound on the sum of the errors introduced by the nonfaulty processors; a subsidiary lemma called `bound_nonfaulty` is used in the process. Lemma `l5` simply combines lemmas `l2, l3` and `l4`; the `culmination` lemma is proved by combining `l5` with Lemma 6.

4.2.3.4 Juggle

The module `juggle` proves the lemma `rearrange_delta`. This result is a straightforward algebraic manipulation. The proof in EHDM is divided into two simpler lemmas, called `step1` and `step2`, which are similar to those used when performing the proof by hand (first multiply by `n - m`, rearrange, divide by `n` and rearrange again). The proofs of the two intermediate lemmas cite several instances of the lemma `mult_div`:

```
mult_div: LEMMA y /= 0 IMPLIES x/y*y = x
```

The assumptions `A0`, `A1`, and `A2` are stated next (they cannot precede the specification of the clock synchronization conditions `S1` and `S2` since assumption `A2` is predicated upon `S1C` and `S2`). The specifications of these assumptions are straightforward, but note that whereas the informal statement of A1 says that if p is nonfaulty through period i, then (this implies that) p has a good clock during the corresponding interval, the formal definition uses equivalence instead of implication. This is necessary because we will later need to prove that if p is nonfaulty through period $i+1$, then it is also nonfaulty through period i.

Next, the constraints `C0` and `C2` to `C6` are specified, and several minor lemmas are stated. Finally, the theorems that assert, respectively, the two clock synchronization conditions S1 and S2 are defined. The proof of the latter is simple and is performed directly in the proof part of the module `algorithm`.

The theorem that asserts the satisfaction of the condition S1 is stated as:

```
Theorem_1: THEOREM S1(i)
```

It is often advantageous to state theorems that are to be proved by induction as predicates on the induction variable—as is done here. Doing so allows instances of the theorem to be stated as explicit hypotheses in lemmas that are themselves used to establish the inductive step. An example is in the formula `culmination` in module `summations`. In the first version of the verification, we stated the condition `S1` as a formula, rather than a predicate. The dependence of the formula `culmination` on `S1` was then implicit (`S1` was cited in the proof of `culmination`), and the EHDM Proof Chain Analyzer reported a circularity in the proof of `S1` which we had to argue away as "apparent rather than real." Use of predicates to avoid this sort of difficulty is part of the lore of mechanical theorem proving. We first heard it articulated by our colleague Shankar, but its application to this particular specification derives from Bill Young's re-verification of the Algorithm using the Boyer-Moore theorem prover [30].

4.2.3 Proof Modules

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

4.2.3.1 Clockprops

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

4.2.2.3 Algorithm

The core of the Interactive Convergence Clock Synchronization Algorithm is defined in the module `algorithm`. We introduce `m` and `n` as constants of type `proc`, and assert that `m < n` (axiom `C0`). The constants `eps` (ϵ), `delta0` (δ_0), and `delta` (δ) are introduced as constants of type `posrealtime`, while `sigma` (Σ) and `Delta` (Δ) are introduced as constants of type `posclocktime`.

Next, the functions `Delta2` ($\Delta^{(\star 3)}_{\star 1, \star 2}$), `D2bar` ($\bar{\Delta}^{(i)}_{r\,p}$), and `Delta1` ($\Delta^{(i)}_p$), are introduced along with the correction, adjusted-value, and logical clock functions: `Corr` ($C^{(i)}_p$), `adjusted` ($A^{(i)}_p(T)$), and `rt` ($c^{(i)}_p(T)$), respectively. These are the definitions at the heart of the Interactive Convergence Clock Synchronization Algorithm. The correction function is defined recursively. In the original version of the verification, the bottom case of the recursion (i.e., the initial correction $C^{(0)}_p$) was defined to be zero, which is how it is defined by Lamport and Melliar-Smith. The current verification uses an uninterpreted and unconstrained function `initial_Corr` ($C^0_{\star 1}$) to supply the bottom case.

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

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

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

The consequent to S1 gives the bound on the skew between the clocks of nonfaulty processors and is stated as the predicate `S1C`:

```
S1C: function[proc, proc, period -> bool] =
  (LAMBDA p, q, i :
     (FORALL T :
        nonfaulty(p, i) AND nonfaulty(q, i) AND in_R_interval(T, i)
          IMPLIES skew(p, q, T, i) <= delta))
```

The clock synchronization condition S1 itself is then stated as follows:

```
S1: function[period -> bool] =
  (LAMBDA i: S1A(i) IMPLIES (FORALL p, q: S1C(p, q, i)))
```

which is easily provable. For more detailed explanation of recursive functions, see the EHDM tutorial [24, Chapter 6].

Seven lemmas concerning the function `sigma` are stated and proved in module `sigmaprops`. The names used are in correspondence with those used for the lemmas in `sums`: for example, `split_sigma` in `sigmaprops` corresponds to `split_sum` and `split_mean` in `sums`. The proofs in `sigmaprops` mostly use induction; the induction schemes employed are from the module `natinduction`.

Some of the proofs in `sigmaprops` use a function `revsigma` which is defined like `sigma`, but with the recursion going in the opposite direction. A lemma called `sigma_rev` establishes that these two functions are extensionally equal. A second function, called `bounded`, is also defined and used internally by `sigmaprops`.

4.2.2 Specification Modules

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

4.2.2.1 Time

The module `time` is the first one that introduces concepts directly concerned with the Interactive Convergence Clock Synchronization Algorithm. It introduces `clocktime`, `realtime`, `posclocktime`, `posrealtime`, and `period` as types, and establishes `number` (i.e., the rationals) as the interpretation of the first two, `posnum` (i.e., the strictly positive rational numbers) as the interpretation of the third and fourth, and the naturals as the interpretation of the fifth. `R, S`, and `T_ZERO` (T^0) are introduced as constants of type `posclocktime`, and `clocktime`, respectively, and then the functions `T_sup` ($T^{(i)}$), `in_R_interval` ($T \in R^{(i)}$), and `in_S_interval` ($T \in S^{(i)}$) are introduced and defined in the obvious way. The constraint C1 (`R >= 3 * S`) is also defined here, and several straightforward lemmas are stated and proved.

4.2.2.2 Clocks

The module `clocks` introduces `proc` (short for processor) as a type interpreted by the naturals, and introduces the uninterpreted function `clock` ($c_{\star 1}(\star 2)$).

Next, the drift rate `rho` (ρ) is introduced as a constant of type `fraction`, together with the predicate `goodclock`. The intended interpretation is that `goodclock(p, T0, TN)` will be true when processor `p` is a good clock in the clock time interval `[T0, TN]`.

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

from $k+1$ to j. Split_mean, the corresponding result for mean, is proved directly from split_sum.

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

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

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

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

Sigma is defined by a recursive definition. In order to ensure that recursive definitions are well-defined, EHDM requires that a *measure function* be specified for each such definition. A measure function must have the same domain as the recursive function being defined, but range type nat. EHDM then generates TCCs stating the requirement that the value of the measure function be strictly decreasing across each recursive call. For the case of sigma, we have the following specification:

```
i: VAR nat
F: VAR function[nat -> number]
n: VAR nat

middle: function[nat, nat, function[nat -> number] -> nat] ==
  (LAMBDA i, n, F : n)

sigma:recursive function[nat, nat, function[nat -> number] -> number] =
  (LAMBDA i, n, F :
     IF n = 0 THEN 0 ELSE sigma(i, pred(n), F) + F(i + pred(n)) END IF)
  BY middle
```

The function middle, whose value is the second (or middle) of its three arguments, is specified as the measure function. EHDM generates the termination TCC

```
sigma_TCC1: FORMULA
  (NOT (n = 0)) IMPLIES middle(i, n, F) > middle(i, pred(n), F)
```

supporting theory that are needed to solve the actual problem at hand. The alternative is to simply assert as axioms the results that are actually needed from the supporting theory. The danger here is self-evident—it is remarkably easy to state plausible, but false axioms.

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

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

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

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

If $j < i$, the value of sum is intended to be zero. The EHDM specification of the function `sum` defines it in terms of the more primitive function `sigma` that is described in the next subsection.

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

The first, `split_sum`, states that under suitable conditions a summation from i to j is equal to the sum of two smaller summations: one from i to k, and the other

4.2.1.7 Natinduction

This module derives specialized induction schemes over the natural numbers from the general Noetherian induction scheme introduced in the previous module. The main result is the lemma `induction`, which states the rule of simple induction over the natural numbers. The proof of this result is very similar to that for the formula of the same name in the module `simple_induction` that is described in the EHDM tutorial [24, Chapter 6]. That reference may be consulted, in particular, for a description of the way in which the assumptions on the module `noetherian` are discharged.

The formula `induction2` is similar to `induction`, but applies to predicates of two arguments, where the induction is performed over the second argument. Its proof, which involves a "currying" function, may be of interest. This induction scheme, and also the one called `mod_induction1`, are both used in the module `sigmaprops`.

`Mod_induction1` is derived from a formula called `mod_induction_m`, which itself is derived from one called `induction_m`, which in turn follows from `induction`. `Induction_m` is the same as induction, but with the value `m`, rather than zero, as the base case. The modified scheme `mod_induction_m` is a specialization of `induction_m` for the proof of predicates of the form $A(i) \supset B(i)$. The inductive step in such cases has the form

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

This is equivalent to

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

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

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

This is the form for the inductive step that is stated in `mod_induction_m` and proved in `mod_m_proof`. The lemma `mod_induction1` is simply the specialization to the case `m = 1`.

A couple of other formulas, `mod_induction` and `induction1`, are stated and proved in the module, but are not used in the present verification.

4.2.1.8 Sums and Sigmaprops

Choosing how primitive the axiomatic basis for a supporting theory should be is a matter of taste, conscience, and the time and funds available. Ideally, each supporting theory should be built up from a small and primitive set of self-evident, well-accepted axioms. Unfortunately, it may then require a considerable expenditure of time and effort to build the body of verified lemmas and theorems for the

We also use the built-in function `pred`, defined by

```
pred: function[nat -> nat] = (LAMBDA n -> nat:
   IF n > 0 THEN n - 1 ELSE 0 END IF)
```

Several derived properties of these two functions are stated and proved in the module `natprops`.

4.2.1.5 Functionprops

The module `functionprops` defines the (higher-order) axiom of function extensionality. This is required for one of the proofs (`mod_sigma_mult_proof`) in the module `sigmaprops`. We define this axiom for functions of type `A -> B`, where `A` and `B` are type parameters to the module `functionprops` as follows:

```
F, G: VAR function[A -> B]
x: VAR A
extensionality: AXIOM (FORALL x : F(x) = G(x)) IMPLIES F = G
```

This example demonstrates some of the power and also some of the limitations to EHDM's module parameterization and type system. The ability to parameterize modules by types and constants gives most of the advantages of polymorphism and dependent typing within a very simple semantic framework. Its limitations are exemplified by the fact that the axiom `extensionality` applies only to functions of arity 1.[10]

4.2.1.6 Noetherian

This module introduces Noetherian induction. It is a standard module described in the EHDM tutorial [24, Chapter 6], and provides the basis for all the specialized induction schemes that are introduced in the `natinduction` module. Notice that the `noetherian` module is parameterized by an arbitrary type and a relation on that type. In order for induction to be sound, we need to ensure that the relation is well-founded. This is accomplished by the `ASSUMING` clause, which requires a measure function to be exhibited for any instantiation of the module.

The statement of Noetherian induction is taken from the text by Manna and Waldinger [20, page 6], as is the method for demonstrating well-foundedness [20, pages 8–9]. For more discussion of Noetherian induction see the EHDM tutorial [24].

---

[10] An experimental verification system under construction at SRI avoids this limitation by supporting tuple types.

when {`y <- 1/z`} would seem more natural. The explanation is that all terms of the form `1/z` generate a TCC requiring the user to establish `z /= 0`. There will presumably be conditions in other premises, or the conclusion, that do establish `z /= 0`. However, the generation of TCCs is performed in a very local context, and the TCC produced from the substitution will be simply `z /= 0`—which is unprovable. The more complicated substitution actually used generates the provable TCC

```
IF z = 0 THEN 1 ELSE z END IF /= 0
```

The more complex substitution will cause no difficulty in the original proof since the fact `z /= 0`, established elsewhere (for example, the conclusion to `quotient_pos_proof` has `z > 0` as a hypothesis), immediately reduces it to the simple case.

The quantity $\frac{\rho}{2}$ appears frequently in the proof. We encode this in the function `half` ($\frac{x}{2}$)that is also introduced in the module `arithmetics`.

4.2.1.3 Absolutes

Absolute values are used extensively in the specification. The absolute value function is introduced in the module `absolutes` by the definition

```
a: VAR number
abs: function[number -> number] =
        (LAMBDA a -> number: IF a<0 THEN -a ELSE a END IF)
```

Several useful lemmas are then stated and proved from this definition.

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

4.2.1.4 Natprops

EHDM does not define a subtraction operator on the natural numbers. The naturals are a subtype of the integers in EHDM, so that the expression `n - m`, where `n` and `m` are naturals, is interpreted by coercing those values to type integer, and then applying the integer subtraction operator to yield an integer result. If this appears in a context where a natural is needed, a TCC will be generated requiring `n > m` to be proved.

In our treatment of summations, it is more convenient to define a direct subtraction-like operator on the naturals. This is the function `diff`, which is introduced in module `natprops` by the definition

```
diff: function[nat, nat -> nat] = (LAMBDA n, m -> nat:
   IF n >= m THEN n - m ELSE 0 END IF)
```

half-open interval $[0, 1)$. These types are used extensively in the specification; for example, `n`, the number of processors, is of type `posint`, `R`, the duration of a period, is of type `posnum`, and `rho` (ρ), the bound on the rate of clock drift, is of type `fraction`. By incorporating the constraints on these values in their types, we avoid the need to separately axiomatize and cite those constraints in proofs, and we increase the effectiveness of typechecking.

The system-generated module `numeric_types_tcc` contains the formulas that must be proved in order to discharge the obligation to ensure that the subtypes introduced in `numeric_types` are nonempty. Successful proofs for these formulas (the trivial system-generated proofs are inadequate) are provided in the module `numeric_types_tcc_proofs`.

4.2.1.2 Arithmetics

Although we said earlier that most of the arithmetic needed was built-in to EHDM, we were not quite telling the truth. The ground prover of EHDM is a decision procedure for *linear* arithmetic: multiplication is decided only when one of the factors is a literal constant. Several of the formulas and constraints needed in the specification and verification of the Interactive Convergence Clock Synchronization Algorithm require use of nonlinear multiplication and division—e.g., terms such as $\frac{n\rho R}{n-m}$ appear in the constraint C6.

Earlier versions of the EHDM ground prover did not interpret division at all, and provided a rather clumsy treatment of nonlinear multiplication. Enhancements due to David Cyrluk provide much improved treatment of these functions in the more recent versions of EHDM. The current version of the EHDM ground prover reduces division by x (say) to multiplication by $1/x$, and automatically applies the cancellation law $x * 1/x = 1$. Because normalization may not always juxtapose a term and its reciprocal, it is sometimes necessary to explicitly cite an instance of the cancellation law in proofs (see Section 4.2.3.4).

Nonlinear multiplication is treated as an uninterpreted function, but the laws of associativity, commutativity, and distributivity are applied and suffice to prove many formulas.

The main purpose of the module `arithmetics` is to introduce an axiom concerning nonlinear multiplication that is needed in the verification. The axiom is `mult_pos`, which describes the conditions under which the product $x * y$ is strictly positive (i.e., x and y must both be strictly positive, or both strictly negative). Several lemmas that are consequences of this axiom are also stated and proved.

Note that some of the proofs (for example, `quotient_pos_proof`) use substitutions such as

```
{y <- 1 / IF z = 0 THEN 1 ELSE z END IF}
```

in order to get on with the main problem. With a formal specification system, the built-in "background knowledge" is generally very limited (usually it is little more than predicate calculus with equality) and the construction of explicit specifications for the supporting theories may often consume the greater part of a specification effort. It has been recognized for a long time that the development of certified libraries of generally useful supporting theories would be one of the most useful contributions to reducing the cost and increasing the reliability of formal specifications. The module library mechanism of the EHDM system provides a suitable framework for standard modules; however, the libraries have not yet been populated.

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

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

Our specification and verification of the Interactive Convergence Clock Synchronization Algorithm is described in the three subsections following. First we describe the EHDM modules that provide the supporting theories, then those that build up the specification of the Algorithm, and finally those that develop the proof that the Algorithm maintains synchronization. Ten of the specification modules generate TCC modules. These are mostly routine; explanations of interesting cases are provided in the description of the parent module concerned. Listings of all the specification modules are given in LaTeX-printed form in Appendix C and in raw form in Appendix D. Cross-references are provided in Appendix A. In the descriptions below, we indicate both the raw and (in parentheses if it has one) the LaTeX-printed form of each identifier.

4.2.1 Supporting Theories

Nine modules provide supporting theories for the specification.

4.2.1.1 Numeric_types

This module introduces three subtypes of the built-in numeric types: `posint` and `posnum` are the subtypes of `integer` and `number` respectively, comprising just the strictly positive values; `fraction` is the subtype of `number` comprising just the example: **Module**

Using time

Exporting proc, $c_{\star 1}(\star 2)$, ρ, goodclock **with** time

Theory

proc: **Type is** nat

p: **Var** proc

$c_{\star 1}(\star 2)$: function[proc, clocktime $\rightarrow$ realtime]

T, T_0, T_1, T_2, T_N: **Var** clocktime

ρ: fraction

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

rho_small: **Lemma** $\frac{\rho}{2} < 1$

goodclock: function[proc, clocktime, clocktime $\rightarrow$ bool] =

$$
\begin{array}{l}
(\,\lambda\, p, T_0, T_N : \\
\quad (\,\forall\, T_1, T_2 : \\
\quad\quad T_0 \leq T_1 \wedge T_0 \leq T_2 \wedge T_1 \leq T_N \wedge T_2 \leq T_N \\
\quad\quad\quad \supset |c_p(T_1) - c_p(T_2) - (T_1 - T_2)| \leq \frac{\rho}{2} * |T_1 - T_2|))
\end{array}
$$

Proof

rho_pos_proof: **Prove** rho_pos **from** fraction_invariant $\{r \leftarrow \rho\}$

rho_small_proof: **Prove** rho_small **from** fraction_invariant $\{r \leftarrow \rho\}$

End example

Figure 4.2: LaTeX-printed Example EHDM Specification Module

able specifications, with two-dimensional layout including sub- and superscripts and "mix-fix" function symbols. For example, a functional expression in EHDM

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

can be converted to the more comprehensible notation

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

In our LaTeX-printed specifications, we adopt the convention that when an uninterpreted function name appears alone (for example, in a declaration), it is printed as a template indicating argument positions. Thus, for example, $\Delta_{\star 1,\star 2}^{(\star 3)}$ makes it clear that the first argument will appear as a subscript, the second as a parenthesized superscript, and the third in normal parentheses. When a function is given an interpretation, we replace the $\star i$'s with the corresponding bound variables from its definition—for example, $\bar{\Delta}_{r\,p}^{(i)}$. The LaTeX-printed version of the example from Figure 4.1 is shown in Figure 4.2 on page 45.

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

Cross-Reference Tools. There are 472 EHDM identifiers declared in our specification of the Interactive Convergence Clock Synchronization Algorithm. Keeping track of the declaration and uses of these identifiers could become quite burdensome, so the EHDM environment provides simple cross-reference functions to assist in this task. Two of these functions allow the user to locate and jump to the declarations and uses, respectively, of a given identifier; the third provides a tabular cross-reference to all declarations in a given EHDM library. (EHDM allows specification modules to be collected into "libraries" and manipulated as a group.)

The table produced by this third function of the EHDM cross-reference tool appears as Table A.2 in Appendix A.

4.2 The Formal Specification and Verification of the Algorithm

A formal specification generally divides into two components: one directly concerned with the problem at hand, and another in which are developed all the "supporting theories" needed in the first but peripheral to its main purpose. The supporting theories provide the "background knowledge" that we would like to be able to assume needed to make a proof succeed. If it finds the correct substitutions, it can write them back into the specification text so that in future the ground prover will be able to perform the proofs on its own.

The instantiator is a full first-order theorem prover: it can prove any true theorem of first-order predicate calculus. However, its effectiveness in finding suitable substitutions is considerably diminished in the presence of interpreted symbols, such as those for equality and arithmetic. Since the specifications of the Interactive Convergence Clock Synchronization Algorithm make heavy use of arithmetic, we did not use the instantiator in this effort. It was the powerful arithmetic capabilities of the EHDM ground prover that were crucial to our ability to perform this verification.[9]

4.1.1.4 Other Components of the EHDM System used in the Proof

Proof-Chain Analyzer. The notion of "proof" that is established by the EHDM theorem prover is a local one: it assures us that the conclusion is indeed a valid consequence of the premises. But it does not tell us whether those premises are axioms or theorems, and if the latter, whether or not they have been proved. Nor does it tell us whether module assumptions have been discharged and all TCC formulas proven. This larger scale analysis is performed by an EHDM tool called the "Proof-Chain Analyzer." The Proof-Chain Analyzer can be invoked with either a `PROVE` or a `FORMULA` declaration as its target. In the latter case, it first searches for a proof of the formula concerned; in either case it then recursively examines the status of all the premises named in the proof. The output of proof-chain analysis includes an enumeration of all the axioms, definitions, and proved intermediate formulas on which a given proof or theorem ultimately depends. Proof-Chain analyses for the clock synchronization conditions in our specification are given in Appendix B.

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

Even given the relatively straightforward concrete syntax of EHDM, it can still be hard to read specifications composed of long series of function applications. Thus, EHDM includes a table-driven "LaTeX-printer" that converts EHDM specifications into LaTeX input which can then be processed by LaTeX [17] to produce very read-

[9]The successful completion of this and other hard verifications attest to the power of the EHDM theorem prover; it is, however, undeniably hard to use. EHDM Version 6 will include an interactive proof-construction facility that should ease the development of proofs. We have also developed a fully interactive theorem prover for experimental purposes.

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

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

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

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

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

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

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

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

It will be clear from our description that the ground prover of EHDM is simply a sophisticated proof checker: all the creative work is in the selection of the premises and of the substitutions—and this is performed by the user. EHDM contains another theorem proving component called the *instantiator* that can perform some of these tasks automatically. Specifically, the instantiator tries to supply the substitutions

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

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

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

The EHDM ground prover is a refutation-based prover; its strategy is to attempt to show that (4.1) (i.e., the conjunction of the premises and the negated conclusion) is unsatisfiable. The first step on the way to accomplishing this goal is to reduce (4.1) to an equivalent quantifier-free form by the process of Skolemization. The details of Skolemization are somewhat tedious to describe (see [20] for a general explanation or [7, Chapter 5] as it applies to EHDM), but the important point is that the existentially quantified variables in the premises, and the universally quantified and unquantified variables in the conclusion, are replaced by constants.[7]

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

The ground prover of EHDM is simply a decision procedure for the combination of propositional calculus with equality over uninterpreted function symbols, plus "extended quantifier-free Presburger arithmetic[8] for both the rationals and integers" [28]. Proof declarations for the EHDM ground prover must indicate the substitutions to be used to produce the ground instance that is submitted to the ground prover. Substitutions are indicated as follows:

---

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

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

```
example: MODULE

USING time

EXPORTING proc, clock, rho, goodclock
  WITH time

THEORY

  proc: TYPE IS nat

  p: VAR proc

  clock: function[proc, clocktime -> realtime]

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

  rho: fraction

  rho_pos: LEMMA half(rho) >= 0

  rho_small: LEMMA half(rho) < 1

  goodclock: function[proc, clocktime, clocktime -> bool] =
    (LAMBDA p, T0, TN :
       (FORALL T1, T2 :
          T0 <= T1 AND T0 <= T2 AND T1 <= TN AND T2 <= TN
            IMPLIES abs(clock(p, T1) - clock(p, T2) - (T1 - T2))
              <= half(rho) * abs(T1 - T2)))

PROOF

  rho_pos_proof: PROVE rho_pos FROM fraction_invariant{r <- rho}

  rho_small_proof: PROVE rho_small FROM fraction_invariant{r <- rho}

END example
```

Figure 4.1: An Example EHDM Specification Module

similar units (e.g., packages in Ada). A module serves to group related concepts together and delimits the scope of names. An (unparameterized) EHDM module consists of three parts, any of which may be empty: an import/export part, a theory part, and a proof part.

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

```
EXPORTING R, in_R_interval
```

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

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

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

Modules may be parameterized by types and constants; an `ASSUMING` clause may be supplied in order to state semantic constraints on actual parameters that may be supplied. For example, the declaration

```
mod: MODULE [a, b: nat]
ASSUMING
  ordered: FORMULA a < b
```

specifies that in any instantiation of the module `mod`, the value of the first actual parameter must be strictly less than that of the second. Formulas stated in `ASSUMING` clauses may be treated as axioms within the module, but generate proof obligations whenever the module is instantiated. These obligations are enforced by the Proof-Chain Analyzer (Section 4.1.1.4).

The reader should now have enough understanding of the specification language of EHDM to be able to read the simple module `example`, which is a simplified form of the module `clocks` used in the actual specification of the Interactive Convergence Clock Synchronization Algorithm. The module is shown in Figure 4.1 on page 40.

4.1.1.3 Proofs

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

A proof declaration in EHDM has the general form it is addition on the integers that is supplied as the interpretation of the `+` sign (`n` is coerced to integer), the result is coerced to a (rational) number, and the equality function used is that for the (rational) numbers.

In addition to the subtype-to-supertype coercion, a supertype-to-subtype coercion is also provided. For example, notice that the constraint C6 involves several division operations, with divisor `n-m`. An EHDM term such as `x / (n - m)` where `x` is a `number` and `m` and `n` are `naturals` is interpreted as follows. First, since there is no subtraction operation defined on the naturals, `m` and `n` are promoted to integers (a supertype of naturals), and `n - m` is interpreted as integer subtraction, yielding an integer result. Then, since functions are total in EHDM, we need to ensure that the division `x / (n - m)` is well-defined—i.e., that `n - m` is nonzero. The signature of the division operator is defined as follows

```
x: VAR number
nonzeronum: TYPE FROM number WITH (LAMBDA x: x /= 0)
/: function[number, nonzeronum -> number]
```

In the example, we are supplying an integer as the second argument to the division operator, which requires a nonzero number in this position. The numbers (rationals) are a common supertype to both the integers and the nonzero numbers, so `n - m` can be promoted to a number, and then reduced to the nonzero number required for type-correctness if we can prove the theorem `(n - m) /= 0` (which follows obviously from the constraint C0).

Theorems such as this are called *Type Consistency Conditions*, or TCCs, and are generated automatically by the EHDM typechecker in order to establish the type-correctness of supertype to subtype coercions, the nonemptiness of subtypes, and the termination of recursive function definitions. All type correctness conditions generated during the typechecking of a module are collected and written as formulas in a system-generated module, called a TCC module. The TCCs are necessary and sufficient to ensure type correctness of their parent module; the specification is incomplete until they have been proved. The requirement to prove all TCC formulas is enforced by the Proof-Chain Analyzer (see Section 4.1.1.4). Trivial proof declarations are generated automatically for each TCC in a TCC module. Often, these suffice to prove the TCC formulas; when they do not, users may attempt to prove the formulas concerned in the same way they would attack proofs of ordinary lemmas.[6]

4.1.1.2 Modules

Specifications in EHDM are structured into named units called *modules* in much the same way as programs written in modern programming languages are composed of

[6] In EHDM Version 6, TCCs are added to the parent module. Those that can be proved trivially are suppressed.

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

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

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

Quantified expressions are written in the following form:

```
monotonicity: THEOREM
  (EXISTS T0, TN :
       goodclock(p, T0, TN)
         AND T0 <= T1 AND T0 <= T2 AND T1 <= TN AND T2 <= TN)
    IMPLIES (T1 > T2 IMPLIES clock(p, T1) >= clock(p, T2))
```

Free variables in EHDM formulas are treated as if they are universally quantified at the outermost level (i.e., formulas denote their universal closure). Thus, `T1, T2` and `p` are implicitly universally quantified in the formula `monotonicity` show above. It is generally easier to read formulas when this outer level of quantification is omitted.

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

```
n: VAR nat
i: VAR int
r: VAR number
X: FORMULA r = i + n
```

[4]The Proof-Chain Analyzer (see Section 4.1.1.4) distinguishes `AXIOM` from the other keywords: `AXIOMS` are not expected to be proved, but all non-`AXIOM`s are ultimately required to be consequences only of `AXIOM`s and completed `PROOF`s. The keywords other than `AXIOM` are all synonymous with each other and may be used to suggest the role of the corresponding formula.

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

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

```
T_ZERO: clocktime = 0
```

Function types are written as follows:

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

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

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

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

Functions are simply constants of a function type:

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

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

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

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

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

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

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

Constant declarations may use `=` or `==`; the latter indicates that the definition is to be expanded in place wherever it appears (like a macro), whereas the former indicates that the definition should only be expanded on command. Recursive definitions are allowed (in declarations using `=`), but must be provided with a *measure function* that is used to ensure termination. This is explained in more detail in Section 4.2.1.8.

Formula declarations have the following schema:

name: KEYWORD value

4.1.1 The Specification Language

The fragment of the EHDM specification language used here is a strongly typed version of the First-Order Predicate Calculus, enriched with elements of other logics—specifically Higher-Order Logic and the Lambda Calculus. The two volumes by Manna and Waldinger [19, 20] provide an introduction to some of these topics that is especially suitable for computer scientists; Andrews [1] gives a more detailed treatment, including a good discussion of Higher-Order Logic.

4.1.1.1 Declarations

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

The declaration

```
clocktime: TYPE IS number
```

introduces clocktime[3] as a synonym for the rational numbers (equivalently, we can think of the rational numbers as supplying the interpretation for the type `clocktime`).

Subtypes of previously defined or built-in types can be introduced and may be associated with a predicate. The natural numbers are provided as a subtype of the integers associated with the predicate $(\lambda x : x \geq 0)$ using the (built-in) declaration

```
nat: TYPE FROM int WITH (LAMBDA x: x >= 0)
```

Variables are introduced by declarations of the form

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

while uninterpreted constants are introduced by declarations of the form

```
T_ZERO: clocktime
```

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

Chapter 4

Formal Specification and Verification in EHDM

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

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

4.1 Overview of EHDM

EHDM is an interactive system for the composition and analysis of formal specifications and abstract programs written in the EHDM specification language [24]. The specification and verification described here was performed on a Sun workstation using EHDM Version 5.2.0.[1] Other substantial verifications of fault-tolerance properties that have been undertaken in EHDM are described in [23, 27].

Our specification and verification of the Interactive Convergence Clock Synchronization Algorithm uses only some of the capabilities of EHDM. Specifically, it uses the functional component of the specification language, the ground prover, and the proof-chain analyzer.[2]

[1] A complete reimplementation, EHDM Version 6, is now in alpha test.

[2] Capabilities not used here include hierarchical mappings, the procedural component of the specification language, the instantiator for the theorem prover, the Hoare-Sentence prover, the Ada Translator, and the multilevel security analyzer.

It should read

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

These seem to be no more than typographical errors.

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

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

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

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

3.5 Sundry Minor Flaws and Difficulties

3.5.1 Falsehood and Unnecessary Generality of Lemma 3

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

3.5.2 Missing Requirements for Clock Synchronization Condition S2

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

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

3.5.3 Typographical Errors in Lemmas 2 and 4

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

The conclusion to Lemma 4 is stated as

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

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

$$T_0 + \Delta_{q\,p} \leq T^{(i+1)}.$$

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

$$T_0 + \Delta_{q\,p} \leq T^{(i+2)}. \tag{3.2}$$

Since $T_0 \leq T^{(i+1)}$ and $T^{(i+2)} = T^{(i+1)} + R$ and $\Delta_{q\,p}$ is small, this seems straightforward. However, although $\Delta_{q\,p}$ is assumed small, and the purpose of this very Lemma is to show it is less than Δ, there is no *a priori* bound on its value and therefore no basis to establish (3.2).[1] Hence, this putative proof of even the repaired version of Lemma 1 is flawed. In our proof, we introduce

$$\Delta_{q\,p}^{(i)} \leq S$$

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

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

3.4.2 Falsehood of Lemma 2

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

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

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

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

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

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

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

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

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

where

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

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

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

Lemma 1: The conclusion is changed from

$$|\Delta_{q\,p}| \lesssim \delta + \epsilon$$

to

$$|\Delta_{q\,p}^{(i)}| < \Delta$$

Lemma 4: The conclusion is changed from

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

to

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

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

Several lemmas use Definition 1 (the notion of a good clock) and Assumption A1 (a nonfaulty processor has a good clock) to establish bounds on certain quantities. In order to apply these definitions, we must establish that the times concerned fall in the interval during which the processor is hypothesized to be nonfaulty. The statements and proofs of Lemmas 1 and 2 [16, page 64] do not do this with sufficient care and both are false as stated.

3.4.1 Falsehood of Lemma 1

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

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

3.3.2 A Flaw in Lemma 4

Lamport and Melliar-Smith's version of Lemma 1 [16, page 64] establishes, under suitable hypotheses, that $|\Delta_{qp}| \lesssim \delta + \epsilon$. However, their proof of Lemma 4 [16, page 65] requires $|\Delta_{qp}| < \delta + \epsilon$, which is not substantiated by these premises. □

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

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

Constraint C5 is changed from

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

to

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

Constraint C4 is changed from

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

to

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

Constraint C6 is formulated as follows by Butler et al. [6]:

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

change is a technical correction necessitated by our use of a strict formalism. An alternative in the case of Δ_{qp} would have been to include it in the scope of the existential quantification in A2 (Skolemization would then have provided the functional dependence on i), but that would have needlessly complicated the technical details of the argument.

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

3.3 Approximations and Neglect of Small Quantities

In order to "simplify the calculations" Lamport and Melliar-Smith make approximations based on the assumption that $n\rho \ll 1$. They neglect quantities of order $n\rho\epsilon$ and $n\rho^2$ [16, Section 3.4] and use the notation $x \approx y$ to indicate approximate equality and $x \lesssim y$ to indicate approximate inequality. ($x \lesssim y$ means $x < y'$ for some $y' \approx y$.)

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

3.3.1 A Flaw in the Main Induction

The goal of the main induction is to establish the clock synchronization condition S1. This is stated [16, page 63] as

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

while the inductive step [16, page 66] establishes

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

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

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

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

Accordingly, we follow Butler [5] and use the Mean-Value Theorem to provide a more tractable definition:

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

This formulation avoids the use of derivatives, but still requires use of the inverse clock function in order to define T_1 and T_2. This can be avoided by defining the notion of a good clock relative to a *clock time* interval:

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

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

$$|c(T_1) - c(T_2) - (T_1 - T_2)| \leq \frac{\rho}{2}\,|T_1 - T_2| \tag{3.1}$$

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

Notice also that although we do not now *explicitly* require a good clock to be monotonic, it follows implicitly as a corollary to our definition that, since ρ is small, the clock function c is strict monotonic increasing (and therefore has an inverse function). This fact is proved as Theorem `monotonicity` in Module `clocks`.

In the original version of the verification and report, we used $<$ rather than $\leq$ in definition (3.1). Bill Young pointed out that this has the effect of excluding systems with perfect clocks (i.e., those with $\rho = 0$). The consequences of the changed definition are that $\leq$ replaces all the formerly strict inequalities in Lemmas 2, 2a–2d, Sublemma A, and Lemma 6.

A more important defect of the original definition is that it is unsatisfiable in the case $T_1 = T_2$. This causes the entire verification to be predicated on an unsatisfiable assumption and thereby rendered the original verification potentially specious. This is discussed further on page 61.

3.2 Explicit Functional Dependencies

We made the functional dependency on i, the synchronization period, explicit in the three subscripted Δ quantities that appear in the Algorithm: where Lamport and Melliar-Smith use Δ_p, Δ_{qp} and $\bar{\Delta}_{qp}$, we use $\Delta_p^{(i)}, \Delta_{qp}^{(i)}$ and $\bar{\Delta}_{qp}^{(i)}$. Thus, $\Delta_{qp}^{(i)}$ is the difference between q's clock and p's observed by p *during the i'th period.* This

Chapter 3

Comparison with the Published Analysis by Lamport and Melliar-Smith

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

Our proof of the correctness of the Interactive Convergence Clock Synchronization Algorithm, which was presented in the previous chapter, follows the original proof of Lamport and Melliar-Smith [16] very closely; our only changes are technical ones. Some of these were motivated by the needs of truly formal specification and verification; others were motivated by the need to correct flaws in the original. We begin with changes in the first class, then describe the flaws we discovered in the published proof.

3.1 The Definition of a Good Clock

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

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

This definition obviously presents a considerable challenge for a completely formal specification—it would require axiomatizing a fragment of the differential calculus.

2.3.2.3 The Correctness Theorem

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

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

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

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

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

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

By the Algorithm, the right hand side equals

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

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

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

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

Proof: The Algorithm defines

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

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

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

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

By the Algorithm,

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

Using the arithmetic identity

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

we obtain

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

and hence

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

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

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

and the result follows. □

Proof: Using the arithmetic identity

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

we obtain

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

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

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

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

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

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

we have

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

and hence

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

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

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

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

The result then follows from $|\Pi| \leq S$. □

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

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

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

$$|c_p^{(i)}(T) + \bar{\Delta}_{r\,p}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{r\,q}^{(i)}]| = |c_p^{(i)}(T) + \Delta_{r\,p}^{(i)} - [c_q^{(i)}(T) + \Delta_{r\,q}^{(i)}]|.$$

Using the arithmetic identity

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

we obtain

$$\begin{aligned}
&|c_p^{(i)}(T) + \Delta_{r\,p}^{(i)} - [c_q^{(i)}(T) + \Delta_{r\,q}^{(i)}]| = \\
&\quad|\quad c_q^{(i)}(T + \Delta_{r\,q}^{(i)}) - [c_q^{(i)}(T) + \Delta_{r\,q}^{(i)}] \\
&\qquad - (c_p^{(i)}(T + \Delta_{r\,p}^{(i)}) - [c_p^{(i)}(T) + \Delta_{r\,p}^{(i)}]) \\
&\qquad + c_p^{(i)}(T + \Delta_{r\,p}^{(i)}) - c_r^{(i)}(T) \\
&\qquad - (c_q^{(i)}(T + \Delta_{r\,q}^{(i)}) - c_r^{(i)}(T))|
\end{aligned}$$

and so

$$\begin{aligned}
|c_p^{(i)}(T) + \Delta_{r\,p}^{(i)} - [c_q^{(i)}(T) + \Delta_{r\,q}^{(i)}]| &\leq \\
&|c_q^{(i)}(T + \Delta_{r\,q}^{(i)}) - c_q^{(i)}(T) + \Delta_{r\,q}^{(i)}| \\
+ \quad &|c_p^{(i)}(T + \Delta_{r\,p}^{(i)}) - c_p^{(i)}(T) + \Delta_{r\,p}^{(i)}| \\
+ \quad &|c_p^{(i)}(T + \Delta_{r\,p}^{(i)}) - c_r^{(i)}(T)| \\
+ \quad &|c_q^{(i)}(T + \Delta_{r\,q}^{(i)}) - c_r^{(i)}(T)|.
\end{aligned}$$

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

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

$$|c_p^{(i)}(T) + \bar{\Delta}_{r\,p}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{r\,q}^{(i)}]| < \delta + 2\Delta.$$

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

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

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

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

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

Proof: By A2, we have

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

and

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

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

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

we obtain:

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

Hence

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

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

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

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

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

□

The final variation on Lemma 2 is Lemma 2d. Unlike the other variations, this lemma is not a specialization of Lemma 2, but is proved independently.

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

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

Proof: By an inductive argument similar to that used for Lemma 2a, we can show that $A_p^{(i)}(T^{(i)}+\Pi)$ is in the interval $[A_p^{(0)}(T^{(0)}), A_p^{(i)}(T^{(i+1)})]$. The result then follows from the definitions of *nonfaulty* and *good clock* similarly to Lemma 2. □

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

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

Proof: By A2, we have

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

and

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

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

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

we obtain

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

Hence

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

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

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

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

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

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

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

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

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

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

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

We have just sketched the proof of

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

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

□

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

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

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

□

The special case $\Phi = 0$ provides

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

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

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

The reader may wonder why we do not include the celebrated constraint $3m < n$. The reason is simply that this is a derived constraint, not a fundamental one. It is easy to see (cf. (2.2) on page 9) that C4 and C6 can be satisfied simultaneously only if indeed $3m < n$, but it is also quite possible for values of other parameters to render C4 or C6 unsatisfiable even if $3m < n$.

2.3.2.2 The Lemmas

We derive Lamport and Melliar-Smith's Lemma 1 from Lemma 2—hence we begin with Lemma 2.

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

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

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

$$|(c_p(A_p^{(i)}(T+\Pi)) - c_p(A_p^{(i)}(T))) - \Pi| \leq \frac{\rho}{2}|\Pi|,$$

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

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

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

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

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

2.3.2 The Proof in Detail

We now prove that the Interactive Convergence Clock Synchronization Algorithm maintains the clock synchronization conditions S1 and S2. The proof closely follows that of Lamport and Melliar-Smith [16, pages 64–66] (though we do separate the two synchronization conditions and prove them individually as Theorems 1 and 2, respectively). In particular, our Lemmas 1–5 correspond exactly to (corrected versions of) theirs. However, since we use Lemma 2 in the proof of Lemma 1, we rearrange the order of presentation accordingly. We also introduce a Lemma 6 and a Sublemma A that is used in its proof and also in the base case of the inductive proof of condition S1. Lamport and Melliar-Smith subsumed both of these in the proof of their main theorem. In addition, we distinguish several special cases for Lemma 2, which we identify as Lemmas 2a–2d. (Lemma 2c is the one that corresponds most closely to Lemma 2 in [16].) The reason for these additional lemmas is that the single Lemma 2 of Lamport and Melliar-Smith is false: more restrictive hypotheses are needed, and it is convenient to create specialized instances for the different circumstances of its application.

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

2.3.2.1 Constraints on Parameters

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

The second and third constraints (i.e., C1 and C2) can be modified (but not eliminated) if desired by suitably adjusting some of the proofs; we chose these particular constraints for simplicity and because we felt that there would be no difficulty satisfying them in any likely implementation. The other five constraints are fundamental to the operation and analysis of the Algorithm.

C0: $m > n \geq 0$

C1: $R \geq 3S$

C2: $S \geq \Sigma$

C3: $\Sigma \geq \Delta > 0$

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

$$|\Delta_{q\,p}^{(i)}| < \Delta.$$

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

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

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

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

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

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

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

$$|c_p^{(i)}(T) + \bar{\Delta}_{r\,p}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{r\,q}^{(i)}]| < \delta + 2\Delta.$$

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

then the nonfaulty processors will remain synchronized through that next period. The actual proof is a mass of details, so it will be helpful to sketch the basic approach first. For reference, the statements of the main Lemmas are collected in Figure 2.1.

2.3.1 Overview of the Proof

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

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

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

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

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

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

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

$$|c_p^{(i)}(T) + \bar{\Delta}_{r\,p}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{r\,q}^{(i)}]|. \tag{2.4}$$

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

$$|c_p^{(i)}(T) + \Delta_{r\,p}^{(i)} - c_r^{(i)}(T) - [c_q^{(i)}(T) + \Delta_{r\,q}^{(i)} - c_r^{(i)}(T)]|.$$

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

$$|c_p^{(i)}(T) + \Delta_{r\,p}^{(i)} - c_r^{(i)}(T)|$$

which, using Lemma 2, can be approximated by

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

Lemma 3 is concerned with quantities of this form.

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

Table 2.1: Notation, Parameters, and Concepts

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

Table 2.2: Typical Values for the Parameters

and

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

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

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

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

Algorithm: For all processors p:

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

where

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

A summary of the notation and definitions introduced so far is given in Table 2.1 on Page 14. Some typical values for the parameters, based on an experimental validation using the SIFT computer [6], are given in Table 2.2 on Page 14.[6]

2.3 Proof that the Algorithm maintains Synchronization

We now need to prove that the Interactive Convergence Clock Synchronization Algorithm maintains the clock synchronization conditions S1 and S2. Condition S2 is easy; the difficult part of the proof is to show that the Algorithm maintains Condition S1. The proof is an induction on i—we show that if the clocks are synchronized through period i, and if sufficient processors remain nonfaulty through period $i+1$,

[6]The closeness of the synchronization that can be achieved is dominated by the parameter ϵ. With modern hardware, it is possible to reduce this value far below the 66.1 μsec. achieved for SIFT. Modern hardware also supports much higher frame rates (i.e., smaller R and S) than those for SIFT.

the instant its clock actually reads $A_p^{(i)}(T)$.[5] An obvious consequence is that the i'th period for processor p runs from when its *adjusted* clock reads $T^{(i)}$ until it reads $T^{(i+1)}$. That is, it is the clock time interval $[A_p^{(i)}(T^{(i)}), A_p^{(i)}(T^{(i+1)})]$. Therefore, if a processor's clock is to work long enough to complete the i'th period, it must be a good clock throughout the interval $[A_p^{(0)}(T^{(0)}), A_p^{(i)}(T^{(i+1)})]$. This motivates the following definition of what it means for a processor to be nonfaulty:

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

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

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

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

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

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

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

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

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

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

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

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

using a mechanical verification system, and do not want to have to axiomatize a fragment of the differential calculus, we use a slightly different formulation taken from Butler [5].

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

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

whenever T_1 and T_2 are clock times in $[T_0, T_N]$.

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

We consider a set of n processors, where processor p has clock c_p. As explained above, logical clock values are formed by subtracting a "correction" from the physical values; the correction used by processor p during the i'th period is denoted $C_p^{(i)}$, so that the real time at which processor p reports logical clock time T during period i is $c_p(T + C_p^{(i)})$. We denote this quantity by $c_p^{(i)}(T)$ and we call $c_p^{(i)}$ the *logical* clock for processor p during the i'th period. We call $T + C_p^{(i)}$ the *adjusted* value of T for processor p in period i and denote it by $A_p^{(i)}(T)$ (so that $c_p^{(i)}(T) = c_p(A_p^{(i)}(T))$).[4]

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

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

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

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

The process control applications that are of interest to us typically perform a schedule of many separate tasks during each period. Our goal is to ensure that tasks which are scheduled to occur on different processors at the same clock time during a particular period actually occur very close to each other in real time. To achieve this, processor p should perform a task scheduled for time T in the i'th period at

[4] In the original version of the verification and report, we followed Lamport and Melliar-Smith and defined the initial correction $C_p^{(0)}$ to be zero. This assumption proved inconvenient when we considered implementations of the algorithm. Inspection of the proof suggested that it could be eliminated, and this was confirmed in the revised formal verification. The consequences were small changes to internal lemmas, such as `adj_always_pos` in module `clockprops`, that are used to satisfy the conditions of Lemma 2 in the proof of Lemma 2a.

standard record of time maintained by international bodies. In order to distinguish these two notions of time, we will describe the internal estimate of time that may be read from a processor's clock as *clock time*, and the external notion of time as *real time*. Real time need not be directly observable; the important point is that it provides a frame of reference that is common to all clocks. Following Lamport and Melliar-Smith, we use lowercase letters to denote quantities that represent real time, and upper case for quantities that represent clock time. Thus, "second" denotes the unit of real time, while "SECOND" denotes the unit of clock time. Within this convention, Roman letters are used to denote "large" values (on the order of tens of milliseconds), while Greek letters are used to denote "small" values (on the order of tens of microseconds).

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

Some techniques for clock synchronization adjust the rates at which clocks "tick" by modifying certain hardware parameters. We are concerned with software solutions, and cannot change the rate at which the clocks "tick." What we can do is fabricate "logical clocks" that are offset from the physical clocks in order to achieve better synchronization. The offsets are adjusted periodically in order to resynchronize the logical clocks.

Suppose the physical clock c reads T' when its logical clock should read T. Then $C = T' - T$ is the offset or *correction* that should be subtracted from the physical clock reading to yield the value to be reported as the logical clock reading. Since the corrected value T is reported when the physical clock reads T' (i.e., $T + C$), we see that the logical clock time T is reported at real time $c(T + C)$.

A physical clock is a "good clock" if it runs at a rate very close to the passage of real time. Modern technology is able to provide clocks that satisfy this requirement to very high degrees of accuracy. Lamport and Melliar-Smith define a "good clock" formally in terms of the derivative of the clock function. However, since we will be

[3] For other classes of applications, the reverse notion may be more appropriate—e.g., if a single event is to be given (clock time) timestamps by different processors, then we may want the different timestamps (all triggered by the same event at the same real time) to be very close together. Lamport and Melliar-Smith [16, page 61] indicate how to convert between this notion of synchronization and the one used here.

which, when taken with (2.1), yields

$$3\epsilon \leq \frac{n-3m}{n-m}\,\Delta. \qquad (2.2)$$

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

$$n > 3m,$$

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

Lamport and Melliar-Smith raise a couple of fine points that should be considered in implementation and application of the Interactive Convergence Clock Synchronization Algorithm. The correction that occurs at each synchronization causes a discontinuity in clock values. If a correction is positive (because the clock has been running slow), then some units of clock time will vanish in the discontinuity as the correction is applied. Any task scheduled to start in the vanished interval might not occur at all. Conversely, a negative correction (for a fast clock), can cause units of clock time to repeat, possibly causing a task to be executed a second time. One solution to these difficulties is to follow each clock synchronization with a "do nothing" task of duration at least Σ. An alternative, that has other attractive properties, is to avoid the discontinuity altogether and spread the application of the correction evenly over the whole period [16, pages 54–55].

2.2 Statement of the Clock Synchronization Problem and Algorithm

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

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

Physically, a clock is a counter that is incremented periodically by a crystal or line-frequency oscillator. By a suitable linear transformation, the counter value is converted to a representation of conventional "time" (e.g., the number of seconds that have elapsed since January 1st, 1960, Coordinated Universal Time). This internal estimation of time may be expected to drift somewhat from the external,

Assumption

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

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

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

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

Thus, provided

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

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

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

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

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

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

Assumptions

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

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

Schneider [25] shows that all Byzantine clock synchronization algorithms can be viewed as different refinements of a single paradigm: periodically, the processors decide that it is time to resynchronize their clocks, each processor reads the clocks of the other processors, forms a "fault tolerant average" of their values, and sets its own clock to that value. There are three main elements to this paradigm:

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

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

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

intuition during the more formal analysis in the section that follows. Although "formal" in the sense of traditional mathematical presentations, this level of analysis is not truly formal (in the sense of being based on an explicit set of axioms and rules of inference)—that level of presentation is described in Chapter 4 and its supporting Appendices.

2.1 Informal Overview

We assume a number of components (generally called "processors") each having its own clock. Nonfaulty clocks all run at approximately the correct rate and are assumed to be approximately synchronized initially. Due to the slight differences in their running rates, the clocks will gradually drift apart and must be resynchronized periodically. We are concerned with the problem of performing this resynchronization; we are not concerned with the problem of maintaining the clocks in synchrony with some external "objective" time (see Lamport [18] for a discussion of this problem), nor are we concerned with the problem of synchronizing the clocks initially, although the closeness with which the initial synchronization is performed will limit how closely the clocks can be brought together in subsequent resynchronizations.[1]

The goal of periodic resynchronizations is to ensure that all nonfaulty clocks have approximately the same value at any time. A secondary goal is to accomplish this without requiring excessively large adjustments to the value of any clock during the synchronization process. Formalizing these two goals and the assumptions identified earlier is one of the major steps in the specification and verification of the Interactive Convergence Clock Synchronization Algorithm. For future convenience, we label and explicitly identify them here (using the same names as [16]), and give them the following informal characterizations:

Requirements

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

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

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

[2] A summary of the notation and definitions used is given in Table 2.1 on Page 14.

could thereby distribute very different values to different components in response to inquiries received very close together. Of course, one can postulate a design in which a single clock value is latched and then distributed to all other components—but then one must provide compelling evidence for the correctness of the latching mechanism and the impossibility of communication errors, and for the correctness of a clock synchronization algorithm built on these assumptions.

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

The strongest clock synchronization algorithms make no assumptions whatever about the behavior of faulty clocks. Lamport and Melliar-Smith [16] describe three such fault-tolerant clock synchronization algorithms. These algorithms work in the presence of any kind of fault—including malicious two-faced clocks such as that described above. Of course, there must not be too many faulty clocks. The first algorithm presented by Lamport and Melliar-Smith, the *Interactive Convergence Algorithm*, can tolerate up to m faults amongst $3m + 1$ clocks. Thus, 4 clocks are required to guarantee the ability to withstand a single fault. Dolev, Halpern and Strong have shown that $3m + 1$ clocks are required to allow synchronization in the presence of m faults unless digital signatures are used [11]. Thus, the Interactive Convergence algorithm requires the minimum possible number of clocks for its class of algorithms.

The Interactive Convergence Clock Synchronization Algorithm is quite easy to describe in broad outline: periodically, each processor reads the differences between its own clock and those of all other processors, replaces those differences that are "too large" by zero, computes the average of the resulting values, and adjusts its clock by that amount. For descriptions of other clock synchronization algorithms, presented in a consistent notation, see the surveys by Ramanathan [22] (which includes hardware techniques) and Schneider [25]. Implementation issues are well described by Kopetz and Ochsenreiter [14]. A new class of probabilistic clock synchronization algorithms that have extremely good performance (in terms of how close the clocks can be synchronized) has recently been introduced by Cristian [9], but so far the algorithms in this class are not tolerant of Byzantine failures.

In the next section we give an informal overview of the analysis of the Interactive Convergence Clock Synchronization Algorithm. This should support the reader's

Chapter 2

Traditional Mathematical Presentation of the Algorithm and its Analysis

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

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

Consider a system with three components: A, B, and C; A and C have good clocks, but B's clock is faulty. A's clock indicates 2.00 pm, C's 2.01 pm, and B's clock indicates 1:58 pm to A but 2.03 pm to C. A sees that C's clock is ahead of its own, and that B's is behind by a somewhat greater amount; it would be natural therefore for A to set its own clock back a little. This situation is reversed, however, when considered from C's perspective. C sees that A's clock is a little behind its own and that B's is ahead by a rather greater amount; it will be natural for C to set its own clock *forward* a little. Thus the faulty clock B has the effect of driving the good clocks A and C further apart. The behavior of B's clock that produces this effect may seem actively malicious and therefore implausible. This is not so, however. A failed clock may plausibly act as a random number generator (noisy diodes are indeed used as hardware random number generators) and required. Enhancements to the EHDM theorem prover have also allowed treatment of proofs involving division and nonlinear multiplication to be much simplified. The substance of the verification has been improved in three respects: the definition of a *good clock* has been corrected, constraints on the initial clock corrections $C_p^{(0)}$ have been eliminated, and the organization of the main induction has been revised to eliminate an apparent circularity.

1.2 Acknowledgments

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

Two of the three improvements in the substance of the verification incorporated in this revised edition of the report were suggested by Bill Young of Computational Logic Inc., who has repeated our verification using an extended version of the Boyer-Moore theorem prover [30].

David Fura of Boeing High Technology Center pointed out obscurities and opportunities for misinterpretation in the explanation of clock corrections given in the first edition, and suggested several improvements in the exposition that have been incorporated in this revised edition.

principal sources of error and difficulty was the use by Lamport and Melliar-Smith of approximations—i.e., approximate equality ($\approx$) and inequalities ($\lesssim$ and $\gtrsim$)—in order to "simplify the calculations." We eventually found that elimination of the approximations not only removed one class of errors, but actually simplified the analysis and presentation. We also found and corrected several other technical flaws in the published proof of Lamport and Melliar-Smith. Our revised presentation of the algorithm and its proof are given in Chapter 2; a discussion of the flaws found in the original is provided in Chapter 3. Some of our corrections require slight modifications to the assumptions underlying the algorithm, and to the constraints on its parameters, and thus change the external specification of the algorithm. Our formal specification and verification of the algorithm is described in Chapter 4; the detailed listings are to be found in the Appendices.

We discuss the lessons learned from this exercise, and our view of the role and utility of formal specification and verification in Chapter 5. To summarize those conclusions: we now believe the Interactive Convergence Clock Synchronization Algorithm to be correct, not because our theorem prover says it is, but because the experience of arguing with the theorem prover has forced us to clarify our assumptions and proofs to the point where we think we really *understand* the algorithm and its analysis. As a result, we can present an argument for the correctness of the algorithm, in the style of a traditional mathematical presentation, that we believe is truly compelling. This presentation is given in Chapter 2 and follows very closely the presentation given in Sections 2.1, 3, and 4 of the original paper [16, pages 53–66]. However, the details of the proof were extracted directly from our formal verification.

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

1.1 Note on the Revised Edition

This second edition of the report presents a revised version of the verification that exploits some of the features introduced into EHDM since the original verification was performed in 1988. The main benefit has been the ability to extend theories conservatively using definitions, and thereby to drastically reduce the number of axioms

Chapter 1

Introduction

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

Lamport and Melliar-Smith's paper [16] is a landmark in the field. They not only introduced the Interactive Convergence Clock Synchronization Algorithm, but two other algorithms as well, and they also developed formalizations of the assumptions and desired properties that made it possible to give a precise statement and proof for the correctness of clock synchronization algorithms. Nonetheless, the proof given by Lamport and Melliar-Smith is hard to internalize: there is much detailed argument, some involving approximate arithmetic and neglect of insignificant terms, and it is not easy to convince oneself that all the details mesh correctly. It is precisely in performing conceptually simple, but highly detailed arguments (i.e., *calculations*) that the human mind seems most fallible, and machines most effective. Consequently, the Interactive Convergence Clock Synchronization Algorithm seems an excellent candidate for mechanical verification. This report describes a mechanized proof of the correctness of the algorithm using the EHDM formal specification and verification environment.

As we performed the formal specification and verification of the Interactive Convergence Clock Synchronization Algorithm, we discovered that the presentation given by Lamport and Melliar-Smith was flawed in several details. One of the

List of Tables

List of Figures

Contents

Abstract

We describe a formal specification and mechanically checked verification of the Interactive Convergence Clock Synchronization Algorithm of Lamport and Melliar-Smith [16]. In the course of this work, we discovered several technical flaws in the analysis given by Lamport and Melliar-Smith, even though their presentation is unusually precise and detailed. As far as we know, these flaws (affecting the main theorem and four of its five lemmas) were not detected by the "social process" of informal peer scrutiny to which the paper has been subjected since its publication. We discuss the flaws in the published proof and give a revised presentation of the analysis that not only corrects the flaws in the original, but is also more precise and, we believe, easier to follow. This informal presentation was derived directly from our formal specification and verification. Some of our corrections to the flaws in the original require slight modifications to the assumptions underlying the algorithm and to the constraints on its parameters, and thus change the external specification of the algorithm.

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

Note: This second edition of the report presents a revised version of the formal specification and verification that exploits some of the features introduced into EHDM since the original verification was performed, and also improves the substance of the verification in three respects.

Formal Verification of the Interactive Convergence Clock Synchronization Algorithm[1]

John Rushby and Friedrich von Henke
Computer Science Laboratory
SRI International

Technical Report CSL-89-3R
February 1989, Revised August 1991

[1]This work was performed for the National Aeronautics and Space Administration under contract NAS1 17067.