Reasoning about TLA Actions

Undergraduate Honors Thesis

Carlos Pacheco
The University of Texas at Austin

Supervisor: J Strother Moore
May 2001

This research was supported by an IBM Partnership Award to J Strother Moore, and a UROP grant from the Educational Advancement Foundation to the Computer Sciences Department at UT Austin.

Contents

1 Algorithms, Languages, and Tools **1**

1.1 Introduction 1

1.2 The Temporal Logic of Actions 2

1.3 Finite Set Theory in ACL2 8

2 Translation **12**

2.1 Translation Guidelines 13

2.2 Translation Conventions 14

2.3 Translation Rules 17

2.4 Requirements for the TLA specifier 21

2.5 Translating the Increment Example 22

2.6 Disk Synod 24

3 Verification **27**

3.1 Consistency of Disk Synod 27

3.2 Controlling ACL2 28

3.3 Type Invariance 29

3.4 A Second Invariant 36

3.5 Issues in Verification 43

4 Conclusion **51**

4.1 Further Work 52

A The Increment Example **58**

B Disk Synod Specification **61**

C Disk Synod Translations **65**

D ACL2 Event Files **77**
D.1 additions.lisp 77
D.2 choose-max.lisp 85
D.3 common-all.lisp 90
D.4 defall.lisp 91
D.5 defexists.lisp 98
D.6 defpkg.lisp 103
D.7 hinv1.lisp 104
D.8 hinv2-exports.lisp 105
D.9 hinv2.lisp 107
D.10 hinv3.lisp 110
D.11 newdefmap.lisp 112
D.12 newpowerset.lisp 120
D.13 tla-translation-macros.lisp 124
D.14 translations.lisp 126
D.15 i2a/i2a.lisp 146
D.16 i2a/i2a-1.lisp 147
D.17 i2c/common-i2c.lisp 154
D.18 i2c/common-p12r.lisp 156
D.19 i2c/ep0.lisp 157
D.20 i2c/ep12.lisp 160
D.21 i2c/fail.lisp 162
D.22 i2c/i2c.lisp 164
D.23 i2c/p0r.lisp 167
D.24 i2c/p12r-d2=d.lisp 170
D.25 i2c/p12r-d2not=d.lisp 172
D.26 i2c/p12r-p2=p.lisp 174
D.27 i2c/p12r-p2=q.lisp 176
D.28 i2c/p12r-pdp.lisp 177
D.29 i2c/p12r-pdq.lisp 178
D.30 i2c/p12r-pdx.lisp 181
D.31 i2c/p12r-q2=p-2.lisp 184
D.32 i2c/p12r-q2not=q.lisp 188
D.33 i2c/p12r.lisp 190
D.34 i2c/p12w.lisp 191
D.35 i2c/startballot.lisp 193

List of Figures

1.1 A simple increment program [16]. 3
1.2 A more detailed increment program [16]. 5
1.3 Set theoretic functions, excerpted from [21]. 10

2.1 TLA–ACL2 translations. 19
2.2 TLA–ACL2 translations (continued). 20
2.3 ACL2 translation of action N 23

3.1 Highlights in Gafni and Lamport's proof of Lemma I2c. [3] . 38
3.2 Our proof of Lemma I2c. 44

Abstract

We use the ACL2 theorem prover to verify invariants of a distributed algorithm specified in TLA (Temporal Logic of Actions). The algorithm, Disk Synod, achieves consensus among a set of processors communicating through disks. We discuss the translation of TLA specifications into a finite set theory framework in ACL2, as well as the proof of two invariant properties of Disk Synod.

Chapter 1

Algorithms, Languages, and Tools

Civilization advances by extending the number of important operations which we can perform without thinking about them.

Alfred Whitehead,
An Introduction to Mathematics.

1.1 Introduction

Reasoning in TLA consists largely of reasoning about actions—relations between pairs of states. According to Lamport [15],

> What makes verification of TLA practical is that most of the work lies in the action reasoning—which is ordinary math—rather than in temporal reasoning. This is especially important for formal verification, because temporal reasoning is a lot harder to do formally than ordinary mathematical reasoning.

This project does not deal at all with temporal reasoning. By most accounts, 90% of all reasoning in TLA specifications occurs at the action level, where temporal logic has been eliminated [15, 20]. Interestingly, action reasoning seems to be the least discussed aspect in previous TLA verification work [2, 11, 20]. On second look, this is not surprising, since reasoning about ordinary math is a problem all to its own.

While there is no hope of providing satisfactory support—a good degree of automation—for an arbitrary theorem stated in TLA, we benefit from the fact that TLA specifications and theorems follow a similar pattern. At

its heart, this thesis concerns reasoning about simple set theory concepts, cast in a specific format.

We build upon the finite set theory framework for ACL2 developed by Moore[21]. Acquaintance with Moore's work is an advantage, but we provide a basic introduction. This project has augmented ACL2's finite set theory with some new concepts, added both by Moore and by the author.

The object of investigation is Disk Synod [3], an algorithm for achieving consensus among a group of processors. Disk Synod was suggested by Lamport as a nontrivial example to test the feasibility of a verification system [18]. Inspection of other specifications [10, 14] suggests that our observations carry over to them as well.

Our endeavor partitions naturally into two tasks: translating TLA constructs to ACL2, and proving properties of the translation. After a brief introduction to TLA and ACL2, we describe a translation scheme developed. Then we discuss the mechanical verification of two invariant properties of Disk Synod that we successfully proved with ACL2.

As this is a work in progress, we will point out without hesitation important issues that have not yet been addressed.

1.2 The Temporal Logic of Actions

We now present a brisk and selective introduction to TLA. For a more detailed exposition, refer to [16] or [14].

TLA is a first-order temporal logic. Its formulas are built from the usual predicate calculus constructs (function symbols, predicate symbols, variable symbols, $\neg$, $\wedge$, $=$, and $\exists$) as well as the special symbols $\Box$, $'$, and $\exists\!\!\!\exists$. Also assumed is an infinite set of *values*, an infinite set of *variable names*, and the booleans *true* and *false*.

The semantics of TLA is defined in terms of *states*. A state is an assignment of values to variables. The expression $x^2 + y - 3$ has value 6 in a state that assigns 2 to x and 5 to y, and it has value 0 in a state that assigns 1 to x and 2 to y. Variables that may assume different values from state to state are called *flexible variables*. Variables denoting a single value that doesn't change with time are called *rigid variables* (programmers usually call them constants).

Algorithms can be thought of as rules that specify sequences of states. A behavior is an infinite sequence of states; it corresponds to a single "run" of an algorithm. For a sequence of states $\langle S_1, S_2, \ldots \rangle$, and two contiguous states S_i and S_{i+1}, we call S_i the *old state* and S_{i+1} the *new state*.

$$
\begin{aligned}
Init_\Phi &\triangleq (x = 0) \wedge (y = 0) \\
M_1 &\triangleq (x' = x + 1) \wedge (y' = y) \\
M_2 &\triangleq (y' = y + 1) \wedge (x' = x) \\
M &\triangleq M_1 \vee M_2 \\
\Phi &\triangleq Init_\Phi \wedge \Box[M]_{(x,y)} \wedge WF_{(x,y)}(M_1) \wedge WF_{(x,y)}(M_2)
\end{aligned}
$$

Figure 1.1: A simple increment program [16].

An *action* is a boolean-valued expression made up of variables, primed variables, and constant symbols. The expression $y' + 3 = y + z$ is an action, where y and z are variables. The unprimed variable y represents the value of y in the old state, and y' represents the value of y in the new state. For our purposes, it's enough to think of a primed variable v' not as an operator applied to a variable, but as just another variable name, which happens to denote the value of v in the next state.

To illustrate the above concepts, we introduce an example due to Lamport [16]. Figure 1.1 describes a system with two variables x and y, both initially 0 and increasing nondeterministically. The next-state action M describes how the system can change from one state to the next. Formula Φ expresses the requirement that the system always executes an M-action (or stutters), and that both x and y are incremented infinitely often.

Invariant Properties

We can prove properties of our increment example. Useful properties of a system are often *invariant* properties—those which are true of the system at every state of its execution. In temporal logic, invariant properties are of the form $\Box P$, where the $\Box$ operator applied to a property P means that P holds in every state. In TLA, one proves an invariance property $\Box P$ of a system with initial state *Init* and next-state N by establishing the following conditions (ignoring the f subscript). [1]

$$
\begin{aligned}
&Init \Rightarrow I \\
&I \Rightarrow P \\
&I \wedge [N]_f \Rightarrow I'
\end{aligned}
$$

[1] Notice that formula I appears primed. What does it mean for a formula to be primed? In our context, we think of I' as formula I where all variables are primed.

An application of a TLA inference rule, along with simple temporal reasoning, lets us establish $\Box P$ from the above conditions. We should emphasize that this last step of converting action-level properties into temporal properties is not where the work lies. In the Disk Synod algorithm, establishing $\Box P$ takes up one page, while the creation of an invariant and verification of the three conditions spans 18 pages. Creativity is needed to formulate a suitable invariant; the proof effort lies in verifying the invariant.

Going back to our simple example, we may wish to establish type invariance of the system variables:

$$T \triangleq (x \in Nat) \wedge (y \in Nat).$$

To establish T, we must show (again ignoring subscripts):

$$Init_{\Phi} \Rightarrow T \tag{1}$$
$$T \wedge [M]_{(x,y)} \Rightarrow T' \tag{2}$$

The proof of (1) is trivial. The proof of (2) is achieved by considering actions M_1 and M_2 separately. For example, in the case of M_1 we must show

$$(x \in Nat) \wedge (y \in Nat) \wedge (x' = x + 1) \wedge (y' = y) \Rightarrow (x' \in Nat) \wedge (y' \in Nat).$$

Another Example

In TLA, there is no distinction between program and specification. Formula Φ in Figure 1.1 could describe a (very simple) program, or specify the properties that a more elaborate program must satisfy. Figure 1.2 shows a TLA formula Ψ that implements Φ.[2] Formula Ψ consists of two processes executing a loop with three atomic operations. The processes share a semaphore variable *sem*.

We will not go into the details of the proof that Ψ implements Φ, but only remark that the theorem to prove is

$$\Psi \Rightarrow \Phi.$$

[2] To say that Ψ implements Φ means that for any behavior, Ψ satisfies the requirements imposed by Φ—in this case, it means that Ψ always executes an action in which either x or y is incremented, and both x and y are incremented infinitely often.

$$
\begin{array}{lcl}
Init_\Psi & \triangleq & \land\ (pc_1 = \text{“a”}) \land (pc_2 = \text{“a”}) \\
& & \land\ (x = 0) \land (y = 0) \\
& & \land\ sem = 1
\end{array}
$$

$$
\begin{array}{lcl}
\alpha_1 & \triangleq & \land\ (pc_1 = \text{“a”}) \land (0 < sem) \\
& & \land\ pc_1' = \text{“b”} \\
& & \land\ sem' = sem - 1 \\
& & \land\ \text{UNCHANGED}\ \langle x, y, pc_2 \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\alpha_2 & \triangleq & \land\ (pc_2 = \text{“a”}) \land (0 < sem) \\
& & \land\ pc_2' = \text{“b”} \\
& & \land\ sem' = sem - 1 \\
& & \land\ \text{UNCHANGED}\ \langle x, y, pc_1 \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\beta_1 & \triangleq & \land\ (pc_1 = \text{“b”}) \\
& & \land\ pc_1' = \text{“g”} \\
& & \land\ x' = x + 1 \\
& & \land\ \text{UNCHANGED}\ \langle y, sem, pc_2 \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\beta_2 & \triangleq & \land\ (pc_2 = \text{“b”}) \\
& & \land\ pc_2' = \text{“g”} \\
& & \land\ y' = y + 1 \\
& & \land\ \text{UNCHANGED}\ \langle y, sem, pc_1 \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\gamma_1 & \triangleq & \land\ pc_1 = \text{“g”} \\
& & \land\ pc_1' = \text{“a”} \\
& & \land\ sem' = sem + 1 \\
& & \land\ \text{UNCHANGED}\ \langle x, y, pc_2 \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\gamma_2 & \triangleq & \land\ pc_2 = \text{“g”} \\
& & \land\ pc_2' = \text{“a”} \\
& & \land\ sem' = sem + 1 \\
& & \land\ \text{UNCHANGED}\ \langle x, y, pc_1 \rangle
\end{array}
$$

$$
\begin{array}{lcl}
N_1 & \triangleq & \alpha_1 \lor \beta_1 \lor \gamma_1 \\
N_2 & \triangleq & \alpha_2 \lor \beta_2 \lor \gamma_2 \\
N & \triangleq & N_1 \lor N_2 \\
w & \triangleq & (x, y, sem, pc_1, pc_2) \\
\Psi & \triangleq & Init_\Psi \land \Box[N]_w \land SF_w(N_1) \land SF_w(N_2)
\end{array}
$$

Figure 1.2: A more detailed increment program [16].

Since both Ψ and Φ are logical formulas, simulation is expressed as implication. Not surprisingly, an invariant is needed in the proof. In section 2.5 , we discuss the invariant and its proof in ACL2.

TLA$^+$

TLA provides the logic necessary to reason about concurrent systems. TLA$^+$ is a specification language based on TLA. It is formally defined and has a precise grammar, which makes it amenable to mechanized tool support. TLA$^+$ introduces the mathematical concepts necessary for specification, as well as a module system not discussed here. Some concepts (or their particular syntax) may be unfamiliar to the reader, so we introduce them here.

The CHOOSE operator. The expression CHOOSE $x : p$ denotes an unspecified value v such that p holds when v is substituted for x, if such a v exists. Otherwise, the expression denotes an arbitrary value.

Functions. In TLA$^+$, a function is an unspecified set with certain properties. The following constructs are primitives:

- $f[e]$: The function f applied to e.
- DOMAIN f : The domain of f.
- $[S \to T]$: The set of all functions with domain S and range a subset of T.
- $[x \in S \mapsto e]$: An explicit way to define a function: for each x in its domain S, x is mapped to e.

Functions can be modified through the EXCEPT construct. The expression

$$[f \text{ EXCEPT } ![m] = u]$$

denotes the function g that is equal to f, except it maps m to u.

There are two ways to write functions of several arguments. One is to let the domain of a function be an n-tuple, and the other is by writing a curried function. A function f with domain a set of triples $A \times B \times C$ and range R applied to the argument $\langle a, b, c \rangle$, is written $f[\langle a, b, c \rangle]$ or $f[a, b, c]$. It belongs to the set $[A \times B \times C \to R]$. On the other hand, a curried function

belonging to the set $[A \to [B \to [C \to R]]]$, applied to three arguments a, b and c, is written $f[a][b][c]$.

TLA also introduces *operators*, which are different from functions. Functions have a domain (a set), while operators do not. Functions can be defined recursively, but operators cannot. Function application uses square brackets, while operator application uses parentheses. Thus $f[x]$ reveals that f is a function, while $g(x)$ reveals that g is an operator. We do not elaborate further on this distinction; more information can be found in [14].

Records. Records are defined in terms of functions; their notation is similar to functional notation. The expression

$$[h_1 \mapsto e_1, h_2 \mapsto e_2, h_3 \mapsto e_3]$$

denotes a record with (*field*, *value*) pairs (h_1, e_1), (h_2, e_2), and (h_3, e_3). More formally, it is the function that maps the string "h_1" to the value e_1, the string "h_2" to the value e_2, and the string "h_3" to the value e_3. The expression

$$[h_1 : S_1, h_2 : S_2, h_3 : S_3]$$

denotes the set of all records with fields h_1, h_2 and h_3, and corresponding value sets. More formally, it is the set of all functions with domain $\{$"h_1", "h_2", "h_3"$\}$ and range the union $S_1 \cup S_2 \cup S_3$, such that if r is in the set, the application $r[$"h_i"$]$ is an element of S_i.

The field h_1 of a record r is written $r.h_1$, which is equivalent to $r[$"h_1"$]$.

Other constructs. The UNCHANGED construct is used to specify flexible variables that do not change within an action. The expression

$$\text{UNCHANGED } \langle x, y, z \rangle$$

is equivalent to

$$(x' = x) \wedge (y' = y) \wedge (z' = z).$$

The following constructs are formally introduced in TLA$^+$. Their meaning should be familiar to computer scientists.

if p **then** e_1
else e_2

case $p_1 \rightarrow e_1 \Box \ldots \Box p_n \rightarrow e_n$

case $p_1 \rightarrow e_1 \Box \ldots \Box p_n \rightarrow e_n \Box$ **other** $\rightarrow e$

let $d \triangleq f$
in e

Mechanical Support for TLA

Ours is not the first effort to provide mechanical support for TLA. A parser and a model checker [14] have detected bugs in TLA specifications. In the theorem proving arena, Larch [2], Isabelle [20] and HOL [11] have been used to verify TLA theorems. Among these, TLP has been used to verify large specifications. The above research has highlighted the benefit in separating temporal reasoning from action reasoning. While most of the previous work pays great attention to the temporal aspects of TLA, our effort deals exclusively with actions. In doing this, we hope to focus our energies in the area where mechanical verification might be most useful to the TLA user: reasoning about large formulas composed of simple mathematical concepts.

Having discussed some basic TLA concepts, we address Moore's work on finite set theory in ACL2 [21]. We invite the reader to draw connections between the TLA concepts just introduced and the ACL2 concepts introduced in the next section. These connections will be made explicit in section 2.3.

1.3 Finite Set Theory in ACL2

How can sets be represented in ACL2? One answer is to use lists. But writing down the list `'(1 2 3)` to represent the set $\{1, 2, 3\}$ is not enough. Lists have properties that sets do not—they distinguish duplication and order.

To deal with sets in a more natural way, Moore has developed a theory of hereditarily finite sets [21]. In this theory, the universe is divided into *ur-elements* and *sets*. The ur-elements are all ACL2 numbers, strings, characters, symbols and conses of the form `(:UR-CONS (`e_1 `.` e_2`))`. All other conses represent *sets*. The symbol `nil` is both a ur-element and the empty set.

Here are some examples of sets in conventional notation and in ACL2. (We use the notation $[x_1, \ldots x_n]$ to represent a list.)

Conventional notation	**ACL2 notation**
$\{\}$	`nil`
$\{\{\}\}$	`'(nil)`
$\{1\}$	`'(1 1 1)`
$\{1, 2, 3\}$	`'(1 2 3)`
$\{1, 2, 3\}$	`'(1 2 2 3)`
$\{1, \{1\}, 2, \{3, 4\}\}$	`'(1 (1) 2 (3 4))`
$\{1, \{1\}, 2, [3, 4]\}$	`'(1 (1) 2 (:ur-cons (3 4)))`

To reason about equality, we use the relation `=`. This relation encompasses both set equality and ur-elementp equality. It can recognize, for example, `'((1 2) 3)` and `'(3 3 (2 1))` as the same set. For ur-elements, `=` reduces to `equal`. When defining a set-theoretic function, it is important to establish `=`-congruence: it allows ACL2 to exchange two equal sets. If this congruence were not established for the second argument of `mem`, ACL2 wouldn't easily prove the following simple fact.

```
(defthm mem-union-lemma
  (implies (and (mem e s1)
                (= s1 s2))
           (mem e s2)))
```

Figure 1.3 shows many basic set-theoretic functions defined by Moore. We have defined some additional functions and document them here.

- `(dot-dot` m n`)`: $\{i \mid i \in \mathbf{N} \wedge m \leq i \leq n\}$.
- `(union*` a`)`: the union over all elements in a.
- `(all-fns` d r`)`: the set of all functions with domain d and range a subset of r.
- `(defrec` *name* `(`h_1 S_1`) ... (`h_k S_k`))`: defines `(`*name*`)` to be the set of all functions with domain $\{h_1, \ldots h_k\}$ and range $S_1 \cup \cdots \cup S_k$, such that the value of *name* on h_i is a member of S_i.

(`ur-elementp` a): `t` or `nil` according to whether a is an ur-element.

(`setp` a): `t` or `nil` according to whether a is a set.

(`scons` e a): $\{e\} \cup a$.

(`brace` α_1 ... α_k): the set whose elements are given by the values of the k expressions; this is known as "roster notation."

(`=` a b): If a and b are the same object, then `t`, otherwise, `nil`.

(`mem` e a): $e \in a$. Both arguments are treated as sets.

(`subsetp` a b): $a \subseteq b$.

(`nats` n): $\{i \mid i \in \mathbf{N} \wedge 0 \leq i \leq n\}$.

(`union` a b): $a \cup b$.

(`intersection` a b): $a \cap b$.

(`diff` a b): $a \setminus b$.

(`choose` a): an element of the set a, if a is non-empty.

(`functionp` f): if f is a set of pairs and no two elements of f have the same `hd`, then `t`; otherwise, `nil`. If a function f contains (`pair` e v), then we say v is the *value* of f on e.

(`domain` f): $\{e \mid \exists x(x \in f \wedge e =$(`hd` x) $)\}$.

(`range` f): $\{e \mid \exists x(x \in f \wedge e =$(`tl` x) $)\}$.

(`apply` f e): if f is a function and e is in its domain, then the value of f on e.

(`except` f e v): If f is a function then the function that is everywhere equal to f except on e where the value is v.

(`func` (e_1 v_1) ... (e_n v_n)): the function mapping e_i to v_i. Presentation order is used to resolve conflicts. That is, if e_i is e_j for some $i < j$, then the function maps e_i to v_i.

(`powerset` a): $\{s \mid s \subseteq a\}$

(`defmap` f (v_1 ... v_k) `:for` x `:in` v_i `:such-that` ϕ):
defines (f v_1 ... v_k) to be $\{x \mid x \in v_i \wedge \phi\}$.

(`defmap` f (v_1 ... v_k) `:for` x `:in` v_i `:map` ϕ):
defines (f v_1 ... v_k) to be $\{e \mid \exists x(x \in v_i \wedge e = \phi)\}$.

Figure 1.3: Set theoretic functions, excerpted from [21].

- `(defmap-fn` f `(`v_1 ... v_k`) :for` x `:in` v_i `:map` ϕ`)`:
  defines `(`f v_1 ... v_k`)` to be $\{\langle x, e\rangle \mid \exists x(x \in v_i \wedge e = \phi)\}$.

- `(defexists` f `(`v_1 ... v_k`) :exists` x `:in` v_i `:such-that` ϕ`)`:
  defines `(`f v_1 ... v_k`)` to be `t` if $\exists x \in v_i : \phi$, and `nil` otherwise.

- `(defall` f `(`v_1 ... v_k`) :forall` x `:in` v_i `:holds` ϕ`)`:
  defines `(`f v_1 ... v_k`)` to be `t` if $\forall x \in v_i : \phi$, and `nil` otherwise.

The ACL2 definitions of `all-fns` and `defrec`, as well as proofs of their characteristic properties, are due to Moore. Appendices D.1, D.5, and D.4 contain the ACL2 files defining the above constructs.

We have only scratched the surface of Moore's work, but it is enough to get us started. For a detailed introduction to finite set theory in ACL2, refer to [21].

Chapter 2

Translation

Several issues arise when we consider the differences between the TLA and ACL2 logics, and the static nature of TLA as a specification language versus the dynamic nature of ACL2 as a theorem prover. We list some of these issues here, limiting our discussion to nontemporal aspects of TLA.

- TLA makes heavy use of existential and universal quantification. ACL2 formally has quantification (in the form of skolem functions), but it isn't well supported; for that reason we do not use it.

- ACL2 overspecifies TLA expressions. This can lead to conclusions in ACL2 that are not necessarily true in TLA. For example, Lamport and Gafni note [3]:

  > We deduce $phase'[p] = 2$ from $phase' = [phase \text{ EXCEPT } ![p] = 2]$ only if $phase$ is a function whose domain contains p.

  However, in ACL2, given

  `(= phase-prime (except phase p 2))`,

  we can deduce `(= (apply phase-prime p) 2)))` with no further hypotheses about `phase`. A more fundamental example is that 4 = "`h`" might be a theorem in TLA. Everything is a set in TLA, so 4 and "`h`" are sets. Since sets like 4 and "`h`" are left unspecified, the set denoted by 4 and the set denoted by "`h`" may in fact be the same set. But in ACL2, `(= 4 "h")` immediately evaluates to `nil`.

  This is a very important issue because it deals with the relationship between a theorem in TLA and its corresponding version in ACL2. It is the next issue we plan to tackle in our project.

- In ACL2, theorems become rules that guide the mechanical theorem prover in finding proofs of further theorems. The user is keenly aware of this, and is sensitive, when writing down a theorem, to the way it will be used by the system. The form of a theorem can make or break a proof, even if two different forms are logically equivalent. TLA formulas, on the other hand, are not written to become effective rules. A TLA user might state an assumption in an elegant form that would not be a useful ACL2 rule.

- In our ACL2 framework, all sets are finite. Our current solution when dealing with an infinite set in TLA is to reword the TLA expression surrounding the set. For example, the expression $x \in \mathit{Nats}$, where *Nats* is the set of natural numbers, can be translated as

  ```
  (and (integerp x) (>= x 0)).
  ```

  All the infinite sets we have encountered so far are amenable to this type of transformation.

- ACL2 is applicative: there is no notion of global variables. In TLA, state variables are global. In substance, this is a minor point—we can simply pass all state variables as parameters among ACL2 functions. But soon our ACL2 translations become hideous to look at, and it's easy to lose sight of a function body in a sea of parameters.

We will address the above issues throughout our discussion.

2.1 Translation Guidelines

In creating a translation scheme, our overriding principles are clarity and simplicity. We need a scheme that makes it obvious how an ACL2 term corresponds to a TLA expression. Otherwise, we might say things in ACL2 that weren't intended in TLA.

A simple translation scheme has another advantage: mechanization. While the translation of Disk Synod was done by hand, our experiments highly suggest that it can be automated. This will become a necessity for larger specifications. We hope the informal rules that we lay out suggest how our translation scheme can be mechanized.

We should mention that TLA constructs and their ACL2 translation differ in how they are formally defined. For instance, the TLA conditional constructs are defined in terms of the CHOOSE operator [14].

$$\textbf{if } p \textbf{ then } e_1 \textbf{ else } e_2 \triangleq \text{CHOOSE } v : (p \Rightarrow v = e_1) \wedge (\neg p \Rightarrow v = e_2)$$

$$\textbf{case } p_1 \rightarrow e_1 \Box \ldots \Box p_n \rightarrow e_n \triangleq \text{CHOOSE } v : (p1 \wedge (v = e1)) \vee \ldots \vee (pn \wedge (v = en))$$

It is obvious that if we want to use ACL2 in any sensible way, we should use its built-in conditionals `IF` and `COND`, which have nothing to do with the function `choose`. This deviation makes its difficult to establish a formal correspondence between TLA and ACL2. But in our opinion, it is vastly more important to use ACL2 to prove facts about TLA specifications than it is to construct a formal correspondence between the two logics.

Before laying out the translation scheme, we discuss three adopted conventions.

2.2 Translation Conventions

The Naming Convention

Consider the following TLA construct:

$$\mathit{collect} \triangleq \{x + 2 : x \in \{3y : y \in S\}\}.$$

How can we translate *collect* into ACL2? Trying to cast it in a more succinct or clever form is not allowed—the role of a translator is to translate, not to illuminate. So let's begin with the innermost set $\{3y : y \in S\}$, and assume that S is a finite set that translates as the ACL2 constant `(S)`. Using the set comprehension macro `defmap`, we obtain

```
(defmap collect-1 (dom) :for y :in dom :map (* 3 y)).
```

The set $\{3y : y \in S\}$ can now be expressed as `(collect-1 (S))`. The outermost set is translated similarly:

```
(defmap collect-2 (dom) :for x :in dom :map (+ x 2)).
```

Finally, we define a function `collect` that corresponds to *collect*.

```
(defun collect ()
  (collect-2 (collect-1 (S))))
```

The above translation does not require imagination; it is direct and therefore attractive. But it does define three functions, and three names to go with them. In larger specifications, name generation can become a problem. To drive our point home, let's consider another TLA construct:

$$P \triangleq \forall\, x, y, z \in S : p(x, y, z).$$

The first thing to note is that P is shorthand for

$$\forall\, x \in S : \forall\, y \in S : \forall\, z \in S : p(x, y, z).$$

Again, we assume that S translates into ACL2 as `(S)`. We also assume that $p(x, y, z)$ translates as `(p x y z)`. Using the `defall` macro (Section 1.3), we obtain the following translation.

```
(defall forall-p-z (dom x y)
 :forall z :in dom :holds (p x y z))

(defall forall-p-y (dom x)
 :forall y :in dom :holds (forall-p-z dom x y))

(defall forall-p-x (dom)
 :forall x :in dom :holds (forall-p-y dom x))
```

We express P as `(forall-p-x (S))`.

Again, the translation is straightforward. And again, there are more function names than we care for. There is no clear way to avoid generating multiple functions for unnamed TLA expressions in our current framework. The best we can do is adopt a naming convention, so that reasonable names are generated mechanically. What we deem a reasonable name is not yet clear, so we postulate an undefined *naming convention*, both expressing the need for a uniform naming strategy, and our current lack of commitment to a particular one.

Flexible Variable Conventions

We now present two conventions dealing with the translation of flexible variables. The flexible variables in a TLA$^+$ specification are declared using the VARIABLES construct. For instance,

VARIABLES x, y, z

declares x, y and z as flexible variables in the given specification.

Convention. *A TLA variable name is translated without change. A primed TLA variable is translated by appending "*`-n`*" to its unprimed variable name.*

For example, the TLA variable x translates to `x`, and x' translates to `x-n`.

The next convention hides flexible variables in ACL2 translations. This makes ACL2 translations look more like the TLA formulas that produced them, and prevents a large number of flexible variables from taking up space in ACL2 definitions. [1]

Convention. *Flexible variables are always hidden by macros.*

Consider a specification with five flexible variables $x_1, \ldots, x_5$, and the action

$$A_1 \triangleq \land\ x_2' = x_2 + 1 \\ \land\ \text{UNCHANGED}\ \langle x_1, x_3, x_4, x_5 \rangle.$$

To translate this action, we need to define a function that takes 10 arguments—one for each unprimed variable, and one for each primed variable. The `defaction` macro lets us hide flexible variables in our definitions. Action A_1 translates as follows:

```
(defaction a1 ()
  (and (= x2-n (+ x2 1))
       (unchanged x1 x3 x4 x5))).
```

[1] The reader might wonder how much space flexible variables as function arguments can take. The answer is: enough to distract us from the functions at hand.

This form expands to the following two events.

```
(defun _a1 (x1 x1-n x2 x2-n x3 x3-n x4 x4-n x5 x5-n)
  (and (= x2-n (+ x2 1))
       (= x1-n x1) (= x3-n x3) (= x4-n x4) (= x5-n x5)))

(defmacro a1 () '(_a1 x1 x1-n x2 x2-n x3 x3-n x4 x4-n x5 x5-n))
```

Afterwards, if we need to use A_1 in a theorem, we can just type `(a1)`. The `defstate` macro is similar to `defaction` except it is used to define state predicates, which do not include primed variables. It is useful when defining invariants. For example, the invariant from formula Φ in Section 1.2,

$$T \triangleq (x \in Nat) \wedge (y \in Nat),$$

is translated using `defstate` as follows (`t` means *true* in ACL2, so we name T differently).

```
(defstate t-inv () (and (integerp x) (>= x 0)
                        (integerp y) (>= y 0)))
```

`Defstate` creates two forms, one to express T and one to express T'.

```
(defun _t-inv (x1 x2) (and (integerp x) (>= x 0)
                           (integerp y) (>= y 0))
(defmacro t-inv   () '(_t-inv x   y))
(defmacro t-inv-n () '(_t-inv x-n y-n))
```

2.3 Translation Rules

With our previous discussion in mind, we use the building blocks discussed in Section 1.3 to translate TLA constructs into their ACL2 counterparts. Figures 2.1 and 2.2 show the translation scheme. It is understood that all sets in question are finite; otherwise the translation does not apply. Our translation scheme is not formal, but we believe it can be made formal enough to be mechanized. The breakdown of categories is from Lamport's *A Summary of TLA*$^+$[12]. The reader can refer to Lamport's summary and find the concepts we left out.

We do not provide entries for some obvious translations, such as $\in$, $\wedge$, $\vee$, $\subseteq$, etc. We not only show what can be translated, but also what cannot. The constructs we cannot directly translate deal with expressions quantified over infinite sets.

Abbreviations

Some TLA expressions have elaborate ACL2 translations that can be difficult to understand. Consider the following example.

$$[f \text{ EXCEPT } ![x][y] = z \\ ![a][b] = c]$$

It denotes the function g equal to f except that $g[x][y]$ equals z and $g[a][b]$ equals c. Its ACL2 translation is straightforward but cryptic:

```
(except (except f a (except (apply f a) b c))
        x
        (except (apply (except f a (except (apply f a) b c))
                       x)
                y z)).
```

Using macros, we can create shorthand notation for complicated expressions. `Except-and` is an example. The form

```
(except-and f (x y z) (a b c))
```

expands to the same expression as the "cryptic" one. Here are other examples of TLA expressions, macros that provide a succinct translation, and their expansion.

- $f[x][y]$ translates as the form
  `(apply-m f x y)`, which expands to
  `(apply (apply f x) y)`.

- $[f$ EXCEPT $![x][y] = z]$ translates as the form
  `(except-m f x y z)`, which expands to
  `(except f a (except (apply f a) b c))`.

Logic

BOOLEAN	`(brace t nil)`
$\forall x : p$	*no translation*
$\exists x : p$	*no translation*
$\forall x \in S : p$	$(f\ S\ v_1\ \ldots\ v_k)$, where f adheres to the naming convention, and is defined by `(defall` f `(`$dom\ v_1\ \ldots\ v_k$`) :forall` x `:in` dom `:holds` p`).`
$\exists x \in S : p$	$(f\ S\ v_1\ \ldots\ v_k)$, where f adheres to the naming convention, and is defined by `(defexists` f `(`$dom\ v_1\ \ldots\ v_k$`) :exists` x `:in` dom `:such-that` p`).`
CHOOSE $x : p$	*no translation*
CHOOSE $x \in S : p$	`(choose (`$f\ S\ v_1\ \ldots\ v_k$`))`, where f adheres to the naming convention, and is defined by `(defmap` f `(`$dom\ v_1\ \ldots\ v_k$`) :for` x `:in` dom `:such-that` p`).`

Sets

SUBSET S	`(powerset` S`)`
UNION S	`(union*` S`)`

Figure 2.1: TLA–ACL2 translations.

Functions

$f[e]$	`(apply` f e`)`
DOMAIN f	`(domain` f`)`
$[x \in S \mapsto e]$	`(`f S v_1 ... v_k`)`, where f adheres to the naming convention, and is defined by `(defmap` f `(`dom v_1 ... v_k`) :for` x `:in` dom `:map` e`).`
$[S \to T]$	`(all-fns` S T`)`
$[f$ EXCEPT $![e_1] = e_2]$	`(except` f e_1 e_2`)`

Records

$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$	`(func (`h_1 e_1`) ... (`h_n e_n`))`
$[h_1 : S_1, \ldots, h_n : S_n]$	`(`$name$`)`, where $name$ adheres to the naming convention and is defined by `(defrec` $name$ `(`h_1 S_1`) ... (`h_n S_n`)).`
$[r$ EXCEPT $![h] = e]$	`(except` r h e`)`

Figure 2.2: TLA–ACL2 translations (continued).

- UNCHANGED $\langle x, y, z \rangle$ translates as the form
  `(unchanged x y z)`, which expands to
  `(and (= x-n x) (= y-n y) (= z-n z))`.

These abbreviations may be considered just an aesthetic issue. But they are important because they make expressions easier to understand, and therefore easier to reason about.

2.4 Requirements for the TLA specifier

A TLA specifier using our translation scheme is obliged to satisfy the following requirements.

Witnesses

When a constant or function is not explicitly declared but assumed to have certain properties, we require a witness—an object that has the same properties as the declared constant or function. Suppose we declare the constant N to be a positive integer.

CONSTANT N
ASSUME $(N \in Nat) \wedge (N > 0)$

The above statements are translated with an `encapsulate` event [7]. This event lets us create an undefined function enjoying the properties exported from the encapsulation. We are required to define locally a function that has the properties claimed. Here is the encapsulated event for N.

```
(encapsulate ((n () t))
             (local (defun n () 1))
             (defthm n-constraint
               (and (integerp (n))
                    (< 0 (n)))
               :rule-classes :type-prescription))
```

Exhibiting a witness is a good practice regardless of ACL2 verification: it keeps us from writing unsound definitions.

In future work, we hope to build a TLA–ACL2 translator that allows the TLA user to provide witnesses as part of a specification, flagged in such a way that the translator can recognize them and use them appropriately.

Infinite sets

As mentioned earlier, we cannot express infinite sets. Our solution is to recast expressions involving them. A TLA user planning to use our framework to verify a specification should be sensitive to this issue and avoid infinite sets. Given that TLA is used to reason about computational systems, we do not expect to see many infinite sets in a specification. They might be more widely used, however, when expressing properties of the specification.

2.5 Translating the Increment Example

We now revisit the example introduced in Section 1.2. Recall formula Ψ from Figure 1.2, denoting two processes that increment variables x and y and share a semaphore *sem*. Figure 2.3 shows the ACL2 translation of action N in Ψ.

In the process of proving that Ψ implements Φ (Figure 1.1), the following invariant is defined [16].

$$
\begin{array}{lll}
I & \triangleq & \land\ x \in Nat \\
 & & \land\ \lor\ (sem = 1) \land (pc_1 = pc_2 = \text{“a”}) \\
 & & \quad \lor\ (sem = 0) \quad \land \lor\ (pc_1 = \text{“a”}) \land (pc_2 \in \{\text{“b”}, \text{“g”}\}) \\
 & & \qquad\qquad\qquad\ \lor\ (pc_2 = \text{“a”}) \land (pc_1 \in \{\text{“b”}, \text{“g”}\})
\end{array}
$$

Here is its ACL2 translation.

```
(defstate i ()
  (and (and (integerp x) (>= x 0))
       (or (and (= sem 1) (= pc1 "a") (= pc2 "a"))
           (and (= sem 0)
                (or (and (= pc1 "a") (mem pc2 (brace "b" "g")))
                    (and (= pc2 "a") (mem pc1 (brace "b" "g")))))))))
```

A theorem to prove is I's invariance across states.

$$I \land N \Rightarrow I'$$

```
(defthm i-invariant
 (implies (and (i) (n))
          (i-n)))
```

```
(defstate init
  (and (= pc1 "a") (= pc2 "a")
       (= x 0) (= y 0)
       (= sem 1)))

(defaction alpha1
  (and (= pc1 "a") (< 0 sem)
       (= pc1-n "b")
       (= sem-n (- sem 1))
       (unchanged x y pc2)))

(defaction alpha2
  (and (= pc2 "a") (< 0 sem)
       (= pc2-n "b")
       (= sem-n (- sem 1))
       (unchanged x y pc1)))

(defaction beta1
  (and (= pc1 "b")
       (= pc1-n "g")
       (= x-n (+ x 1))
       (unchanged y sem pc2)))

(defaction beta2
  (and (= pc2 "b")
       (= pc2-n "g")
       (= y-n (+ y 1))
       (unchanged x sem pc1)))

(defaction gamma1
  (and (= pc1 "g")
       (= pc1-n "a")
       (= sem-n (+ sem 1))
       (unchanged x y pc2)))

(defaction gamma2
  (and (= pc2 "g")
       (= pc2-n "a")
       (= sem-n (+ sem 1))
       (unchanged x y pc1)))

(defaction n1
  (or (alpha1) (beta1) (gamma1)))

(defaction n2
  (or (alpha2) (beta2) (gamma2)))

(defaction n
  (or (n1) (n2)))
```

Figure 2.3: ACL2 translation of action N

ACL2 proves `i-invariant` automatically. The file containing ACL2 events for the increment example is in Appendix A.

2.6 Disk Synod

Disk Paxos [3] is an algorithm for implementing an arbitrary distributed system with a network of processors and disks. Disk Paxos is fault tolerant—it maintains consistency in the event of lost or delayed messages and certain types of processor failure. It also ensures progress as long as one processor can read and write a majority of the disks, and all other processors are either non-faulty or have failed completely.

Implementing an arbitrary distributed system reduces to solving the consensus problem. A system is represented as a deterministic state machine executing a series of commands [17, 22]. The state machine represents a group of processors, and all processors must agree on each command. To reach agreement, the consensus problem must be solved. Here is a description of the problem from Gafni and Lamport [3].

> In the consensus problem, each processor p starts with an input value $input[p]$, and all processors output the same value, which equals $input[p]$ for some p. A solution should be:
>
> - **Consistent** All value output are the same.
> - **Nonblocking** If the system is stable and a non-faulty processor can communicate with a majority of disks, then the processor will eventually output a value.

Disk Synod is the consensus algorithm used by Disk Paxos. Our work deals with the translation of Disk Synod and the proof of its consistency. We will not explain Disk Synod; for a discussion of the algorithm, refer to [3]. All we need to know is that Disk Synod concerns a group of processors *Proc* and a group of disks *Disk*, communicating with each other by writing to and reading values from the disks in *Disk*.

The next-state action of Disk Synod consists of seven actions.

$$
\begin{array}{lll}
\mathit{Next} & \triangleq & \exists p \in \mathit{Proc} : \\
& & \quad \lor \mathit{StartBallot}(p) \\
& & \quad \lor \exists d \in \mathit{Disk} : \lor \mathit{Phase0Read}(p, d) \\
& & \qquad\qquad\qquad \lor \mathit{Phase1or2Write}(p, d) \\
& & \qquad\qquad\qquad \lor \exists q \in \mathit{Proc} \setminus \{p\} : \mathit{Phase1or2Read}(p, d, q) \\
& & \quad \lor \mathit{EndPhase1or2}(p) \\
& & \quad \lor \mathit{Fail}(p) \\
& & \quad \lor \mathit{EndPhase0}(p)
\end{array}
$$

The algorithm has three phases, in which processors read values from a set of disks, try to submit a chosen value by writing to disks, or declare their output value as the chosen one. These activities are suggested by the names of the various actions. What it means for a processor to fail is specified in action $\mathit{Fail}(p)$. Appendix B contains the complete specification of Disk Synod.

Let's look at an action and understand its general shape. Action $\mathit{Phase1or2Write}(p, d)$ is defined as follows.

$$
\begin{array}{l}
\mathit{Phase1or2Write}(p, d) \triangleq \\
\quad \land\ \mathit{phase}[p] \in \{1, 2\} \\
\quad \land\ \mathit{disk}' = [\mathit{disk} \text{ EXCEPT } ![d][p] = \mathit{dblock}[p]] \\
\quad \land\ \mathit{disksWritten}' = [\mathit{disksWritten} \text{ EXCEPT } ![p] = @ \cup \{d\}] \\
\quad \land\ \text{UNCHANGED } \langle \mathit{input}, \mathit{output}, \mathit{phase}, \mathit{dblock}, \mathit{blocksRead} \rangle
\end{array}
$$

The first conjunct,

$$\mathit{phase}[p] \in \{1, 2\}$$

acts as a guard. If processor p is not currently in phase 1 or 2, this conjunct evaluates to false and therefore action $\mathit{Phase1or2Write}(p, d)$ evaluates to false, meaning that p cannot execute this action in the current state (an action being "executed" from one state to the next is equivalent to the action being true across these states).

The second and third conjuncts,

$$
\begin{array}{l}
\mathit{disk}' = [\mathit{disk} \text{ EXCEPT } ![d][p] = \mathit{dblock}[p]] \\
\mathit{disksWritten}' = [\mathit{disksWritten} \text{ EXCEPT } ![p] = @ \cup \{d\}]
\end{array}
$$

describe what variables are changed when p executes $\mathit{Phase1or2Write}(p)$,

and how they change. [2] Finally, the fourth conjunct

UNCHANGED $\langle input, output, phase, dblock, blocksRead \rangle$

describes the state variables that remain unchanged when p executes $Phase1or2Write(p)$.

Throughout the rest of this paper, we will refer not to action *Next* but instead to action *HNext* (see Appendix B). The latter action conjoins *Next* with two actions describing the values of variables used to prove the consistency of Disk Synod. Nothing is lost in our context if we assume that *HNext* looks just like *Next* above.

The complete translation of Disk Synod can be found in Appendices C and D.14. Appendix C consists of each TLA construct followed by its ACL2 translation. It omits ACL2 events that are included in the actual translation file, such as congruence theorems and other lemmas needed to reason about the translation. Its purpose is to show the reader TLA and ACL2 expressions side by side. Appendix D.14 contains the ACL2 file that includes Disk Synod's translations, with all the events omitted in Appendix C.

The complete translation of Disk Synod consists of 61 events. `Defmap` declarations and regular function definitions comprise 16 events. We use 9 `defaction` forms, corresponding to the seven subactions of *Next* plus the definitions of *Next* and *HNext*. In addition, we declare 14 quantification events (`defall` and `defexists`). It takes ACL2 183 seconds to process the translation.[3] A large fraction of the time is spent admitting quantification and set comprehension events. We are currently working on a modification to these types of events that will make them both more robust and faster to process.

There are two notable places where we do not follow the translation scheme. The first is in translating TLA's assumed constants. These are the first constructs appearing in Disk Synod, and also the first constructs we translated in the project. At the time, we translated without a rigorous translation scheme. This is reflected in the translation—rather than faithfully translating TLA constructs, we translated them directly into useful ACL2 rules, both saving time and obscuring a mechanical translation. We plan to go back and rework the translations under our now-stable framework.

The second place where we deviate from the translation scheme is in translating existentially quantified expressions. We do this because ACL2 has trouble reasoning with our translation on this front. An in-depth discussion of the problem is found in Section 3.5.

[2] The notation $[f \text{ EXCEPT } ![x] = @ \cup S]$ is equivalent to $[f \text{ EXCEPT } ![x] = f[x] \cup S]$.

[3] We used a 450 MHz Pentium III processor.

Chapter 3

Verification

3.1 Consistency of Disk Synod

In the long version of their paper, Gafni and Lamport [3] establish consistency of Disk Synod by incrementally establishing an invariant. In other words, they establish $I = I_1 \wedge \cdots \wedge I_i$ for increasing i, until the invariant I is strong enough to yield consistency of the algorithm. We do not discuss how the invariant is used to establish consistency.

The final invariant in the Disk Synod proof consists of six smaller invariants.

$$HInv \triangleq HInv1 \wedge HInv2 \wedge HInv3 \wedge HInv4 \wedge HInv5 \wedge HInv6$$

Establishing *HInv* amounts to proving the following theorems, where *HInit* is Disk Synod's initial state and *Hnext* its next-state action.

THEOREM $I1$ $\quad HInit \Rightarrow HInv$
THEOREM $I2$ $\quad HInv \wedge HNext \Rightarrow HInv'$

In this project, we tackle the more challenging THEOREM $I2$. We have used ACL2 to prove versions of $I2$ for $Hinv1$ and $HInv3$:

LEMMA $I2a$ $\quad HInv1 \wedge HNext \Rightarrow HInv1'$.
LEMMA $I2c$ $\quad HInv1 \wedge HInv2 \wedge HInv3 \wedge HNext \Rightarrow HInv3'$.

Our choice of invariants is not arbitrary. Invariant $HInv1$ is interesting to us because it is boring to the TLA specifier—it states type invariance of the specification's variables. It's an invariant that we expect a good verification system to establish automatically. Invariant $HInv3$ is interesting

to us because it is the first invariant for which Gafni and Lamport provide a proof. Can ACL2 follow their proof?

Our proof effort yielded several positive results. In addition to finding syntactic mistakes in the written proof outlines given by Gafni and Lamport, we found a nontrivial error in the course of verifying $I2c$ with ACL2—the original formulation of LEMMA $I2c$ omitted *HInv*2 as an assumption.
The translated lemmas are:

```
(defthm i2a
  (implies (and (hinv1)
                (hnext p d q b-witness1 ip-witness1 b-witness))
           (hinv1-n))).


(defthm i2c
  (implies (and (hinv1)
                (hinv2)
                (hinv3)
                (hnext p d q b-witness1 ip-witness1 b-witness))
           (hinv3-n))).
```

In Section 3.5, we explain the six arguments in action *HNext*.

3.2 Controlling ACL2

It is important to control the amount of information in ACL2's database. Having too many rules can, at best, substantially slow down the prover, and at worst, lead ACL2 astray and result in failure. For this reason, we limit the assumptions visible to ACL2 during a particular proof step. Our general approach is to keep most definitions and theorems disabled, and enable them during particular proofs based on their need.

In the proof of theorem $I2c$, for instance, we keep all three assumed invariants disabled, and enable a particular invariant only when its information is necessary in a proof step. In fact, enabling an invariant rarely provides useful information (except for *HInv*1), because the form in which the invariant is stated does not yield good rewrite rules. In the case of *Hinv*2, we create a separate file `HInv2-exports` that massages *HInv*2 into rules useful to ACL2. These rules are disabled by default. A rule is enabled only inside the theorem that needs it. The reader may wonder how much we cripple ACL2 by disabling all the invariants. This approach does

not cripple ACL2 at all. The cases when invariant information is necessary are interesting cases that would have failed anyway without user guidance. Providing information explicitly in a theorem is also a way to document a nontrivial proof step, making explicit the facts it follows from.

3.3 Type Invariance

Among the first theorems one encounters in a typical TLA specification is a theorem stating type invariance. Such is the case for Disk Synod. Formula $HInv1$ states the type of all flexible variables.

$$
\begin{array}{l}
HInv1 \triangleq \\
\quad \land\ input \in [Proc \to Inputs] \\
\quad \land\ output \in [Proc \to Inputs \cup NotAnInput] \\
\quad \land\ disk \in [Disk \to [Proc \to DiskBlock]] \\
\quad \land\ phase \in [Proc \to 0..3] \\
\quad \land\ dblock \in [Proc \to DiskBlock] \\
\quad \land\ disksWritten \in [Proc \to \text{SUBSET } Disk] \\
\quad \land\ blocksRead \in [Proc \to [Disk \to \text{SUBSET } [block : DiskBlock, proc : Proc]]] \\
\quad \land\ allInput \in \text{SUBSET } Inputs \\
\quad \land\ chosen \in Inputs \cup NotAnInput
\end{array}
$$

$$\text{LEMMA } I2a \qquad HInv1 \land HNext \Rightarrow HInv1'.$$

The translation of $HInv1$ is straightforward (see Appendix D.7). We have proved LEMMA $I2a$ with ACL2. A total of 32 lemmas lead up to its proof. Of these, 19 are general set theoretic lemmas that we have incorporated into our set theory framework. The remaining 13 are lemmas specific to Disk Synod, all of which can be generated automatically following an approach we present below. The proof of $I2a$, including all 32 auxiliary lemmas, takes 47 seconds on a 450MHz Pentium III processor.

ACL2 splits theorems into cases when appropriate. For example, a goal of the form `(IF` e_1 e_2 e_3`)` often generates a case split on e_1, resulting in two subgoals: one to establish e_2 assuming e_1, and one to establish e_3 assuming `(not` e_1`)`. The next-state action $HNext$ is a large expression. It contains several places where ACL2 generates a case split, and each case split doubles the number of subgoals to establish. This leads to a severe case explosion in the proof of $I2a$. The solution is to prove $I2a$ one action at a time. In other words, we prove

LEMMA $I2a_i$ $\quad HInv1 \wedge A_i \Rightarrow HInv1'$

replacing A_i by each of the seven actions in *HNext*. Lemma $I2a$ follows directly from these smaller lemmas.

ACL2 further divides the task of proving each $I2a_i$ into proving each conjunct of $HInv1$ separately. Here is a subgoal involving the variable *disk*.

$$HInv1 \wedge A_i \Rightarrow disk' \in [Disk \rightarrow [Proc \rightarrow DiskBlock]] \qquad (1)$$

Notice that most of the flexible variables in Disk Synod—including *disk*—denote functions. These variables are generally modified using the EXCEPT construct. Suppose that action A_i changes *disk* as follows.

$$disk' = [disk \text{ EXCEPT } ![d][p] = db]$$

Recall that $HInv1$ is among our hypotheses, so we can assume that $disk \in [Disk \rightarrow [Proc \rightarrow DiskBlock]]$. The variable $disk'$ is everywhere equal to *disk*, except that $disk'[d][p]$ equals *db*. If we can show that $disk'$ respects the type $[Disk \rightarrow [Proc \rightarrow DiskBlock]]$ in the only place where it differs from *disk*, we can establish (1). To do this, we must verify three facts:

$$(d \in Disk) \wedge (p \in Proc) \wedge (db \in DiskBlock).$$

The proof of $Hinv1$ is basically a large-scale version of the above example, comprising about 200 cases across all actions of *HNext*. A key lemma is the one characterizing the set $[D \rightarrow R]$ of functions with domain D and range a subset of R. In ACL2, we write this set as `(all-fns D R)`.

```
(defthm all-fns-property
  (iff (mem g (all-fns a b))
       (and (functionp g)
            (= (domain g) (sfix a))
            (subsetp (range g) b)))).
```

Establishing type invariance of a variable $v \in$ `(all-fns D R)` changed through `(except v x y)` is equivalent to establishing the following three subgoals.

- `(functionp (except` v x y`))`
- `(= (domain (except` v x y`)) ` D`)`
- `(subsetp (range (except` v x y`)) ` R`)`

The next three lemmas let us reason about type properties of functions changed through `except`.

```
(defthm functionp-except
  (implies (functionp f)
           (functionp (except f x v)))
  :hints...)

(defthm domain-=-except
  (implies (and (mem x s1)
                (functionp g)
                (= (domain g) s1))
           (= (domain (except g x y)) s1)))

(defthm range-subsetp-except
  (implies (and (mem y s2)
                (subsetp (range f) s2))
           (subsetp (range (except f x y)) s2))
  :hints...)
```

Generating lemmas automatically

We now have the lemmas needed to handle type invariance of EXCEPT expressions. But we aren't finished proving all the lemmas that will establish $HInv1$. The missing lemmas are tedious but shallow; we claim that most of them can be generated automatically.

Consider the operator $allBlocksRead(p)$ in Disk Synod.

$$
\begin{array}{l}
allBlocksRead(p) \triangleq \\
\quad \textbf{let} \;\; allRdBlks \triangleq \text{UNION } \{blocksRead[p][d] \;:\; d \in Disk\} \\
\quad \textbf{in} \;\; \{br.block \;:\; br \in allRdBlks\}
\end{array}
$$

What is the type of $allBlocksRead(p)$? If we assume $Hinv1$ and $p \in Proc$, can we recognize the set to which $allBlocksRead(p)$ belongs?

First let's consider the set that comes after the UNION operator in $allBlocksRead(p)$. Call it α.

$$\alpha \triangleq \{blocksRead[p][d] : d \in Disk\}$$

From $HInv1$, we know the type of the variable *blocksread*:

$blocksRead \in [Proc \rightarrow [Disk \rightarrow \text{SUBSET } BlockProc]]$,
where $BlockProc$ is the set of records $[block : DiskBlock, proc : Proc]$.

Given the type of $blocksRead$, from $p \in Proc$ and $d \in Disk$ it follows that $blocksread[p][d]$ belongs to the set SUBSET $BlockProc$.[1] Thus, the elements of α are members of SUBSET $BlockProc$, which means α is a collection of sets of $BlockProc$ records.

Having determined the type of α, we determine the type of the expression surrounding it.

$$allRdBlks \triangleq \text{UNION } \{blocksRead[p][d] : d \in Disk\}\}$$

Applying UNION to a collection of sets of records—$allRdBlks$ denotes a set of $BlockProc$ records. Finally, we must determine the type of the set returned by $allBlocksRead(p)$. Call it β.

$$\beta \triangleq \{br.block : br \in allRdBlks\}$$

But this last step is easy: collecting all the *block* fields in a set of $BlockProc$ records produces a set of $DiskBlock$ objects (by the definition of $BlockProc$). We conclude that $allBlocksRead(p)$ is a set of $DiskBlock$ objects.

Can we mechanize this train of thought? Let's first look at the translation of $allBlocksRead(p)$ into ACL2.

```
(defmap allblocksread-map-2 (allrdblks)
   :for br :in allrdblks :map (apply br "block"))

(defmap allblocksread-map-3 (disk p blocksread)
  :for d :in disk :map (apply-m blocksread p d))
```

[1] Remember that SUBSET S is the powerset of S.

```
(defun allblocksread (p blocksread)
  (allblocksread-map-2
      (union* (allblocksread-map-3 (disk) p blocksread))))
```

`Allblocksread-map-3` corresponds to the set α, and `allblocksread-map-2` corresponds to β. The following propositions are a more formal version of the arguments we used to determine the type of *allBlocksRead*(p).

Proposition 1. Consider the construct
`(defmap` f `(`dom $v_1 \ldots v_n$`) :for x :in` dom `:map (apply` v_i `x)).`
If $S \subseteq D$, and
$v_i \in$ `(all-fns` D R`)`,
then `(`f S $v_1 \ldots v_n$`)` $\subseteq R$.

Proposition 2. Consider the construct
`(defmap` f `(`dom y $v_1 \ldots v_n$`) :for x :in` dom `:map (apply-m` v_i `x` y`)).`
If $S \subseteq D_1$,
$y \in D_2$, and
$v_i \in$ `(all-fns` D_1 `(all-fns` D_2 R`))`,
then `(`f S $v_1 \ldots v_n$`)` $\subseteq R$.

Proposition 3. Consider the constructs
`(defrec` Rec `(`h_1 S_1`) ... (`h_k S_k`))` and
`(defmap` f `(`dom $v_1 \ldots v_n$`) :for x :in` dom `:map (apply x` h_i`)).`
If $S \subseteq Rec$,
then `(`f S $v_1 \ldots v_n$`)` $\subseteq S_i$.

Propositions 1 and 2 can be paraphrased as follows: if we have a function $g \in [D \to R]$ and we collect a number of its applications $g[x \in D]$, we are left with a subset of R. **Proposition 2** applies to functions of two arguments, where we collect applications of the form $g[x \in D_1][y \in D_2]$. We can contemplate analogous propositions for functions with a larger number of arguments, up to a reasonable limit.

Proposition 3 states that if we have a set of records and we collect all their fields $h_i : S_i$, we are left with a subset of S_i.

A computer program with knowledge of these propositions and with access to the Disk Synod translation, including the translation of *HInv*1, should be able to deduce the type of *allBlocksRead*(p). The program we have in mind works as follows. Let's say we want to determine the type of

the expression $(f\ \ e_1 \ldots e_n)$, where the e_i's may themselves be expressions. We recursively determine the type of each e_i, and then determine the type of $(f\ \ e_1 \ldots e_n)$, perhaps with the help of **Propositions 1–3.**

Our computer program doesn't have to be sound. Its output is a list of theorems that are submitted to ACL2. The point is to create meaningful theorems that capture the type of our constructs. Let's try our program on `allblocksread` and see if it works in this case.

```
(defun allblocksread (p blocksread)
  (allblocksread-map-2
     (union* (allblocksread-map-3 (disk) p blocksread))))
```

To determine the type of `allblocksread`, we must first determine the type of the expression

```
(union* (allblocksread-map-3 (disk) p blocksread)).          (2)
```

This leads us to ask for the type of

```
(allblocksread-map-3 (disk) p blocksread),                   (3)
where
(newdefmap allblocksread-map-3 (disk p blocksread)
  :for d :in disk :map (apply-m blocksread p d)).
```

With a slight variation of **Proposition 2**, we postulate [2]

```
(defx allblocksread-map-3-subsetp
  (implies (and (mem blocksread
                     (all-fns (proc)
                              (all-fns (disk)
                                       (powerset (blockproc)))))
                (mem p (proc)))
           (subsetp (allblocksread-map-3 (disk) p blocksread)
                    (powerset (blockproc))))
  :strategy subset-relation).
```

The above theorem is used in the proof of *HInv*1.

[2] The reader may regard the form `defx` as a `defthm` event. For more information on this construct, refer to [21].

Now let's go back to (2). First, here is a fact about `union*` that would need to be stored in our computer program.

```
(defthm union*-powerset
  (implies (subsetp s (powerset a))
           (subsetp (union* s) a))
  :hints...)
```

This fact, together with the deduced type of (3) above (captured by Theorem `allblocksread-map-3-subsetp`) lets us deduce the type of (2).

```
(defthm allblocksread-union*-map-3
  (implies (and (mem blocksread
                     (all-fns (proc)
                              (all-fns (disk)
                                       (powerset (blockproc)))))
                (mem p (proc)))
           (subsetp (union* (allblocksread-map-3 (disk) p blocksread))
                    (blockproc))))
```

Finally, we arrive at the outermost function,

```
(allblocksread-map-2 (union* (allblocksread-map-3 (disk) p blocksread))),
```
where
```
(defmap allblocksread-map-2 (allrdblks)
  :for br :in allrdblks :map (apply br "block")).
```

By **Proposition 3** and the deduced type of (2) above (captured by Theorem `allblocksread-union*-map-3`), we obtain

```
(defx subsetp-allblocksread-map-2-diskblock
  (implies (subsetp dom (blockproc))
           (subsetp (allblockqsread-map-2 dom) (diskblock)))
  :strategy subset-relation)
```
where
```
(defrec blockproc  ("block" (diskblock))  ("proc" (proc))).
```

Note that before writing down the theorem, we generalized it, using variable `dom` instead of the term

```
(union* (allblocksread-map-3 (disk) p blocksread)).
```

The above theorem is actually used in the proof of $HInv1$.

Most the effort in deducing the type of `allblocksread` involved keeping track of the expression we were working on, and deciding which one of **Propositions 1–3** to apply. The one piece of ingenuity consisted in using a theorem about `union*`. In proving $HInv1$ with ACL2, we have observed that the number of theorems invoked in deducing type information is rather small, so a modest database of theorems would probably be enough for a program that generates type lemmas. Of course, our comments must be taken with a grain of salt until such a tool is built.

After proving type lemmas about specific Disk Synod constructs, the proof of LEMMA $I2a$ goes through.

3.4 A Second Invariant

The second invariant we establish, $HInv3$, states that when certain conditions hold for two processors p and q and a disk d, at least one processor has some information about the other stored in d. Here is the invariant and the lemma to establish, as given by Gafni and Lamport.

$$
\begin{array}{l}
HInv3 \triangleq \\
\quad \forall p, q \in Proc, d \in Disk : \\
\quad \land\ phase[p] \in \{1, 2\} \\
\quad \land\ phase[q] \in \{1, 2\} \\
\quad \land\ hasRead(p, d, q) \\
\quad \land\ hasRead(q, d, p) \\
\quad \Rightarrow \lor\ [block \mapsto dblock[q], proc \mapsto q] \in blocksRead[p][d] \\
\qquad \lor\ [block \mapsto dblock[p], proc \mapsto p] \in blocksRead[q][d]
\end{array}
$$

LEMMA $(Wrong!)$ $\quad I2c \quad HInv1 \land HInv3 \land HNext \rightarrow HInv3'$

We have verified LEMMA $I2c$ with ACL2. The proof takes a total of 38 seconds, not counting the time spent in the translation of Disk Synod. A total of 137 lemmas are needed. This number will probably go down to about 50 lemmas once we develop better techniques to reason about quantification.

In addition to spotting a typographic error in Gafni and Lamport's written proof of $I2c$, we have discovered a nontrivial error in the statement of the theorem: $HInv2$ was omitted as a hypothesis. Our proof effort has yielded a correction to Lemma $I2c$, which should be modified to state

LEMMA $I2c \quad HInv1 \wedge HInv2 \wedge HInv3 \wedge HNext \Rightarrow HInv3'$.

Gafni and Lamport provide a proof outline for LEMMA $I2c$. Figure 3.1 shows the general structure of their outline. They start by assuming that the left-hand side of the implication in $HInv3$ holds for some $p, q \in Proc$ and $d \in Disk$. Then they show that the right-hand side follows for p, q and d. [3]

Let's consider case `<1>1` in Figure 3.1. In this case, since we assume both $\neg HInv3(p, q, d).L$ and $HInv3(p, q, d).L'$, it follows that one of the conjuncts in $HInv3(p, q, d).L$ must have changed from false to true. This means that

1. $\neg hasRead(p, d, q)$ and $hasRead(p, d, q)'$, *or*
2. $\neg hasRead(q, d, p)$ and $hasRead(q, d, p)'$, *or*
3. $phase[p] \notin \{1, 2\}$ and $phase'[p] \in \{1, 2\}$, *or*
4. $phase[q] \notin \{1, 2\}$ and $phase'[q] \in \{1, 2\}$.

Since Gafni and Lamport have a good intuition of their algorithm, they can predict which actions produce one of the above changes. They attribute conditions 1 and 2 to the action $Phase1or2Read$, and conditions 3 and 4 to the action $EndPhase0$. These actions are covered in cases `<2>1` through `<2>4` in Figure 3.1. Finally, Gafni and Lamport state that "Steps `<2>1-<2>4` cover the four subactions of HNext that can make one of those conjuncts true." In other words, they consider `<2>1` through

[3] Gafni and Lamport use a special notation to refer to particular pieces of a formula. The notation $HInv3(p, q, d).L'$ refers to the *primed* left-hand side of $HInv3$, instantiated with p, q and d —which in this case happen to have the same name as the variables p, q and d in $HInv3$. Recall that priming a formula means replacing all the flexible variables occurring in it by their primed versions.

```
We prove Lemma I2c by proving:

ASSUME: 1. HInv1 ∧ HInv3 ∧ HNext
        2. CONSTANTS p, q ∈ Proc, d ∈ Disk
        3. HInv3(p,q,d).L′
PROVE:  HInv3(p,q,d).R′

<1>1. CASE: ¬HInv3(p,q,d).L
   <2>1. CASE: Phase1or2Read(p,d,q)
   <2>2. CASE: Phase1or2Read(q,d,p)
   <2>3. CASE: EndPhase0(p)
   <2>4. CASE: EndPhase0(q)
   <2>5. Q.E.D.
      PROOF: By assumption 3 and the level <1> case assumption,
       one of the four conjuncts of HInv3(p,q,d).L is changed
       from false to true. Steps <2>1-<2>4 covers the four
       subactions of Next that can make one of those
       conjuncts true.
<1>2. CASE: HInv3(p,q,d).L
<1>3. Q.E.D.
  PROOF: Immediate from steps <1>1 and <1>2.
```

Figure 3.1: Highlights in Gafni and Lamport's proof of Lemma I2c. [3]

<2>4 the only interesting cases—all other possibilities leave the conjuncts of $HInv3(p, q, d).L$ unchanged.

How many uninteresting cases are there?

First, we must realize that the constants p, q and d assumed in Figure 3.1 are not the only processors and disks to consider. Let's look carefully at our hypotheses. Among them, we have

ASSUME : 1. $HInv1 \wedge HInv3 \wedge HNext$
2. CONSTANTS $p, q \in Proc, d \in Disk$
3. $HInv3(p, q, d).L'$

case $\neg HInv3(p, q, d).L$.

What does it mean to assume *HNext*? It means that for some pair of processors—call them p_2 and q_2—and some disk—call it d_2—the formula *HNext* holds, or intuitively, an action of *HNext* is "executed." Does it follow that it is the assumed CONSTANT values p, d, q which take part in this execution? Put another way, does it follow that $p_2 = p$, $q_2 = q$, and $d_2 = d$? Definitely not!

Obviously, Gafni and Lamport *know* that if our particular p, d, q are not all part of the executed action, $HInv3(p, q, d).L$ remains unchanged, making it impossible for any of conditions 1–4 to hold, and thus establishing case `<1>1` by contradiction. For ACL2—or equivalently, for someone not familiar with the algorithm—this is not necessarily obvious. Imagine the following scenarios.

- $p_2 \neq p$, $d_2 \neq d$, and $p_2 \neq q$. In this case, an action is executed by a processor p_2 different from both p and q, a processor q_2 which we know nothing about, and a disk d_2 different from d. Does it matter if $q_2 = p$ or $q_2 = q$? Does it follow that $HInv3(p, q, d).L$ remains unchanged?

- $p_2 = p$, $d_2 \neq d$, and $q_2 \neq q$. Processor p participates in the execution of an action with some processor q_2 and disk d_2. Does it follow that $HInv3(p, q, d).L$ remains unchanged?

- $p_2 = p$, $d_2 = d$, and $q_2 \neq q$. Both our processor of interest p and our disk of interest d participate in the execution of an action. We know that $q \neq q_2$, but is $q = p_2$? Does it follow that $HInv3(p, q, d).L$ remains unchanged?

- $p_2 = p$, $d_2 \neq d$, $q_2 = q$. Both p and q participate in an action, but d is not part of the execution. Does it follow that $HInv3(p, q, d).L$ remains unchanged?

- $p_2 = p$, $d_2 = d$, $q_2 = q$. Our three variables of interest participate in the execution of an action. We still must ask whether $p = q$. Does it follow that $HInv3(p, q, d).L$ remains unchanged?

We also need to consider the following scenarios, not included in cases `<2>1-<2>4` from Figure 3.1.

- Suppose that the action making *HNext* true is $StartBallot(p_2)$. Does it follow that $HInv3(p, q, d).L$ remains unchanged?

- Suppose that the action making *HNext* true is $Phase0Read(p_2, d_2)$. Does it follow that $HInv3(p, q, d).L$ remains unchanged?

- Suppose that the action making *HNext* true is $Phase1or2Write(p_2, d_2)$. Does it follow that $HInv3(p, q, d).L$ remains unchanged?

- Suppose that the action making *HNext* true is $EndPhase1or2(p_2)$. Does it follow that $HInv3(p, q, d).L$ remains unchanged?
- Suppose that the action making *HNext* true is $Fail(p_2)$. Does it follow that $HInv3(p, q, d).L$ remains unchanged?

The above scenarios are not meant to be an exhaustive consideration of all the cases that must be considered in a mechanical proof of Lemma $I2c$. But they illustrate the fact that there is a number of "trivial" cases to be considered in the mechanical verification of $I2c$.

The Well-Behaved Property

Dispatching many of the above cases depends on a property implicitly assumed by Gafni and Lamport: when a processor p changes the value of a shared variable, it changes only its own slot in the variable. Consider the variable *phase* in Disk Synod. We know its type is $[Proc \to 0..3]$.[4] This variable records some information about the processors in the system—we don't care what. For a particular processor p, $phase[p]$ holds p's share of information. When an action executes and it involves a change of p's state, only $phase[p]$ is modified, if *phase* is modified at all. We find an example in the action *InitializePhase*.

$$
\begin{array}{l}
InitializePhase(p) \triangleq \\
\quad \land\ disksWritten' = [disksWritten \text{ EXCEPT } ![p] = \{\}] \\
\quad \land\ blocksRead' = [blocksRead \text{ EXCEPT } ![p] = [d \in Disk \mapsto \{\}]]
\end{array}
$$

This action changes p's slots in the shared variables *disksWritten* and *blockRead*. However, it is not illegal to define an action that changes the values of arbitrary processors. Imagine an alternate version of *IntializePhase*.

$$
\begin{array}{l}
EvilPhase(p) \triangleq \\
\quad \textbf{let } p_{evil} \triangleq \text{CHOOSE } Proc \\
\quad \textbf{in } \quad \land\ disksWritten' = [disksWritten \text{ EXCEPT } ![p_{evil}] = \{\}] \\
\qquad\qquad \land\ blocksRead' = [blocksRead \text{ EXCEPT } ![p_{evil}] = [d \in Disk \mapsto \{\}]]
\end{array}
$$

The action *EvilPhase* changes $disksWritten[p_{evil}]$ and $blocksRead[p_{evil}]$ for some processor p_{evil} in the set of processors *Proc*, not necessarily the p its argument advertises.

[4] The TLA notation $m..n$ denotes the set $\{i \in Nat : m \leq i \leq n\}$ when $m \leq n$.

Gafni and Lamport assume this kind of behavior never happens in their specification of Disk Synod (and they are correct). The theorem capturing their assumption is straightforward to prove in ACL2.

```
(defthm wb-hnext
  (implies (and (hnext p2 d2 q2 b-witness1 ip-witness1 b-witness)
                (not (= p2 p)))
           (and (= (apply input-n p) (apply input p))
                (= (apply output-n p) (apply output p))
                (= (apply-m disk-n d p) (apply-m disk d p))
                (= (apply phase-n p) (apply phase p))
                (= (apply dblock-n p) (apply dblock p))
                (= (apply diskswritten-n p) (apply diskswritten p))
                (= (apply-m blocksread-n p d)
                   (apply-m blocksread p d))
                (= (apply blocksread-n p) (apply blocksread p)))))
```

We now illustrate how the well-behaved property can be used to deal with some cases. We only do this for the easiest one.

- $p_2 \neq p$, $d_2 \neq d$, and $p_2 \neq q$. An action is executed by a processor p_2 different from both p and q, a processor q_2 which we know nothing about, and a disk d_2 different from d. Does it matter if $q_2 = p$ or $q_2 = q$? Does it follow that $HInv3(p, q, d).L$ remains unchanged?

In this case, we can use the well-behaved property to reduce $HInv3'$ to $HInv3$. The intuition is that if p, d and q have nothing to do with the action executed, then $HInv3(p, q, d)$, which states a relationship between p, d and q, should remain unchanged after *HNext* executes. Note that we can assume $HInv3(p, q, d)$, since it is an instance of the assumption $HInv3$.

Translating $HInv3(p, q, d)'$ yields the following ACL2 expression.

```
1.  (implies (and (mem (apply phase-n p) (hide (brace 1 2)))
2.                (mem (apply phase-n q) (hide (brace 1 2)))
3.                (hasread p d q blocksread-n)
4.                (hasread q d p blocksread-n))
5.           (or (mem (func ("block" (apply dblock-n q)) ("proc" q))
6.                    (apply-m blocksread-n p d))
7.               (mem (func ("block" (apply dblock-n p)) ("proc" p))
8.                    (apply-m blocksread-n q d))))
```

Given the hypothesis `(hnext p2 d2 q2 b-witness1 ip-witness1 b-witness)` and the assumption `(not (= p2 p))`, we can use `wb-hnext` to deduce `(= (apply phase-n p) (apply phase p))`. Using this equality, line `1` becomes

```
1*.  (implies (and (mem (apply phase p) (hide (brace 1 2))).
```

The same reasoning applied to `(not (= q2 p))` transforms line `2` into

```
2*.  (mem (apply phase q) (hide (brace 1 2))).
```

The definition of `hasread` is

```
(defun hasread (p d q blocksread)
  (exists-hasread (apply-m blocksread p d) q)).
```

Using this definition, the assumption `(not (= p2 p))`, and the well-behaved property, we rewrite lines 3 and 4 as follows.

```
(hasread p d q blocksread-n)
=
(exists-hasread (apply-m blocksread-n p d) q))
=
(exists-hasread (apply-m blocksread p d) q))
=
(hasread p d q blocksread).
```

Continuing in this manner, we are able to replace all the primed variables in lines 1–8 with unprimed variables, which implies

$$HInv3(p, q, d) \equiv HInv3(p, q, d)'.$$

Since $HInv3 \Rightarrow HInv3(p, q, d)$ and $HInv3(p, q, d) \equiv HInv3(p, q, d)'$, we have $HInv3 \Rightarrow HInv3(p, q, d)'$, establishing LEMMA $I2c$ for the case we're considering.

Notice that we have established the invariance of $Hinv3$ for the case "$p_2 \neq p$, $d_2 \neq d$, and $p_2 \neq q$." This case has no correspondence to any of the cases in Gafni and Lamport's outline (Figure 3.1). We are following an

altogether different proof of Lemma $I2c$. Figure 3.2 shows an outline of our proof. We have marked some points of interest. Line `*3*` corresponds to the case we just worked out. Line `*1*` corresponds to Gafni and Lamport's case `<2>1`. Notice the effort involved just to arrive at their starting point. In fact, once we arrive at case `<4>1` in line `*1*`, establishing the conclusion for this particular case is straightforward. Most of our effort is involved in considering all the case splits that Gafni and Lamport avoid. Line `*2*` (case `<3>3`) produces the same number of case splits as `<3>2`. To save space, we do not include all cases in our proof outline.

Ideally, ACL2 would have easily followed Gafni and Lamport's proof. We would like to give ACL2 a high-level hint to split our proof into the numerous cases in Figure 3.2 and have it establish automatically those which the designers of Disk Synod considered uninteresting. The verdict on this front is not in—we are in the process of analyzing our proof of *HInv*3 to see if we can follow Gafni and Lamport's proof.

Still, we claim that our own proof is not unreasonable. It gives as much space to the "obvious" as it does to the "non-obvious" but it makes explicit the fact that *HNext* is a general assumption that doesn't necessarily apply to a specific processor we may have in mind.

3.5 Issues in Verification

This section presents two examples of issues confronted by the user of a verification system. The first deals with representation. In TLA, one can be an elegant mathematician. But sometimes elegance gets in the way of proofs—it can be difficult to prove basic properties about a construct in its original TLA form. To establish these properties, we may need to take a step back and define an equivalent construct, in a style more amenable to ACL2. After proving the desired properties for the new construct, we must relate it back to its TLA counterpart.

The second example illustrates the reality that no theorem prover is ready-made for every task. It poses a problem in the interplay between ACL2 and our translation scheme, and offers a possible solution.

Finding a Maximal Element

Consider a construct found in actions $EndPhase0(p)$ and $EndPhase1or2(p)$ in the Disk Synod specification. The construct appears unnamed within the actions; here we give it a name.

```
We prove Lemma I2c by proving:

ASSUME: 1. HInv1 ∧ HInv2 ∧ HInv3
        2. HNext(p2,d2,q2)
        2. CONSTANTS p, q ∈ Proc, d ∈ Disk
        3. HInv3(p,q,d).L′
PROVE:  HInv3(p,q,d).R′

<1>1. CASE: StartBallot(p2)
<1>2. CASE: Phase0Read(p2,d2)
<1>3. CASE: Phase1or2Write(p2,d2)
<1>4. CASE: EndPhase1or2(p2)
<1>5. CASE: Fail(p2)
<1>6. CASE: Phase1or2Read(p2,d2,q2)
      <2>1. CASE: d2 = d
            <3>1. CASE: p2 ≠ p ∧ p2 ≠ q
            <3>2. CASE: p2 = p
                  <4>1. CASE: q2 = q                                    *1*
                        <5>1. CASE: p = q
                        <5>2. CASE: p ≠ q
                        <5>3. Q.E.D. Immediate from <5>1 and <5>2.
                  <4>2. CASE: q2 ≠ q
                  <4>3. Q.E.D. Immediate from <4>1 and <4>2.
            <3>3. CASE: p2 = q                                          *2*
            <3>4. Q.E.D. Immediate from steps <3>1-<3>3.
      <2>2. CASE: d2 ≠ d
            <3>1. CASE: p2 ≠ p ∧ p2 ≠ q                                 *3*
            <3>2. CASE: p2 = p
            <3>3. CASE: p2 = q
            <3>4. Q.E.D. Immediate from steps <3>1-<3>3.
      <2>3. Q.E.D. Immediate from steps <2>1 and <2>2.
<1>7. CASE: EndPhase0(p2)
<1>8. Q.E.D. Cases <1>1-<1>7 cover all the actions of Next.
```

Figure 3.2: Our proof of Lemma I2c.

$$\begin{array}{l} \mathit{ChooseMaxBal} \triangleq \\ \quad \text{CHOOSE } r \in \mathit{allBlocksRead}(p) : \forall s \in \mathit{allBlocksRead}(p) : r.\mathit{bal} \geq s.\mathit{bal} \end{array}$$

This expression denotes a record from the set $\mathit{allBlocksRead}(p)$ with maximal *bal* field. The element of a set with some "maximal" property is a concept bound to come up in different contexts, so let's generalize our discussion and consider instead

$$\mathit{ChooseMax} \triangleq \text{CHOOSE } r \in S : \forall x \in S : g(r) \geq g(x).$$

Leaving g undefined, we capture the statement in its most general form. Here is its translation.

```
(encapsulate
 ((g (x) t))
 (local (defun g (x) (declare (ignore x)) t))
 (defcong = = (g x) 1))
(defall forall-x-is-greater (y s)
  :forall x :in s
  :holds (>= (g r) (g x)))
(defmap collect-maximal-elements (s1 s2)
  :for x :in s1
  :such-that (forall-x-is-greater x s2))
(defun choose-max (s)
  (choose (collect-maximal-elements s s)))
```

`Choose-max` has the important `=`-congruence property (see Section 1.3).

```
(defcong = = (choose-max s) 1)
```

However, it is difficult to prove the two important theorems about `choosemax`:

```
(defthm choose-max-exists
  (implies (not (ur-elementp s))
           (mem (choose-max s) s))).


(defthm choose-max->=
  (implies (mem x s)
           (>= (g (choose-max s))
               (g x)))).
```

A more natural way to define *ChooseMax* is to define a recursive function that finds the maximal element. Having such a function, it is much easier to prove the two above theorems. In fact, they follow immediately for the function below.

```
(defun choose-max-alt (s)
  (cond ((ur-elementp s) nil)
        ((ur-elementp (scdr s)) (scar s))
        ((>= (g (scar s))
             (g (choose-max-alt (scdr s))))
         (scar s))
        (t (choose-max-alt (scdr s)))))
(defthm choose-max-alt-exists
  (implies (not (ur-elementp s))
           (mem (choose-max-alt s) s)))
(defthm choose-max-alt->=
  (implies (mem x s)
           (>= (g (choose-max-alt s)) (g x))))
```

Unfortunately, `choose-max-alt` doesn't respect `=`-congruence. The reason is that `(choose-max-alt S)` picks the first maximal element it finds in the list representation of the set `S`. If two equal sets `S1` and `S2` having more than one maximal element happen to have their maximal elements presented in different order, the values returned by `(choose-max-alt S1)` and `(choose-max-alt S2)` could differ.

Our goal is to prove `choose-max-exists` and `choose-max->=` with the help of `choose-max-alt`. The key is to relate both functions, `choose-max` and `choose-max-alt`. We cannot prove them equivalent, because they aren't. But we can establish facts that let us transfer theorems from `choose-max-alt` to `choose-max`. Here are two observations.

1. If S is not empty, then `(choose-max-alt` S`)` is an element of

   `(collect-maximal-elements` S S`)`,

   which is involved in the definition of `choose-max`.

2. If S is not empty, then

   `(fix (g (choose-max` S`)))` = `(fix (g (choose-max-alt` S`)))`.

Observations like (1) and (2) let us translate knowledge about `choose-max-alt` into knowledge about `choose-max`. For the ACL2 events corresponding to this claim, see Appendix D.2.

The Witness Problem

Consider the definition of action *Fail* in the Disk Synod algorithm.

$$
\begin{array}{l}
Fail(p) \triangleq \\
\quad \land \exists ip \in Inputs : input' = [input \text{ EXCEPT } ![p] = ip] \\
\quad \land phase' = [phase \text{ EXCEPT } ![p] = 0] \\
\quad \land dblock' = [dblock \text{ EXCEPT } ![p] = InitDB] \\
\quad \land output' = [output \text{ EXCEPT } ![p] = NotAnInput] \\
\quad \land InitializePhase(p) \\
\quad \land \text{UNCHANGED } disk
\end{array}
$$

It translates into the following ACL2 events.

```
(defexists exists-fail-1 (inputs input input-n p)
  :exists ip :in inputs :such-that (= input-n (except input p ip))
  :mem-corollary nil)
(defaction fail (p)
  (exists-fail-1 (inputs) input input-n p)
  (= phase-n  (except phase p 0))
  (= dblock-n (except dblock p (initdb)))
  (= output-n (except output p (notaninput)))
  (initializephase p diskswritten diskswritten-n
                   blocksread blocksread-n)
  (unchanged disk))
```

Now, imagine a function Θ with the following property.

```
(defthm theta-lemma
  (implies (mem y (inputs))
           (Θ (except w x y))))
```

We are asked to prove the following theorem.

```
(defthm fail-implies-theta
 (implies (fail p)
          (Θ input-n)))
```

The way to proceed is to use lemma `exhibit-member-fail-1`.

```
(defthm exhibit-member-fail-1
  (iff (exists-fail-1 (inputs) input intput-n p)
       (and (mem (choose
                   (exists-fail-1-map (inputs) input
                                      input-n p))
                 inputs)
            (= input-n
               (except input
                       p
                       (choose
                         (exists-fail-1-map (inputs) input
                                            input-n p)))))))
```

`Exhibit-member-fail-1` creates a witness object that satisfies the properties stated in the existentially quantified statement

$\exists ip \in Inputs : input' = [input \text{ EXCEPT } ![p] = ip]$.

Inspecting `exhibit-member-fail-1`, we find that the witness is

$W \triangleq$ `(choose (exists-fail-1-map (inputs) input input-n p))`.

How is W created? Behind the scenes, `defexists` creates a function `exists-fail-1-map`, analogous to `exists-fail-1`, that collects all elements satisfying

```
(= input-n (except input p ip)).
```

If `(exists-fail-1 (inputs) input input-n p)` holds, then some element in `(inputs)` satisfies the above equation, and `(exists-fail-1-map (inputs) input input-n p))` is nonempty. W collects an element from this nonempty set.

We can use `exhibit-member-fail-1` to rewrite `fail-implies-theta`.

```
(implies (and (and (mem W (inputs))
                   (= input-n
                      (except input p W)))
              (= phase-n  (except phase p 0))
              (= dblock-n (except dblock p (initdb)))
              (= output-n (except output p (notaninput)))
              (initializephase p diskswritten diskswritten-n
                               blocksread blocksread-n)
              (unchanged disk)
         (Θ input-n)))
```

To establish `(Θ input-n)`, we replace `input-n` by `(except input p W)` in the conclusion. Then `fail-implies-theta` follows from `theta-lemma`. However, ACL2 doesn't replace `input-n`. The reason is that `input-n` occurs inside W:

```
(choose (exists-fail-1-map (inputs) input input-n p)).
```

ACL2 reasonably refuses to replace a variable with an expression containing it—this could lead to an infinite loop. But without the replacement, the conclusion `(Θ input-n)` remains unchanged, and we are unable to use `theta-lemma` and prove the theorem.

Currently, this problem is unresolved. To succeed in our proofs, we depart from our translation scheme, translating statements of the form $\exists x \in S : p(x)$ into

```
(and (mem w_i S)
     (p w_i)).
```

We are careful to choose variables w_i that do not occur in any other context within the translated expression. *HNext* needs six such variables. That is why a call of *HNext* looks like

```
(hnext p d q b-witness1 ip-witness1 b-witness).
```

In LEMMA $I2a$ and LEMMA $I2c$, all existentially quantified expressions appear in *HNext*, which is on the left side of the implication sign. So in this case our new translation is sound. We consider this a temporary fix, not a permanent alternative.

A solution proposed by Kaufmann and Moore[9] would involve a change in ACL2's generalization strategy. The idea is that when ACL2 encounters a function symbol flagged as "generalizable," it generalizes the current expression. In our case, `choose` would be a generalizable function symbol. (If `choose` is used in other contexts where we do not want generalization to happen, we might create a function `choose-witness`, equal to `choose`, and flag it as generalizable, leaving the original `choose` unchanged.) Now, when ACL2 runs into W, it generalizes it to a fresh variable[5] and produces a stronger goal.

```
(implies (and (and (mem W23 (inputs))
                   (= input-n
                      (except input p W23)))
              (= phase-n  (except phase p 0))
              (= dblock-n (except dblock p (initdb)))
              (= output-n (except output p (notaninput)))
              (initializephase p diskswritten diskswritten-n
                               blocksread blocksread-n)
              (unchanged disk))
         (Θ input-n))
```

Now, `input-n` is successfully replaced by `(except input p W23)`, and `fail-implies-theta` is established.

This addition to ACL2 is currently the best proposed solution to our problem, because it leaves our translation scheme intact.

[5] By 'fresh', we mean a variable that doesn't already appear in the goal.

Chapter 4

Conclusion

Our project has accomplished several objectives. The first and most stable result is a straightforward translation scheme. We have also become familiar with the structure of invariant proofs in TLA. We successfully used ACL2 to prove two invariants of Disk Synod. Our mechanical proof of *HInv*3 suggests that the translation scheme can be used effectively to verify nontrivial invariants. A rewarding aspect of proving nontrivial invariants of a realistic algorithm is that we have taken as a starting point a non-toy problem, and learned much from each invariant proved. Starting with something as large as Disk Synod might seem like a bad idea for a verification experiment, but in our case it proved beneficial—it forced us to come up with a translation that would make sense for a sizable specification, and to develop our system so it could handle relatively large invariants from the outset. Our success with Disk Synod attests to the system's strength, and shows promise for attacking larger verification projects.

We have also developed a good intuition for type invariance proofs of TLA specifications, and are confident that a program can be written that mechanically generates most of the lemmas needed to establish type invariance.

Our proofs of *HInv*1 and *HInv*3 are fast. The proof of *HInv*1 breaks into approximately 200 cases. Adding the time taken by ACL2 to establish auxiliary lemmas, each case is proved in approximately 0.25 seconds. The proof of *HInv*3 is faster: all auxiliary lemmas plus the main theorem are established in 38 seconds.

Many issues are still unresolved. The most important is our desire to have ACL2 focus on the most interesting parts of a proof and establish uninteresting aspects automatically. In [10], Lamport et al. comment on the use of a theorem prover to verify concurrent systems.

> Ultimately, one reaches a point where prose can be eliminated and the proof checked by computer. However, the function of proofs in engineering is not to attain absolute certainty, but to achieve a reasonable degree of confidence with a reasonable amount of effort. We believe that, at the moment, for many large applications, the most cost-effective approach stops short of mechanical verification.

Our continuing research will focus on strengthening our system to make mechanical verification cost-effective at all levels of proof.

4.1 Further Work

To conclude, we offer a list of outstanding tasks.

Short-term Further Work

- As it turns out, we don't use $HInv1$ in the proof of $I2c$, even though it is necessary. Is this alarming? We claim it isn't, in this particular case. The reason we don't use $HInv1$ is exemplified precisely in Chapter 2:

  > We deduce $phase'[p] = 2$ from $phase' = [phase \text{ EXCEPT } ![p] = 2]$ only if $phase$ is a function whose domain contains p. However, in ACL2, given `(= phase-prime (except phase p 2))`, we can deduce `(= (apply phase-prime p) 2))` with no further hypotheses about `phase`.

  In other words, the hypotheses that ACL2 fails to ask for are type hypotheses. But we prove type invariance separately, so (for example) while ACL2 does not ask for the hypothesis `(mem` p `(proc))` whenever it encounters an expression of the form `(except input` p y`)`, this hypothesis would be successfully relieved by virtue of $HInv1$'s established invariance.

  Some solutions to this problem have been offered. One of them uses ACL2's guard mechanism to certify the correct "type" of TLA specifications. This work is in very early stages so we do not discuss it further, and note that our next immediate goal in the project is to resolve the mismatch between ACL2 and TLA.

- Reasoning about nested quantifiers like $\forall\, x, y, z \in S : p(x, y, z)$ remains a manual task. We need methods that can automatically establish the above fact from a proof of $x, y, z \in S \Rightarrow p(x, y, z)$.

- Our current version of `defaction` is hardwired for Disk Synod actions. A more sophisticated version would read a list of flexible variables (usually declared at the beginning of a TLA specification), and afterwards generate the appropriate functions with respect to the specification at hand. The same applies to `defstate`.

- In this first experiment, hand translation of Disk Synod helped us develop intuition for a good translation scheme. Larger verification projects will require an automatic translator. The desire for automation guided many of our translation decisions.

- The naming convention discussed in Section 2.2 needs to be formulated, presumably when implementing an automatic translator.

- The "type lemma" generator discussed in Section 3.3 should follow—and possibly coexist with—a TLA–ACL2 translator. Regardless of general invariant verification, a type checker alone would be a useful tool.

Long-term Further Work

- A graphical interface to ACL2 has been discussed before by ACL2 users—it would be particularly helpful in our system. We mentioned the problem of name generation for unnamed concepts in TLA. An interface that allowed us to, say, place the mouse over a name and see the logical construct it represents, would speed up the verification process.

- An efficiently executable model of a TLA specification would allow the user to "run" the specification and learn from its behavior. Since our translated specifications *are* Lisp code, can we execute them? The theoretical answer is yes, but the practical answer is no—consider the amount of time needed to execute (`mem` e (`all-fns` D R)) even if D and R are small sets. A solution would be to provide an escape mechanism: an expression like the above would not be computed. Instead, ACL2 would inspect e and check if its elements are of the proper form.

- We have focused on nontemporal aspects of TLA verification. If our system could be tied to a prover that dealt with temporal properties, we would have a complete and powerful framework to reason about TLA specifications.

Acknowledgements

I am deeply grateful to my advisor, Professor J Moore, for his encouragement, trust, and wisdom, and for being my first role model, a person I admire and aspire to emulate. I also thank Professor Bob Boyer for answering questions large and small, and for reviewing this work.

Bibliography

[1] Martín Abadi and Stephan Merz. On TLA as a Logic. M. Broy, editor, *Deductive Program Design*, Springer-Verlag, NATO ASI series F, 1996.

[2] Urban Engberg, *Reasoning in the Temporal Logic of Actions.* PhD thesis, Aarhus University, 1994.

[3] Eli Gafni and Leslie Lamport. Disk Paxos. Technical Report 163, Compaq Systems Research Center, July 2000.

[4] Eli Gafni and Leslie Lamport. Disk Paxos. in Maurice Herlihy, editor, Distributed Computing: 14th International Conference, DISC 2000 *Lecture Notes in Computer Science number 1914,* pages 330-344, Springer-Verlag, 2000.

[5] Stephen J. Garland and Nancy A. Lynch. Using I/O Automata for Developing Distributed Systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems,* pages 285-312, Cambridge University Press, 2000.

[6] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, *Computer-Aided Reasoning: An Approach,* Kluwer Academic Publishers, 2000.

[7] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies,* chapter 6, Kluwer Academic Publishers, 2000.

[8] Matt Kaufmann and J Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning,* 26(2), pp. 161-203, 2001.

[9] Matt Kaufmann and J Moore. Personal communication.

[10] Peter Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel, A Lazy Caching Proof in TLA, *Distributed Computing 12, 2/3,* pages 151-174, 1999.

[11] Thomas Långbacka. A HOL formalisation of the Temporal Logic of Actions. In Thomas E Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications,* volume 859 of *Lecture Notes in Computer Science,* pages 332-345, Berlin, 1994. Springer-Verlag.

[12] Leslie Lamport. *A Summary of TLA$^+$.* June 2000. In `http://www.research.compaq.com/SRC/personal/lamport/tla/papers.html`.

[13] Leslie Lamport. How to Write a Proof. *American Mathematical Monthly 102, 7 (August-September 1993)* pages 600-608.

[14] Leslie Lamport. *Specifying Concurrent Systems with TLA+.* Unpublished draft (dated February 9, 2000).

[15] Leslie Lamport. *Some Thoughts on Specification.* Message posted to the TLA mailing list, May 5 1992. Available at `http://www.research.compaq.com/SRC/personal/lamport/tla/notes.html`.

[16] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems,* 16(3):872-923, May 1994.

[17] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, July 1978.

[18] Leslie Lamport. Personal communication.

[19] Leslie Lamport and Stephan Merz. Specifying and Verifying Fault-Tolerant Systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, H. Langmaack, W.-P.de Roever, and J. Vytopil, editors. LNCS 863, 41-76.

[20] Stephan Merz. An Encoding of TLA in Isabelle. In `http://www.informatik.uni-muenchen.de/~merz/isabelle/index.html` Institut für Informatik, Universität München, Germany.

[21] J Moore. Finite Set Theory in ACL2. January 2001. Submitted for publication.
`http://www.cs.utexas.edu/users/moore/publications/finite-set-theory/index.html`

[22] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299-319, December 1990

Appendix A

The Increment Example

```
; increment.lisp

(include-book "/projects/acl2/v2-5/books/finite-set-theory/set-theory")
(include-book "../paxos/acl2-files/tla-translation-macros")
(in-package "S")

(defmacro defaction (name a)
  `(progn
     (defun ,(packn-in-pkg (list "_" name) 'defaction)
       (pc1 pc1-n pc2 pc2-n x x-n y y-n sem sem-n)
       ,a)
     (defmacro ,name ()
       '(,(packn-in-pkg (list "_" name) 'defaction)
 pc1 pc1-n pc2 pc2-n x x-n y y-n sem sem-n))))

(defmacro defstate (name a)
  `(progn
     (defun ,(packn-in-pkg (list "_" name) 'defaction)
       (pc1 pc2 x y sem)
       ,a)
     (defmacro ,name ()
       '(,(packn-in-pkg (list "_" name) 'defaction)
 pc1 pc2 x y sem))
     (defmacro ,(packn-in-pkg (list name "-N") 'defaction) ()
       '(,(packn-in-pkg (list "_" name) 'defaction)
 pc1-n pc2-n x-n y-n sem-n))))

(defstate init
  (and (= pc1 "a") (= pc2 "a")
       (= x 0) (= y 0)
       (= sem 1)))
```

```
(defaction alpha1
  (and (= pc1 "a") (< 0 sem)
       (= pc1-n "b")
       (= sem-n (- sem 1))
       (unchanged x y pc2)))

(defaction alpha2
  (and (= pc2 "a") (< 0 sem)
       (= pc2-n "b")
       (= sem-n (- sem 1))
       (unchanged x y pc1)))

(defaction beta1
  (and (= pc1 "b")
       (= pc1-n "g")
       (= x-n (+ x 1))
       (unchanged y sem pc2)))

(defaction beta2
  (and (= pc2 "b")
       (= pc2-n "g")
       (= y-n (+ y 1))
       (unchanged x sem pc1)))

(defaction gamma1
  (and (= pc1 "g")
       (= pc1-n "a")
       (= sem-n (+ sem 1))
       (unchanged x y pc2)))

(defaction gamma2
  (and (= pc2 "g")
       (= pc2-n "a")
       (= sem-n (+ sem 1))
       (unchanged x y pc1)))

(defaction n1
  (or (alpha1) (beta1) (gamma1)))

(defaction n2
  (or (alpha2) (beta2) (gamma2)))

(defaction n
  (or (n1) (n2)))
```

```
(defun _i (x sem pc1 pc2)
  (and (and (integerp x) (>= x 0))
       (or (and (= sem 1) (= pc1 "a") (= pc2 "a"))
   (and (= sem 0) (or (and (= pc1 "a") (mem pc2 (brace "b" "g")))
      (and (= pc2 "a") (mem pc1 (brace "b" "g")))))))))

(defmacro i ()
  '(_i x sem pc1 pc2))

(defmacro i-n ()
  '(_i x-n sem-n pc1-n pc2-n))

(thm
 (implies (and (i) (n))
  (i-n)))

; end of file
```

Appendix B

Disk Synod Specification

Here, we provide the TLA specification of the Disk Synod algorithm. Our specification is organized slightly differently than Gafni and Lamport's because we do not make use of TLA$^+$'s module system. In particular, we introduce some TLA contructs that aren't strictly part of Disk Synod (like action *HNext*), and are defined in a separate module in Gafni and Lamport's paper. This does not affect the validity of the constructs or the theorems. For more details, refer to [3].

```
vars ≜ ≪ input, output, disk, phase, dblock, disksWritten, blocksRead ≫

CONSTANT N, Inputs
ASSUME (N ∈ Nat) ∧ (N > 0)

Proc ≜ 1..N

NotAnInput ≜ CHOOSE c : c ∉ Inputs

CONSTANTS Ballot(_), Disk, IsMajority(_)

ASSUME ∧ ∀ p ∈ Proc : Ballot(p) ⊆ {n ∈ Nat : n > 0}
       ∧ ∀ q ∈ Proc  p : Ballot(p) ∩ Ballot(q) = {}
       ∧ ∀ S,T ∈ SUBSET Disk :
                 IsMajority(S) ∧ IsMajority(T) ⇒ (S ∩ T ≠ {})

DiskBlock ≜ [ mbal : (UNION {Ballot(p) : p ∈ Proc }) ∪ {0},
              bal  : (UNION {Ballot(p) : p ∈ Proc }) ∪ {0},
              inp  : Inputs ∪ {NotAnInput}
            ]

hasRead(p,d,q) ≜ ∃ br ∈ blocksRead[p][d] : br.proc = q
```

```
allBlocksRead(p) ≜
   LET allRdBlks ≜ UNION { blocksRead[p][d] : d ∈ Disk }
   IN  { br.block : br ∈ allRdBlks }

InitDB ≜ [ mbal ↦ 0 , bal ↦ 0 , inp ↦ NotAnInput ]

Init ≜ ∧ input     ∈ [Proc → Inputs]
       ∧ output      = [p ∈ Proc ↦ NotAnInput]
       ∧ disk        = [d ∈ Disk ↦ [p ∈ Proc ↦ InitDB]]
       ∧ phase       = [p ∈ Proc ↦ 0]
       ∧ dblock      = [p ∈ Proc ↦ InitDB]
       ∧ disksWritten = [p ∈ Proc ↦ {}]
       ∧ blocksRead   = [p ∈ Proc ↦ [d ∈ Disk ↦ {}]]

InitializePhase(p) ≜
  ∧ disksWritten′ = [disksWritten EXCEPT ![p] = {}]
  ∧ blocksRead′   = [blocksRead   EXCEPT ![p] = [d ∈ Disk ↦ {}]]

StartBallot(p) ≜
   ∧ phase[p] ∈ {1,2}
   ∧ phase′ = [phase EXCEPT ![p] = 1]
   ∧ ∃ b ∈ Ballot(p) :
         ∧ b > dblock[p].mbal
         ∧ dblock′ = [dblock EXCEPT ![p].mbal = b]
   ∧ InitializePhase(p)
   ∧ UNCHANGED <input,output,disk>

Phase1or2Write(p,d) ≜
  ∧ phase[p] ∈ {1,2}
  ∧ disk′ = [disk EXCEPT ![d][p] = dblock[p]]
  ∧ disksWritten′ = [disksWritten EXCEPT ![p] = @ ∪ {d}]
  ∧ UNCHANGED <input,output,phase,dblock,blocksRead>

Phase1or2Read(p,d,q) ≜
  ∧ d ∈ disksWritten[p]
  ∧ IF disk[d][q].mbal < dblock[p].mbal
    THEN ∧ blocksRead′ =
              [blocksRead EXCEPT ![p][d] =
               @ ∪ { [block /→ disk[d][q], proc /→ q] } ]
         ∧ UNCHANGED <input,output,disk,phase,dblock,disksWritten>
    ELSE
     StartBallot(p)
```

```
Phase0Read(p,d) ≜
  ∧ phase[p] = 0
  ∧ blocksRead′ = [blocksRead EXCEPT
                     ![p][d] = @ ∪ { [ block /→ disk[d][p],
                                       proc /→ p ] } ]
  ∧ UNCHANGED <input,output,disk,phase,dblock,disksWritten>

Fail(p) ≜
   ∧ ∃ ip ∈ Inputs : input′ = [input EXCEPT ![p] = ip]
   ∧ phase′  = [phase  EXCEPT ![p] = 0 ]
   ∧ dblock′ = [dblock EXCEPT ![p] = InitDB]
   ∧ output′ = [output EXCEPT ![p] = NotAnInput]
   ∧ InitializePhase(p)
   ∧ UNCHANGED disk

EndPhase0(p) ≜

   ∧ phase[p] = 0
   ∧ isMajority( { d ∈ Disk : hasRead(p,d,p) } )
   ∧ ∃ b ∈ Ballot(p) :
        ∧ forall r ∈ allBlocksRead(p) : b > r.mbal
        ∧ dblock′ = [ dblock EXCEPT
                        ![p] = [ (CHOOSE r ∈ allBlocksRead(p) :
                                    forall s ∈ allBlocksRead(p) :
                                      r.bal ≥ s.bal)
                                 EXCEPT !.mbal = b ] ]
   ∧ InitializePhase(p)
   ∧ phase′ = [ phase EXCEPT ![p] = 1 ]
   ∧ UNCHANGED <input,output,disk>

EndPhase1or2(p) ≜
   ∧ IsMajority({ d ∈ disksWritten[p] :
                    forall q ∈ Proc  {p} : hasRead(p,d,q)})
   ∧ ∨  ∧ phase[p] = 1
        ∧ dblock′ =
            [dblock EXCEPT
              ![p].bal = dblock[p].mbal,
              ![p].inp =
                LET blocksSeen ≜ allBlocksRead(p) ∪ { dblock[p] }
                    nonInitBlks ≜
                      { bs ∈ blocksSeen : bs.inp ≠ NotAnInput }
                    maxBlk ≜
                      CHOOSE b ∈ nonInitBlks :
                         forall c ∈ nonInitBlks : b.bal ≥ c.bal
```

```
                IN
                  IF nonInitBlks = {} THEN input[p]
                                      ELSE maxBlk.inp ]
          ∧ UNCHANGED output
       ∨  ∧ phase[p] = 2
          ∧ output′ = [output EXCEPT ![p] = dblock[p].inp]
          ∧ UNCHANGED dblock
    ∧ phase′ = [phase EXCEPT ![p] = @ + 1]
    ∧ InitializePhase(p)
    ∧ UNCHANGED <input, disk>

MajoritySet ≜ { D ∈ SUBSET Disk : IsMajority(D) }

 blocksOf(p) ≜
  LET rdBy(q,d) ≜ {br ∈ blocksRead[q][d] : br.proc = p }
  IN { dblock[p] } ∪ { disk[d][p] : d ∈ Disk }
                 ∪ { br.block : br ∈ UNION {rdBy(q,d) : q ∈ Proc,
                                                        d ∈ Disk }}

 allBlocks ≜ UNION { blocksOf(p) : p ∈ Proc

Next ≜
   ∃ p ∈ Proc :
        ∨ StartBallot(p)
        ∨ ∃ d ∈ Disk : ∨ Phase0Read(p,d)
                       ∨ Phase1or2Write(p,d)
                       ∨ ∃ q ∈ Proc  {p} :
                            Phase1or2Read(p,d,q)
        ∨ EndPhase1or2(p)
        ∨ Fail(p)
        ∨ EndPhase0(p)

HNext ≜
  ∧ Next
  ∧ chosen′ = LET hasOutput(p) ≜ output′[p] ≠ NotAnInput
              IN IF ∨ chosen ≠ NotAnInput
                    ∨ forall p in Proc : -hasOutput(p)
                 THEN chosen
                 ELSE output′[CHOOSE p in Proc : hasOutput(p)]
  ∧ allInput′ = allInput cup { input′[p] : p in Proc }
```

Appendix C

Disk Synod Translations

```
vars ≜ ≪ input, output, disk, phase, dblock, disksWritten, blocksRead ≫

CONSTANT N, Inputs
ASSUME (N ∈ Nat) ∧ (N > 0)

Proc ≜ 1..N

NotAnInput ≜ CHOOSE c : c ∉ Inputs
```

```
(encapsulate ((n1 () t))

             (local (defun n1 () 1))

             (defthm n1-constraint
               (and (integerp (n1))
                    (< 0 (n1)))
               :rule-classes :type-prescription))

(defstub inputs () t)

(defun Proc () (dot-dot 1 (n1)))

(encapsulate ((notaninput () t))
             (local (defun notaninput () (inputs)))
             (local (defthm notaninput-helper
                      (equal (acl2::hide (notaninput))
                             (inputs))
                      :hints (("Goal" :expand (acl2::hide (notaninput))
                                      :in-theory
                                      (disable
                                       (:executable-counterpart notaninput))))))

             (defthm notaninput-constraint
               (not (mem (notaninput) (inputs)))))
```

```
CONSTANTS Ballot(_), Disk, IsMajority(_)

ASSUME ∧ ∀ p ∈ Proc : Ballot(p) ⊆ {n ∈ Nat : n > 0}
       ∧ ∀ q ∈ Proc  p : Ballot(p) ∩ Ballot(q) = {}
       ∧ ∀ S,T ∈ SUBSET Disk :
                 IsMajority(S) ∧ IsMajority(T) ⇒ (S ∩ T ≠ {})
```

```
(encapsulate ((ballot (p) t))
             (local (defun ballot (p) (declare (ignore p)) nil))

             (defthm ballot-is-set-of-nats
               (implies (and (mem b (ballot p))
                             (mem p (Proc)))
                        (and (integerp b)
                             (> b 0)))
               :rule-classes ((:forward-chaining
                               :trigger-terms ((mem b (ballot p))))))

             (defcong = equal (ballot p) 1)

             (defthm ballot-partitions-nats
               (implies (mem q (diff (proc) (brace p)))
                        (ur-elementp (intersection (ballot p) (ballot q))))))

(encapsulate ((disk () t)
              (ismajority (s) t))
             (local (defstub disk () t))
             (local (defun ismajority (s) (declare (ignore s)) nil))
             (defcong = equal (ismajority s) 1)

             (defthm is-majority-thm1
               (implies (and (ismajority s)
                             (ismajority s2)
                             (subsetp s (disk))
                             (subsetp s2 (disk)))
                        (not (ur-elementp (intersection s s2))))))
```

```
DiskBlock ≜ [ mbal : (UNION {Ballot(p) : p ∈ Proc }) ∪ {0},
              bal  : (UNION {Ballot(p) : p ∈ Proc }) ∪ {0},
              inp  : Inputs ∪ {NotAnInput}
            ]

hasRead(p,d,q) ≜ ∃ br ∈ blocksRead[p][d] : br.proc = q

allBlocksRead(p) ≜
   LET allRdBlks ≜ UNION { blocksRead[p][d] : d ∈ Disk }
   IN  { br.block : br ∈ allRdBlks }
```

```
(newdefmap diskblock-map1 (proc) :for p :in proc :map (ballot p))
(in-theory (disable (:executable-counterpart diskblock-map1)))

(defrec diskblock
  ("mbal" (hide (union (union* (diskblock-map1 (proc)))
                       (brace 0))))
  ("bal" (hide (union (union* (diskblock-map1 (proc)))
                      (brace 0))))
  ("inp" (hide (union (inputs) (brace (notaninput))))))

; The following construct is not explicitly defined in Disk Synod.
(defrec blockproc
  ("block" (diskblock))
  ("proc" (proc)))

(defexists exists-hasread (blocksread-p-d q)
  :exists br :in blocksread-p-d :such-that (= (apply br "proc") q))

(defun hasread (p d q blocksread)
  (exists-hasread (apply-m blocksread p d) q))

(newdefmap allblocksread-map-2 (allrdblks)
  :for br :in allrdblks :map (apply br "block"))

(newdefmap allblocksread-map-3 (disk p blocksread)
  :for d :in disk :map (apply-m blocksread p d))

(defun allblocksread (p blocksread)
  (allblocksread-map-2 (union* (allblocksread-map-3 (disk) p blocksread))))
```

```
InitDB ≜ [ mbal ↦ 0 , bal ↦ 0 , inp ↦ NotAnInput ]

InitializePhase(p) ≜
  ∧ disksWritten′ = [disksWritten EXCEPT ![p] = {}]
  ∧ blocksRead′   = [blocksRead   EXCEPT ![p] = [d ∈ Disk ↦ {}]]

StartBallot(p) ≜
   ∧ phase[p] ∈ {1,2}
   ∧ phase′ = [phase EXCEPT ![p] = 1]
   ∧ ∃ b ∈ Ballot(p) :
         ∧ b > dblock[p].mbal
         ∧ dblock′ = [dblock EXCEPT ![p].mbal = b]
   ∧ InitializePhase(p)
   ∧ UNCHANGED <input,output,disk>
```

```
(defun InitDB ()
  (func ("mbal" 0) ("bal" 0) ("inp" (NotAnInput))))

(defmap-fn map-to-nil (dom)
  :for x :in dom :map nil)

(defun _initializephase (p
                         disksWritten diskswritten-n
                         blocksRead blocksread-n)

  (and (= diskswritten-n (except disksWritten p nil))
       (= blocksread-n (except blocksRead p (map-to-nil (disk))))))

(defmacro initializephase (p)
  `(_initializephase ,p disksWritten diskswritten-n
                     blocksRead blocksread-n))

(defaction startballot (p b-witness)
  (and (mem (apply phase p) (brace 1 2))
       (= phase-n (except phase p 1))
       (mem b-witness (ballot p))
       (> b-witness (apply-m dblock p "mbal"))
       (= dblock-n (except dblock p
                           (except (apply dblock p) "mbal" b-witness)))
       (initializephase p)
       (unchanged input output disk)))
```

```
Phase1or2Write(p,d) ≜
  ∧ phase[p] ∈ {1,2}
  ∧ disk′ = [disk EXCEPT ![d][p] = dblock[p]]
  ∧ disksWritten′ = [disksWritten EXCEPT ![p] = @ ∪ {d}]
  ∧ UNCHANGED <input,output,phase,dblock,blocksRead>

Phase1or2Read(p,d,q) ≜
  ∧ d ∈ disksWritten[p]
  ∧ IF disk[d][q].mbal < dblock[p].mbal
    THEN ∧ blocksRead′ =
              [blocksRead EXCEPT ![p][d] =
               @ ∪ { [block /→ disk[d][q], proc /→ q] } ]
         ∧ UNCHANGED <input,output,disk,phase,dblock,disksWritten>
    ELSE
     StartBallot(p)
```

```
(defaction phase1or2write (p d)
  (and (mem (apply phase p) (brace 1 2))
       (= disk-n (except-m disk d p (apply dblock p)))
       (= diskswritten-n
          (except diskswritten p (union (apply diskswritten p) (brace d))))
       (unchanged input output phase dblock blocksread)))

(defaction phase1or2read (p d q b-witness)
  (and (mem d (apply diskswritten p))
       (if (< (apply-m disk d q "mbal") (apply-m dblock p "mbal"))
           (and (= blocksread-n
                   (except-m blocksread
                             p
                             d
                             (union (apply-m blocksread p d)
                                    (brace (func ("block"
                                                  (apply-m disk d q))
                                                 ("proc" q))))))
                (unchanged input output disk phase dblock diskswritten))
         (startballot p b-witness))))
```

```
Phase0Read(p,d) ≜
  ∧ phase[p] = 0
  ∧ blocksRead′ = [blocksRead EXCEPT
                     ![p][d] = @ ∪ { [ block /→ disk[d][p],
                                       proc /→ p ] } ]
  ∧ UNCHANGED <input,output,disk,phase,dblock,disksWritten>

Fail(p) ≜
   ∧ ∃ ip ∈ Inputs : input′ = [input EXCEPT ![p] = ip]
   ∧ phase′  = [phase  EXCEPT ![p] = 0 ]
   ∧ dblock′ = [dblock EXCEPT ![p] = InitDB]
   ∧ output′ = [output EXCEPT ![p] = NotAnInput]
   ∧ InitializePhase(p)
   ∧ UNCHANGED disk
```

```
(defaction phase0read (p d)
  (and (= (apply phase p) 0)
       (= blocksread-n
          (except-m blocksread
                    p
                    d
                    (union (apply-m blocksread p d)
                           (brace (func ("block" (apply-m disk d p))
                                        ("proc" p))))))
       (unchanged input output disk phase dblock diskswritten)))

(defaction fail (p ip-witness1)
  (and (mem ip-witness1 (inputs))
       (= input-n (except input p ip-witness1))
       (= phase-n  (except phase p 0))
       (= dblock-n (except dblock p (initdb)))
       (= output-n (except output p (notaninput)))
       (initializephase p)
       (unchanged disk)))
```

```
EndPhase0(p) ≜
   ∧ phase[p] = 0
   ∧ isMajority( { d ∈ Disk : hasRead(p,d,p) } )
   ∧ ∃ b ∈ Ballot(p) :
        ∧ forall r ∈ allBlocksRead(p) : b > r.mbal
        ∧ dblock′ = [ dblock EXCEPT
                        ![p] = [ (CHOOSE r ∈ allBlocksRead(p) :
                                    forall s ∈ allBlocksRead(p) :
                                      r.bal >= s.bal)
                                 EXCEPT !.mbal = b ] ]
   ∧ InitializePhase(p)
   ∧ phase′ = [ phase EXCEPT ![p] = 1 ]
   ∧ UNCHANGED <input,output,disk>
```

```
(newdefmap ep0-map1 (disk p blocksread)
  :for d :in disk
 :such-that (hasread p d p blocksread))

(defall forall-endphase0-1 (allblocksread-p b)
  :forall r :in allblocksread-p
  :holds (> b (apply r "mbal")))

(defaction endphase0 (p b-wit)
  (and (= (apply phase p) 0)
       (ismajority (ep0-map1 (disk) p blocksread))
       (mem b-wit (ballot p))
       (forall-endphase0-1 (allblocksread p blocksread) b-wit)
       (= dblock-n
          (except dblock p
                  (except (choose-max-bal (allblocksread p blocksread))
                          "mbal" b-wit)))
       (initializephase p)
       (= phase-n (except phase p 1))
       (unchanged input output disk)))
```

```
EndPhase1or2(p) ≜
   ∧ IsMajority({ d ∈ disksWritten[p] :
                  forall q ∈ Proc  {p} : hasRead(p,d,q)})
   ∧ ∨ ∧ phase[p] = 1
       ∧ dblock′ =
           [dblock EXCEPT
             ![p].bal = dblock[p].mbal,
             ![p].inp =
               LET blocksSeen ≜ allBlocksRead(p) ∪ { dblock[p] }
                   nonInitBlks ≜
                     { bs ∈ blocksSeen : bs.inp ≠ NotAnInput }
                   maxBlk ≜
                     CHOOSE b ∈ nonInitBlks :
                        forall c ∈ nonInitBlks : b.bal ≥ c.bal
               IN
                 IF nonInitBlks = {} THEN input[p]
                                     ELSE maxBlk.inp ]
       ∧ UNCHANGED output
     ∨ ∧ phase[p] = 2
       ∧ output′ = [output EXCEPT ![p] = dblock[p].inp]
       ∧ UNCHANGED dblock
   ∧ phase′ = [phase EXCEPT ![p] = @ + 1]
   ∧ InitializePhase(p)
   ∧ UNCHANGED <input, disk>
```

```
(defall forall-ep12-1 (proc-minus-p p d blocksread)
  :forall q :in proc-minus-p
  :holds (hasread p d q blocksread))

(defexists exists-ep12-1 (blocksseen)
  :exists bs :in blocksseen
  :such-that (not (= (apply bs "inp") (notaninput))))

(newdefmap map-ep12-1 (diskswritten-p p blocksread)
  :for d :in diskswritten-p
  :such-that (forall-ep12-1 (diff (proc) (brace p)) p d blocksread))

(newdefmap noninitblks (blocksseen)
  :for bs :in blocksseen
  :such-that (not (= (apply bs "inp") (notaninput))))
```

```
(defaction endphase1or2 (p)

  (and (ismajority (map-ep12-1 (apply diskswritten p) p blocksread))
       (or (and (= (apply phase p) 1)
                (= dblock-n
                   (except-and dblock
                               (p "bal" (apply-m dblock p "mbal"))
                               (p "inp" (if (=  (noninitblks
                                                 (union (allblocksread p
                                                                       blocksread)
                                                        (brace
                                                         (apply dblock p))))
                                                nil)
                                            (apply input p)
                                           (apply (choose-max-bal
                                                   (noninitblks
                                                    (union (allblocksread p
                                                                          blocksread)
                                                           (brace
                                                            (apply dblock p)))))
                                                  "inp"))))))
                (unchanged output))
            (and (= (apply phase p) 2)
                 (= output-n (except output p (apply-m dblock p "inp")))
                 (unchanged dblock)))
       (= phase-n (except phase p (+ (apply phase p) 1)))
       (initializephase p)
       (unchanged input disk)))
```

```
MajoritySet ≜ { D ∈ SUBSET Disk : IsMajority(D) }

 blocksOf(p) ≜
  LET rdBy(q,d) ≜ {br ∈ blocksRead[q][d] : br.proc = p }
  IN { dblock[p] } ∪ { disk[d][p] : d ∈ Disk }
                ∪ { br.block : br ∈ UNION {rdBy(q,d) : q ∈ Proc,
                                                        d ∈ Disk }}

 allBlocks ≜ UNION { blocksOf(p) : p ∈ Proc
```

```
(newdefmap majorityset (powset-disk)
  :for d :in powset-disk
  :such-that (ismajority d))

(newdefmap rdby (p blocksread-q-d)
  :for br :in blocksread-q-d
  :such-that (= (apply br "proc") p))

(newdefmap rdby-proc-disk-1-1 (disk-dom p q blocksread)
  :for d :in disk-dom
  :map (rdby p (apply-m blocksread q d)))

(newdefmap rdby-proc-disk-1 (proc-dom disk-dom p blocksread)
  :for q :in proc-dom
  :map (rdby-proc-disk-1-1 disk-dom p q blocksread))

(newdefmap br-block (union-rdby-proc-disk-1)
  :for br :in union-rdby-proc-disk-1
  :map (apply br "block"))

(newdefmap disk-d-p (disk-dom disk p)
  :for d :in disk-dom :map (apply-m disk d p))

(defun blocksof (p dblock disk blocksread)
  (union (apply dblock p)
         (union (disk-d-p (disk) disk p)
                (br-block (union* (rdby-proc-disk-1 (proc) (disk) p blocksread))))))

(newdefmap blocks-of-p-map (proc-dom dblock disk blocksread)
  :for p :in proc-dom
  :map (blocksof p dblock disk blocksread))

(defun allblocks (dblock disk blocksread)
  (union* (blocks-of-p-map (proc) dblock disk blocksread)))
```

```
Next ≜
   ∃ p ∈ Proc :
       ∨ StartBallot(p)
       ∨ ∃ d ∈ Disk : ∨ Phase0Read(p,d)
                      ∨ Phase1or2Write(p,d)
                      ∨ ∃ q ∈ Proc  {p} :
                          Phase1or2Read(p,d,q)
       ∨ EndPhase1or2(p)
       ∨ Fail(p)
       ∨ EndPhase0(p)
```

```
(defaction next (p              ; witness for next
                 d              ; witness for next
                 q              ; witness for next
                 b-witness1     ; witness for startballot
                 ip-witness1    ; witness for fail
                 b-witness)     ; witness for endphase0

  (and (mem p (proc))
       (or (startballot p b-witness1)
           (and (mem d (disk))
                (or (phase0read p d)
                    (phase1or2write p d)
                    (and (mem q (diff (proc) (brace p)))
                         (phase1or2read p d q b-witness1))))
           (endphase1or2 p)
           (fail p ip-witness1)
           (endphase0 p b-witness))))
```

```
HNext ≜
  ∧ Next
  ∧ chosen′ = LET hasOutput(p) ≜ output′[p] ≠ NotAnInput
              IN IF ∨ chosen ≠ NotAnInput
                    ∨ forall p in Proc : -hasOutput(p)
                 THEN chosen
                 ELSE output′[CHOOSE p in Proc : hasOutput(p)]
  ∧ allInput′ = allInput cup { input′[p] : p in Proc }
```

```
(defall forall-hnext (proc output-n)
  :forall p :in proc :holds (= (apply output-n p) (notaninput)))

(newdefmap map-hnext (proc output-n)
  :for p :in proc :such-that (not (= (apply output-n p) (notaninput))))

(newdefmap map2-hnext (proc input-n)
  :for p :in proc :map (apply input-n p))

(defun _chosen-allinput-action (chosen chosen-n
                                allinput allinput-n
                                input-n
                                output-n)
  (and (= chosen-n (if (or (not (= chosen (notaninput)))
                           (forall-hnext (proc) output-n))
                       chosen
                     (apply output-n
                            (choose (map-hnext (proc) output-n)))))
       (= allinput-n (union allinput (map2-hnext (proc) input-n)))))

(defmacro chosen-allinput-action ()
  '(_chosen-allinput-action chosen chosen-n allinput allinput-n
                            input-n output-n))

(defaction hnext-with-vars (p d q b-witness1 ip-witness1 b-witness
                            chosen chosen-n allinput allinput-n)
  (and (next p d q b-witness1 ip-witness1 b-witness)
       (chosen-allinput-action)))

(defmacro hnext (p d q b-witness1 ip-witness1 b-witness)
  `(hnext-with-vars ,p ,d ,q ,b-witness1 ,ip-witness1 ,b-witness
                    chosen chosen-n allinput allinput-n))
```

Appendix D

ACL2 Event Files

At the time of this writing, the proofs of LEMMA $I2c$ and LEMMA $I2a$ are inelegant in some places. We believe that many of the theorems included in these files can be eliminated or generalized. We now present the files including all events that lead to both lemmas in their present form.

The files are organized into three directories: the base directory plus two subdirectories `i2a` and `i2c`. The location of each file within this simple directory structure is reflected by the section name.

Proofs of the two main theorems, $I2a$ and $I2c$, can be found in files `i2a/i2a.lisp` and `i2c/i2c.lisp`.

D.1 additions.lisp

```
; additions.lisp

; Some set theory additions
; -------------------------

; I used these commands to certify this book:
; (include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
; (certify-book "additions" 1)

(in-package "S")

(include-book "/home/pacheco/acl2-sources/books/finite-set-theory/all-fns")

; ----------------------------------------------------------------
; Two theorems about mem, range and except.

(defthm mem-range-except-1
```

```
  (mem e (range (except f x e)))
  :hints (("Goal" :in-theory (enable except))))

(defthm mem-range-except-2
 (implies (and (mem e (range (except f x y)))
               (functionp f))
          (or (mem e (range f))
              (= e y)))
 :rule-classes
 ((:forward-chaining
   :trigger-terms ((mem e (range (except f x y))))))
   :hints (("Goal" :in-theory (enable except))))

; ------------------------------------------------------------
; At some point, we discussed the following two theorems. I'm not sure
; what was the conclusion and if only one was necessary. For now, I'll
; leave both in the file, since they cause no harm.

(encapsulate nil
             (local
              (defthm range-except-singleton-lemma1
                (implies (= (range f) (brace e))
                         (subsetp (range (except f x e)) (brace e)))
                :hints
                (("Goal"
                  :use (:instance subsetp-range-except
                                  (f f)
                                  (x x)
                                  (v e))
                  :in-theory (disable subsetp-range-except)))))

             (local
              (defthm range-except-singleton-lemma2
                (subsetp (brace e) (range (except f x e)))))

             (defthm range-except-singleton
               (implies (= (range f) (brace e))
                        (= (range (except f x e)) (brace e)))
               :hints (("Goal"
                        :use (range-except-singleton-lemma1
                              range-except-singleton-lemma2
                              (:instance =-iff-subsetps
                                         (a (range (except f x e)))
                                         (b (brace e))))))))
```

```
(defthm better
  (implies (and (equal (cardinality (range f)) 1)
                (= (choose (range f)) e))
           (= (range (except f x e))
              (range f))))

; ----------------------------------------------------------------
; The following are general theorems, mostly about the relationship
; among functions, their domains and ranges.


(defthm range-except-nil
  (= (range (except nil x e))
     (brace e))
  :hints (("Goal" :in-theory (enable except))))

(defthm apply-range
  (implies (and (functionp f)
                (mem e (domain f)))
           (mem (apply f e) (range f))))

(defthm setp-sfix
  (implies (setp s)
           (= (sfix s) s)))

(defthm mem-apply-2
  (implies (and (functionp f)
                (mem e (domain f))
                (subsetp (range f) s))
           (mem (apply f e) s)))

(defthm range-subsetp-except-1
  (implies (subsetp (range f) r)
           (subsetp (range (except f x y)) (scons y r)))
  :hints (("Goal" :in-theory (enable except))))

(defthm range-subsetp-except-2
  (implies (and (mem y s2)
                (subsetp (range f) s2))
           (subsetp (range (except f x y)) s2))
  :hints (("Goal" :in-theory (enable except))))

(defthm domain-subsetp-except-2
 (implies (and (= (domain f) d)
```

```
              (functionp f))
          (= (domain (except f x y)) (scons x d))))

(defthm domain-subsetp-except-1
  (implies (and (mem x s1)
                (functionp g)
                (= (domain g) s1))
           (= (domain (except g x y)) s1)))



; ------------------------------------------------------------
; Dot-dot

; (dot-dot a b) creates the set { i | i \in Ints /\  a <= i <= b }

(defun dot-dot (a b)
  (declare (xargs :measure (acl2-count (- b a))))
  (if (and (integerp a)
           (integerp b)
           (< a b))
      (scons a (dot-dot (+ a 1) b))
      (brace (nfix a))))


(defthm ur-elementp-dot-dot
  (not (ur-elementp (dot-dot a b))))

(defcong = = (dot-dot a b) 1)
(defcong = = (dot-dot a b) 2)


; After proving the right theorems about dot-dot, its definition
; should be disabled, I think. I disable it now even though I don't
; have all the right theorems about it yet, because its opening up is
; causing problems in some proofs.

(in-theory (disable dot-dot))

; We want something that looks like the following theorem

; (defthm mem-dot-dot
;  (equal (mem e (dot-dot a b))
;         (and (integerp e)
;              (<= (nfix a) e)
```

```
;              (<= e (nfix b))))
;  :hints (("Goal" :in-theory (enable =))))

; ----------------------------------------------------------------
; Union*

; (union s) takes a s = { s_1, s_2, ..., s_n } where all the s_i are
; themselves sets, and unions all the s_i's together.

(defun union* (collection-of-sets)
  (if (ur-elementp collection-of-sets)
      nil
      (union (scar collection-of-sets)
             (union* (scdr collection-of-sets)))))

(defthm setp-union*
  (setp (union* s)))

(defthm mem-union*
  (implies (and (mem e s)
                (mem s s2))
           (mem e (union* s2))))

(defx fooo (implies (and (mem sr s2)
                         (subsetp x (union* s2)))
                    (subsetp (union sr x) (union* s2)))
  :strategy subset-relation)

(congruence (union* s) 1 :method :subsetp)

; ----------------------------------------------------------------
; Defmap-fn

; defmap-fn is similar to defmap using the :map argument. Consider one
; such call of defmap:

; (defmap foo-set (s) :for x :in s :map (foo x))

; The function (foo-set s) builds a set consisting of all (foo x) such
; that x is in s, i.e. the mapping of s under foo. Now consider the
; analogous call of defmap-fn:

; (defmap-fn foo-fn (s) :for x :in s :map (foo x))

; The function (foo-fn s) creates the FUNCTION (set of ordered pairs)
```

```
; with domain s and range the mapping of s under foo. In other words, it
; creates the function with pairs < x , (foo x) >, where x is in s.

(defmacro defmap-fn (name vars
                          &key
                          (for 'nil forp)
                          (in 'nil inp)
                          (map 'nil mapp))


  (cond
   ((not (and (symbolp name)
              (acl2::symbol-listp vars)
              forp
              (symbolp for)
              (not (acl2::member-equal for vars))
              inp
              (symbolp in)
              (acl2::member-equal in vars)
              mapp))
    `(acl2::er acl2::soft 'defmap
               "No documentation yet. Sorry!"))
   (t ;;; :map
    (let* ((x for)
           (s in)
           (sloc (- (length vars) (length (member s vars))))
           (body map)
           (fx (genname1 x 1 (cons x vars)))
           (s1 (genname1 s 1 (cons fx (cons x vars))))
           (call `(,name ,@vars))
           (rcall `(,name ,@(put-nth `(scdr ,s) sloc vars))))
      `(encapsulate
        nil
        (defun ,name (,@vars)
          (if (ur-elementp ,s)
              nil
            (let ((,x (scar ,s)))
              (except ,rcall ,x ,body))))

        (defthm ,(packn-in-pkg (list "SETP-" name) 'defmap)
          (setp ,call))

        (defthm ,(packn-in-pkg (list "UR-ELEMENTP-" name) 'defmap)
          (equal (ur-elementp ,call)
                 (ur-elementp ,s)))
```

```
        (defthm ,(packn-in-pkg (list "WEAK-MEM-" name) 'defmap)
          (implies (and (mem ,x ,s)
                        (= ,fx ,body))
                   (mem (pair ,x ,fx) ,call)))


        (defthm ,(packn-in-pkg (list "SUBSETP-" name) 'defmap)
          (implies (subsetp ,s1 ,s)
                   (subsetp (,name ,@(put-nth s1 sloc vars))
                            ,call)))

        ,@(defmap-congruences vars call (+ sloc 1) 1)

        (defthm ,(packn-in-pkg (list "FUNCTIONP-" name) 'defmap)
          (functionp (,name ,@vars)))

        (defthm ,(packn-in-pkg (list "DOMAIN-" name) 'defmap)
          (= (domain (,name ,@vars))
             (sfix ,s)))

        )))))


; ---------------------------------------------------------------------------
; Rules about all-fns

; This rule subsumed by J's. I keep it just for hints that may use its
; name.

(defthm all-fns-def
   (iff (mem f (all-fns d r))
        (and (functionp f)
             (= (domain f) (sfix d))
             (subsetp (range f) r))))

(defthm all-fns-except
  (implies (and (mem f (all-fns d r))
                (mem x d)
                (mem y r))
           (mem (except f x y) (all-fns d r))))

(defthm  all-fns-apply
  (implies (and (subsetp (range f) (all-fns d r))
```

```
               (mem x (domain f))
               (functionp f))
          (mem (apply f x) (all-fns d r))))

(defthm  all-fns-apply-2
  (implies (and (mem f (all-fns d r))
               (mem x (domain f)))
          (mem (apply f x) r)))

(defthm  all-fns-apply-3
  (implies (and (mem f (all-fns d r))
               (mem x d))
          (mem (apply f x) r)))

; In reality all-fns-apply-apply should be covered by all-fns-apply-3. But it
; helps with type invariance proofs because we look explicitly for
; things of the form [ a -> [ b -> c ] ] in the hypotheses.

; ugly but i'm trying to write a thesis here...
(defthm all-fns-apply-apply
  (implies (and (mem f (all-fns a (all-fns b c)))
               (mem x a)
               (mem y b))
          (mem (apply (apply f x) y) c))
  :instructions (promote
                 (:REWRITE all-fns-APPLY-3 ((D B)))
                 rewrite))

; The rule all-fns-def should be disabled in order to reason at a higher
; level.
(in-theory (disable all-fns-def))

; ----------------------------------------------------------------------

(defthm union-nil
  (implies (setp s) (= (union s nil) s)))

(defthm subsetp-scons-2
  (implies (and (subsetp a b)
               (mem e b))
          (subsetp (scons e a) b)))

; this worked wonders! how exactly does forward chaining work?
(defthm all-fns-property-forward-chain
  (implies (mem f (all-fns d r))
```

```
           (and ; (not (ur-elementp (all-fns d r)))
                (functionp f)
                (= (domain f) (sfix d))
                (subsetp (range f) r)))
  :rule-classes ((:forward-chaining))
  :hints (("Goal" :in-theory (enable all-fns-def))))

; should i add something like this? its setp hypothesis reminds me of
; powerset-property.

; (defthm not-ur-elementp-s-iff-s
;   (implies (setp s) (iff (not (ur-elementp s)) s)))

; This rule's target is too general. You might find application of the
; rule by specializing it depending on the TLA spec (especially by
; looking at the type invariant).

; (defthm subsetp-apply
;   (implies (and (functionp f)
;                 (mem x (domain f))
;                 (subsetp (range f) (powerset s)))
;            (subsetp (apply f x) s))
;   :hints (("Goal" :use ((:instance mem-apply-2
;                                    (e x)
;                                    (s (powerset s)))
;                         (:instance weak-powerset-property
;                                    (e (apply f x))))))
;   :rule-classes nil)


; End of file ------------------------------------------------------------------
```

D.2 choose-max.lisp

```
; choose-max.lisp

; I used these commands to certify this book:
; (include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
; (certify-book "choose-max" 1)

(in-package "S")
(include-book "newdefmap")
```

```
(include-book "defall")

(encapsulate
 ((g (x) t))

 (local (defun g (x) (declare (ignore x)) t))
 (defcong = = (g x) 1))

; Here is a definition of a function that returns one of possibly several
; maximal elements in a set (as ordered by >=).

(defun choose-max0 (s)
    (cond ((ur-elementp s) nil)
        ((ur-elementp (scdr s)) (scar s))
        ((>= (g (scar s))
             (g (choose-max0 (scdr s))))
         (scar s))
        (t (choose-max0 (scdr s)))))

; One important fact is that every nonempty set has a maximal element.

(defthm choose-max0-exists
  (implies (not (ur-elementp s))
           (mem (choose-max0 s) s)))

; Another key fact about the maximal element.

(defthm choose-max0->=
  (implies (mem x s)
           (>= (g (choose-max0 s)) (g x))))

; Unfortunately, choose-max doesn't respect the following congruence:

; (defcong = = (choose-max s) 1).

; The reason is that (choose-max s) picks the first maximal element it
; finds in the list representation of the set s. If two equal sets s1
; and s2 having more than one maximal element happen to have their
; maximal elements presented in different order, then the value
; returned by (choose-max s1) and (choose-max s2) could be different.

; Here is a different version that does respect set-equivalence.

; This predicate checks that y is a maximal element with respect to the set s.
(defall forall-x-is-greater0 (y s)
```

```
  :forall x :in s
  :holds (>= (g y) (g x)))

; Here we collect all elements from s1 that are maximal with respect
; to s2. That is, we create the set of all x in s1 s.t. x>=y for all y
; \in s2.

(newdefmap collect-maximal-elements0 (s1 s2)
  :for x :in s1
  :such-that (forall-x-is-greater0 x s2))

; Finally, choose-max1 collects all the maximal elements of set s, and
; chooses one. Since (choose x) always picks the same element for a
; given set, it choose-max enjoys the congruence property that
; choose-max1 doesn't.

(defun choose-max1 (s)
  (choose (collect-maximal-elements0 s s)))

(defcong = = (choose-max1 s) 1)

; We would also like to show that it enjoys the two other important
; properties that follow immediately for choose-max-0, namely
; choose-max-0-exists and choose-max-0->=. They don't follow
; immediately.

; This encapsulate is clunky. As an exercise, I should elegantize the
; proofs.
(encapsulate
 nil

 (local
  (progn
    (in-theory (enable forall-x-is-greater0-predicate))
    (in-theory (enable forall-x-is-greater0))

    (defthm forall-x-is-greater-scons-2
      (implies (and (forall-x-is-greater0 x s)
                    (>= (g y) (g x)))
               (forall-x-is-greater0  y (scons x s))))

    (defthm choose-max-0-forall-x-is-greater
      (implies (not (ur-elementp s))
               (forall-x-is-greater0 (choose-max0 s) s))
      :hints (("Goal" :induct (choose-max0 s))))
```

```
    (defthm choose-max-0-mem-collect-maximal-elements
      (implies (not (ur-elementp s))
               (mem (choose-max0 s) (collect-maximal-elements0 s s))))

    (defthm collect-maximal-elements-not-empty
      (implies (not (ur-elementp s))
               (not (ur-elementp (collect-maximal-elements0 s s))))
      :hints (("Goal" :use
               (:instance choose-max-0-mem-collect-maximal-elements))))

    ; there's a smarter way to prove this theorem. do it.
    (defthm mem-choose-collect-maximal-elements-dumb
      (implies (not (ur-elementp (collect-maximal-elements0 s s)))
               (mem (choose (collect-maximal-elements0 s s)) s))
      :hints (("Goal" :use ((:instance mem-collect-maximal-elements0
                                       (x (choose (collect-maximal-elements0 s s)))
                                       (s1 s)
                                       (s2 s))
                            (:instance mem-choose
                                       (a (collect-maximal-elements0 s s))))
               :in-theory nil)))
    )) ; end local progn

; This is the only theorem we export from the encapsulate.

    (defthm choose-max-exists1
      (implies (not (ur-elementp s))
               (mem (choose-max1 s) s)))
)

; Here I use functional instantiation to prove the previous theorem
; for the version of choose-max1 used in the Disk Paxos algorithm.

(defall forall-x-is-greater-bal (y s)
  :forall x :in s
  :holds (>= (apply y "bal") (apply x "bal")))

(newdefmap collect-maximal-elements-bal (s1 s2)
  :for x :in s1
  :such-that (forall-x-is-greater-bal x s2))

(defun choose-max-bal (s)
  (choose (collect-maximal-elements-bal s s)))
```

```
(defthm very-silly
  (EQUAL (COLLECT-MAXIMAL-ELEMENTS-BAL S1 S2)
         (AND (NOT (UR-ELEMENTP S1))
              (LET ((X (SCAR S1)))
                   (IF (FORALL-X-IS-GREATER-BAL X S2)
                       (SCONS (SCAR S1)
                              (COLLECT-MAXIMAL-ELEMENTS-BAL (SCDR S1)
                                                            S2))
                       (COLLECT-MAXIMAL-ELEMENTS-BAL (SCDR S1)
                                                     S2)))))
  :hints (("Goal" :in-theory (enable collect-maximal-elements-bal)))
  :rule-classes nil)

(in-theory (enable forall-x-is-greater-bal
                   collect-maximal-elements-bal
                   forall-x-is-greater-bal-predicate))


(defcong = = (choose-max-bal s) 1
  :hints (("Goal" :by (:functional-instance
                       =-implies-=-choose-max1-1
                       (choose-max1 choose-max-bal)
                       (g (lambda (x) (apply x "bal")))
                       (forall-x-is-greater0 forall-x-is-greater-bal)
                       (collect-maximal-elements0 collect-maximal-elements-bal)
                       (forall-x-is-greater0-predicate
                        forall-x-is-greater-bal-predicate)))))

(defthm choose-max-bal-exists
  (implies (not (ur-elementp s))
           (mem (choose-max-bal s) s))
  :hints (("Goal" :by (:functional-instance
                       choose-max-exists1
                       (choose-max1 choose-max-bal)
                       (g (lambda (x) (apply x "bal")))
                       (forall-x-is-greater0 forall-x-is-greater-bal)
                       (collect-maximal-elements0 collect-maximal-elements-bal)
                       (forall-x-is-greater0-predicate
                        forall-x-is-greater-bal-predicate)))))

(in-theory (disable choose-max-bal))

; end file ------------------------------------------------------------------
```

D.3 common-all.lisp

```
; common-all.lisp

; I used the following command to certify this book:
; (ld "defpkg.lisp")
; (certify-book "common-all" 1)

(in-package "S")

(include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
(include-book "translations")

(defmacro enable-all-actions ()
'(in-theory (enable startballot phase0read phase1or2write
                    phase1or2read endphase1or2 fail endphase0
                    chosen-allinput-action
                    next hnext initializephase)))

(defmacro disable-all-actions ()
'(in-theory (disable startballot phase0read phase1or2write
                     phase1or2read endphase1or2 fail endphase0
                     chosen-allinput-action
                     next hnext initializephase)))

; Default: all actions are disabled.
(disable-all-actions)

(in-theory (disable hasread))
; Proc's definition doesn't help in proving type invariance, so I
; disable it.
(in-theory (disable proc))

; Found this rule expensive through accumulated-persistence. I don't
; seem to need it.
(in-theory (disable subsetp-not-subsetp-trick))

; More rules that might be good to disable:
; (in-theory (disable mem-container))
; (in-theory (disable subsetp))
; (in-theory (disable mem-subsetp))
; (in-theory (disable mem))
; (in-theory (disable powerset-property))
; (in-theory (disable apply-outside-domain))
```

```
; end file ------------------------------------------------------------------
```

D.4 defall.lisp

```
; defall.lisp

; I used these commands to certify this book:
; (include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
; (certify-book "defall" 1)

(in-package "S")

; Universal Quantification
; ------------------------

; Defall is my way of expressing universally quantified statements in
; a finite set theory context. Consider a call to defall

; (defall all-satisfy-q (a s b) :forall x :in s :holds (q a b x))

; where (q a b) is a previously-defined predicate.

; [Explanation to be continued...]

; I comment each piece of the macro with the translation generated by
; the following code:


(defun defall-predicate-congruences (vars call i)
  (cond
   ((endp vars) nil)
   (t (cons `(defcong = equal ,call ,i)
            (defall-predicate-congruences (cdr vars) call (+ 1 i))))))

(defun defall-pred-set-congruences (vars call sloc i)
  (cond
   ((endp vars) nil)
   (t (cons (if (equal sloc i)
                `(defx  :strategy :congruence ,call ,i :method :canonicalize)
              `(defcong = equal ,call ,i))
            (defall-pred-set-congruences (cdr vars) call sloc (+ 1 i))))))
```

```
; Problem to fix with this defall: some theorems use variables such as
; e, a, b, c, but I don't check to make sure that there variables are
; not among those in `vars'. So for instance, if in some theorem I
; want to express set membership within a defall named foo with
; arguments (d e f), and I happen to say

; (mem e ,(pred-set-call ,@(pred-set-vars))), it will become

; (mem e (foo d e f)), which is not what I mean.



(defmacro defall (name
                  vars
                  &key
                  (forall 'nil forallp)
                  (in 'nil inp)
                  (holds 'nil holdsp))
  (cond
   ((not (and (symbolp name)
              (acl2::symbol-listp vars)
              forallp
              (symbolp forall)
              (not (acl2::member-equal forall vars))
              inp
              (symbolp in)
              (acl2::member-equal in vars)
              holdsp))
              `(acl2::er acl2::soft 'defquant
                         "Not documented."))
   (t
    (let* ((x forall)
           (s in)
           (sloc (- (length vars) (length (member s vars))))
           (pred-name (packn-in-pkg (list name "-PREDICATE") 'defall))
           (pred-set-name name)
           (pred-vars (substitute x s vars))
           (pred-set-vars vars)
           (pred-call `(,pred-name ,@pred-vars))
           (pred-set-call `(,pred-set-name ,@pred-set-vars))
           (rcall `(,pred-set-name ,@(substitute `(scdr ,s) s vars))))
      `(encapsulate
        nil
```

```
; We will consider the following call to defall:

; (DEFALL FORALL-P-HOLDS (A S B) :FORALL X :IN S :HOLDS (P A X B))

; We need to declare a function expressing the :holds
; condition in order to place the condition inside a black
; box. This enables the proofs below to go through with no
; problems. The :holds condition should have nothing to do
; with the success of a defall declaration. However, if we do
; not hide its details, it can interfere with a successful
; proof.

; (DEFUN FORALL-P-HOLDS-PREDICATE (A X B) (P A X B))

(defun ,pred-name (,@pred-vars)
  ,holds)

; (DEFCONG = EQUAL (FORALL-P-HOLDS-PREDICATE A X B) 1)
; (DEFCONG = EQUAL (FORALL-P-HOLDS-PREDICATE A X B) 2)
; (DEFCONG = EQUAL (FORALL-P-HOLDS-PREDICATE A X B) 3)

,@(defall-predicate-congruences pred-vars pred-call 1)

; Having proven the necessary congruences about pred-name, I
; disable it so it doesn't interfere with the rest of the
; proofs.

; (IN-THEORY (DISABLE FORALL-P-HOLDS-PREDICATE))

(in-theory (disable ,pred-name))

; Now we define the actual function that will be used outside
; the macro, with name equal to the one the user supplied.

; (DEFUN FORALL-P-HOLDS (A S B)
;   (IF (UR-ELEMENTP S)
;       T
;       (IF (FORALL-P-HOLDS-PREDICATE A (SCAR S) B)
;           (FORALL-P-HOLDS A (SCDR S) B)
;           NIL)))

(defun ,pred-set-name (,@pred-set-vars)
  (if (ur-elementp ,s)
      t
      (if (,pred-name ,@(substitute `(scar ,s) s vars))
```

```
          ,rcall
          nil)))


; (DEFCONG = EQUAL (FORALL-P-HOLDS A S B) 1)
; (DEFX :STRATEGY :CONGRUENCE (FORALL-P-HOLDS A S B) 2
;       :METHOD :CANONICALIZE)
; (DEFCONG = EQUAL (FORALL-P-HOLDS A S B) 3)

,@(defall-pred-set-congruences vars pred-set-call (+ sloc 1) 1)

; trying this one 1.27.01

(defthm ,(packn-in-pkg (list pred-set-name "-MEM") 'defall)
  (implies (and (,pred-set-name ,@pred-set-vars)
                (mem ,x ,s))
           (,pred-name ,@pred-vars)))

; The next two theorems, FORALL-P-HOLDS-APPLY and
; FORALL-P-HOLDS-PREDICATE-IMPLIES, are used in the proof of
; FORALL-P-HOLDS-APPLY-2. While I'm certain I don't want to
; have FORALL-P-HOLDS-PREDICATE-IMPLIES around (and thus
; declare it locally), I haven't made up my mind yet about
; FORALL-P-HOLDS-APPLY yet. So I export it.

; (DEFTHM FORALL-P-HOLDS-APPLY
;   (IMPLIES (AND (FORALL-P-HOLDS A (RANGE S) B)
;                 (FUNCTIONP S)
;                 (MEM E (DOMAIN S)))
;            (FORALL-P-HOLDS-PREDICATE A (APPLY S E) B)))

(defthm ,(packn-in-pkg (list pred-set-name "-APPLY") 'defall)
  (implies (and (,pred-set-name ,@(substitute `(range ,s) s vars))
                (functionp ,s)
                (mem e (domain ,s)))
           (,pred-name ,@(substitute `(apply ,s e) s vars))))

; Even though the `implies' below could be substituted by
; `equal' which is stronger, the theorem is proved much faster
; with `implies'. Since I only use this theorem as an
; auxiliary one in a proof below, I don't care about strength
; but only speed.

; (LOCAL (DEFTHM FORALL-P-HOLDS-PREDICATE-IMPLIES
;          (IMPLIES (FORALL-P-HOLDS-PREDICATE A X B)
```

```
;                    (P A X B))
;          :HINTS
;          (("Goal" :IN-THEORY
;            (ENABLE FORALL-P-HOLDS-PREDICATE)))
;          :RULE-CLASSES NIL))

(local
 (defthm ,(packn-in-pkg (list pred-name "-IMPLIES") 'defall)
   (implies (,pred-name ,@pred-vars)
            ,holds)
   :hints (("Goal" :in-theory (enable ,pred-name)))
   :rule-classes nil))

; (DEFTHM FORALL-P-HOLDS-APPLY-2
;   (IMPLIES (AND (FORALL-P-HOLDS A (RANGE S) B)
;                 (FUNCTIONP S)
;                 (MEM E (DOMAIN S)))
;            (P A (APPLY S E) B))
;   :HINTS
;   (("Goal" :USE
;     ((:INSTANCE FORALL-P-HOLDS-APPLY)
;      (:INSTANCE FORALL-P-HOLDS-PREDICATE-IMPLIES
;                 (X (APPLY S E))))
;     :IN-THEORY NIL)))

(defthm ,(packn-in-pkg (list pred-set-name "-APPLY-2") 'defall)
  (implies (and (,pred-set-name ,@(substitute `(range ,s) s vars))
                (functionp ,s)
                (mem e (domain ,s)))
           ,(subst `(apply ,s e) forall holds))
  :hints (("Goal" :use ((:instance
                         ,(packn-in-pkg (list pred-set-name "-APPLY")
                                        'defall))
                        (:instance
                         ,(packn-in-pkg (list pred-name "-IMPLIES")
                                        'defall)
                         (,forall (apply ,s e))))
           :in-theory nil)))

; (DEFTHM FORALL-P-HOLDS-RANGE-EXCEPT
;   (IMPLIES (AND (FORALL-P-HOLDS A (RANGE S) B)
;                 (FORALL-P-HOLDS-PREDICATE A Y B))
;            (FORALL-P-HOLDS A (RANGE (EXCEPT S X Y))
;                            B))
;   :HINTS
```

```
;   (("Goal" :IN-THEORY (ENABLE EXCEPT))))

(defthm ,(packn-in-pkg (list pred-set-name "-RANGE-EXCEPT") 'defall)
  (implies (and (,pred-set-name ,@(substitute `(range ,s) s vars))
                (,pred-name ,@(substitute 'y x pred-vars)))
           (,pred-set-name ,@(substitute `(range (except ,s x y))
                                         s vars)))
  :hints (("Goal" :in-theory (enable except))))

; I had problems with some rewrite rules that converted terms
; into if-expressions. This is my current solution.

; (DEFTHM FORALL-P-HOLDS-IF-BREAK-UP
;   (EQUAL (FORALL-P-HOLDS A (IF A1 B1 C1) B)
;          (IF A1 (FORALL-P-HOLDS A B1 B)
;              (FORALL-P-HOLDS A C1 B))))

(defthm ,(packn-in-pkg (list pred-set-name "-IF-BREAK-UP") 'defall)
  (equal (,pred-set-name ,@(substitute `(if a1 b1 c1) x pred-vars))
         (if a1
             (,pred-set-name ,@(substitute 'b1 x pred-vars))
           (,pred-set-name ,@(substitute 'c1 x pred-vars)))))

; (DEFTHM FORALL-P-HOLDS-CHOOSE
;   (IMPLIES (AND (NOT (UR-ELEMENTP S))
;                 (FORALL-P-HOLDS A S B))
;            (FORALL-P-HOLDS-PREDICATE A (CHOOSE S)
;                                      B))
;   :HINTS
;   (("Subgoal *1/3" :CASES ((= (CHOOSE S) (SCAR S))))))

(defthm ,(packn-in-pkg (list pred-set-name "-CHOOSE") 'defall)
  (implies (and (not (ur-elementp ,s))
                (,pred-set-name ,@pred-set-vars))
           (,pred-name ,@(substitute `(choose ,s) x pred-vars)))
  :hints (("Subgoal *1/3" :cases ((= (choose ,s) (scar ,s))))))

; (DEFTHM FORALL-P-HOLDS-UNION
;   (IMPLIES (AND (FORALL-P-HOLDS A S B)
;                 (FORALL-P-HOLDS A S1 B))
;            (FORALL-P-HOLDS A (UNION S S1) B)))

(defthm ,(packn-in-pkg (list pred-set-name "-UNION") 'defall)
  (implies (and (,pred-set-name ,@pred-set-vars)
                (,pred-set-name ,@(substitute 's1 s pred-set-vars)))
```

```
                (,pred-set-name
                 ,@(substitute `(union ,s s1) s pred-set-vars))))


        ; (DEFTHM FORALL-P-HOLDS-INTERSECTION
        ;   (IMPLIES (AND (FORALL-P-HOLDS A S B)
        ;                 (FORALL-P-HOLDS A S1 B))
        ;            (FORALL-P-HOLDS A (INTERSECTION S S1)
        ;                            B)))

        (defthm ,(packn-in-pkg (list pred-set-name "-INTERSECTION") 'defall)
          (implies (and (,pred-set-name ,@pred-set-vars)
                        (,pred-set-name ,@(substitute 's1 s pred-set-vars)))
                   (,pred-set-name
                    ,@(substitute `(intersection ,s s1) s pred-set-vars))))

        ; (DEFTHM FORALL-P-HOLDS-SCONS
        ;   (IMPLIES (AND (FORALL-P-HOLDS-PREDICATE A X B)
        ;                 (FORALL-P-HOLDS A S B))
        ;            (FORALL-P-HOLDS A (SCONS X S) B)))

        (defthm ,(packn-in-pkg (list pred-set-name "-SCONS") 'defall)
          (implies (and (,pred-name ,@pred-vars  )
                        (,pred-set-name ,@pred-set-vars))
                   (,pred-set-name ,@(substitute `(scons ,x ,s) s vars))))

        ; (DEFTHM FORALL-P-HOLDS-NIL
        ;   (FORALL-P-HOLDS A NIL B))

        (defthm ,(packn-in-pkg (list pred-set-name "-NIL") 'defall)
          (,pred-set-name ,@(substitute nil s vars)))

        ; Finally, we disable both functions. Currently, I hold the
        ; view that these functions should never be opened; we should
        ; only have the needed theorems to reason about them. This is
        ; most true of the recursive function. (I think) opening
        ; recursive functions can be expensive, and disabling all my
        ; recursive functions defined through defmap and defall made a
        ; huge impact in performance.

        (in-theory (disable ,pred-set-name))
        (in-theory (enable ,pred-name)))))))

; ------------------------------------------------------------------------
```

```
; Instantiatable theory -- plug and play!

; proving ‘:forall: x :in: s : p(x)’ statements
; using ‘x :in: s => p(x)’

#|

(encapsulate
 ((p (x) t)
  (hyps () t)
  (s () t))
 (local (defun p (x) (declare (ignore x)) t))
 (local (defun hyps () nil))
 (local (defun s () t))
 (defcong = equal (p x) 1)
 (defthm mem-foo (implies (and (hyps)
                               (mem x (s)))
                          (p x))))

(defall foo (s)
   :forall x :in s :holds (p x))

; ; ; once mem-foo theorem is proved, a macro creates:

(defthm helper2-foo (implies
                     (and (hyps)
                          (subsetp s1 (s)))
                     (foo s1))
  :hints (("Goal" :in-theory (enable foo))))

(defthm main-foo (implies (hyps)
                          (foo (s))))

|#

; End of file ------------------------------------------------------------------
```

D.5 defexists.lisp

```
; defexists.lisp

; I used these commands to certify this book:
```

```
; (include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
; (certify-book "defexists" 1)

(in-package "S")

(include-book "newdefmap")

; Existential Quantification
; --------------------------

; My first thought was to express existential quantification simply by
; writing

; (foo-map s)

; where

; (defmap foo-map (s) :for x :in s :such-that p)

; If there is some element in s with property p, then (foo-map s) will
; be a nonempty set. If there is no element in s satisfying p, then
; (foo-map s) will return nil. Thus (foo-map s) is logically
; equivalent (i.e. is nil/non-nil in unison) to existential
; quantification.

; The motivation behind defexists is my desire to have a place where I
; can think about existential reasoning without interfering with other
; kinds of constructs that I might define using defmap and which might
; have nothing to do with existential reasoning. There are theorems
; that are useful for set constructors that use defmap but are not
; useful for existential quantification statements, and visceversa.
; It also seems more satisfying to express existential quantification
; using a boolean construct.

; However, I use an abbreviated version of defmap, including the same
; recursive definition that defmap defines, inside defexists. I do
; this because a theorem I want to prove the following theorem within
; defexists:

; (equal (exists-x-such-that-p s)
;          (and (mem (witness s) s)
;               (p (witness s))

; where (defexists exists-x-such-that-p (s) :exists x :in s
;                                           :such-that (p x))
```

```
; (Witness s) is easily defined as follows:

; (defmap collect-elements-such-that-p (s) :for x :in s
;                                           :such-that (p x))

; (defun witness (s) (choose (collect-elements-such-that-p s)))

; Since I don't care to export collect-elements-such-that-p, I define
; it locally and prove some theorems about it, directly lifted from
; the defmap macro.

(defun defexists-congruences (vars call i)
  (cond
   ((endp vars) nil)
   (t (cons `(defcong = equal ,call ,i)
            (defexists-congruences (cdr vars) call (+ 1 i))))))

(defmacro defexists (name vars
                          &key
                          (exists 'nil existsp)
                          (in 'nil inp)
                          (such-that 'nil such-thatp)

                          ; I'll have to check if I need the rule that
                          ; led to this keyword in the proofs for
                          ; defexists. If not, the keyword can go away.
                          (mem-corollary 't))

  (cond
   ((not (and (symbolp name)
              (acl2::symbol-listp vars)
              existsp
              (symbolp exists)
              (not (acl2::member-equal exists vars))
              inp
              (symbolp in)
              (acl2::member-equal in vars)
              such-thatp))
    `(acl2::er acl2::soft 'defmap "Not documented."))
   (t (let* ((x exists)
             (s in)
             (sloc (- (length vars) (length (member s vars))))
             (body such-that)
             (s1 (genname1 s 1 (cons x vars)))
```

```
;            (call `(,name ,@vars))
             (pred-name (packn-in-pkg (list name "-PRED") 'newdefmap))
             (name-local (packn-in-pkg (list name "-LOCAL") 'newdefmap))
             (name-map (packn-in-pkg (list name "-MAP") 'newdefmap))
             (rcall `(,name-map ,@(put-nth `(scdr ,s) sloc vars)))
             (call-local `(,name-local ,@vars))
             (pred-call `(,pred-name ,@(substitute x s vars))))


 `(encapsulate
   nil
    (local
     (progn
       (defun ,pred-name
         ,(substitute x s vars)
         ,body)

       ,@(newdefmap-pred-congruences (substitute x s vars) pred-call 1)

       (in-theory (disable ,pred-name))

                                       ; We define name-local in terms of pred-name.

       (defun ,name-local (,@vars)
         (if (ur-elementp ,s)
             nil
           (let ((,x (scar ,s)))
             (if ,pred-call
                 (scons (scar ,s) (,name-local ,@(put-nth `(scdr ,s)
                                                          sloc vars)))
               (,name-local ,@(put-nth `(scdr ,s) sloc vars))))))

       (defthm ,(packn-in-pkg (list "SETP-" name-local) 'newdefmap)
         (setp (,name-local ,@vars)))

       (defthm ,(packn-in-pkg (list "UR-ELEMENTP-" name-local) 'newdefmap)
         (equal (ur-elementp ,call-local)
                (equal ,call-local nil)))


       (defthm ,(packn-in-pkg (list "MEM-" name-local) 'newdefmap)
         (equal (mem ,x ,call-local)
                (and ,pred-call         ; we write it this way in case body
                     (mem ,x ,s)))      ; is not Boolean!
         :otf-flg t)
```

```
    (defthm ,(packn-in-pkg (list "SUBSETP-" name-local) 'newdefmap)
      (subsetp ,call-local ,s))

    ,@(newdefmap-local-congruences vars call-local (+ sloc 1) 1)

    ,@(if mem-corollary
          `((defthm ,(packn-in-pkg (list "MEM-" name-local "-CORROLLARY")
                                   'newdefmap)
              (implies (and (subsetp ,s1 ,call-local)
                            (mem ,x ,s1))
                       ,pred-call)))
        nil)

    (defthm ,(packn-in-pkg (list "CARDINALITY-" name-local) 'newdefmap)
      (<= (cardinality ,call-local)
          (cardinality ,s))
      :rule-classes :linear)))

(defun ,(packn-in-pkg (list name "-MAP") 'defexists) (,@vars)
         (if (ur-elementp ,s)
             nil
           (let ((,x (scar ,s)))
             (if ,body
                 (scons (scar ,s) ,rcall)
               ,rcall))))

; Here is the definition that expresses existential quantification.
(defun ,name ,vars
         (not (ur-elementp
               (,(packn-in-pkg (list name "-MAP") 'defexists)
                ,@vars))))

; Now, we show that foo-local and foo-map are equal. Thus, any
; theorems that held for foo-local trivially hold for foo.
(local
 (defthm ,(packn-in-pkg (list name-map "-EQUALS-" name-local) 'newdefmap)
   (equal (,name-map ,@vars) (,name-local ,@vars))
   :hints (("Goal" :in-theory (enable ,pred-name)))))

; We also show that pred-call and body are equal.
(local
 (defthm PRED-EQUALS-BODY
   (equal ,pred-call ,body)
   :hints (("Goal" :in-theory (enable ,pred-name)))))
```

```
    (defthm ,(packn-in-pkg (list name "-IS-BOOLEAN") 'defexists)
      (booleanp (,name ,@vars))
      :rule-classes :type-prescription)

    ,@(defexists-congruences vars `(,name ,@vars) 1)

    (local (in-theory (disable
                       PRED-EQUALS-BODY)))

    (in-theory (disable ,(packn-in-pkg (list name "-MAP") 'defexists)
                        ,name))

    ; We prove MEM-CHOOSE using <name>-LOCAL first.
;       (defthm ,(packn-in-pkg (list name "-MEM-CHOOSE") 'defexists)
;         (iff (,name ,@vars)
;            (and (mem (choose (,name-map ,@vars))
;                      ,s)
;                 ,(subst `(choose (,name-map ,@vars)) x body))))

    )))))

; End of file ------------------------------------------------------------------
```

D.6 defpkg.lisp

```
; defpkg.lisp

(defpkg "S"
  (set-difference-equal
   (union-eq '(PACK
               ORDINARYP
               <<
               <<-IRREFLEXIVITY
               <<-TRICHOTOMY
               <<-MUTUAL-EXCLUSION
               <<-TRANSITIVITY
               FAST-<<-TRICHOTOMY
               FAST-<<-MUTUAL-EXCLUSION
               FAST-<<-TRANSITIVITY
               FAST-<<-RULES
```

```
              SLOW-<<-RULES
              <<-RULES)
             (union-eq *acl2-exports*
                       *common-lisp-symbols-from-main-lisp-package*))
   '(union intersection subsetp add-to-set functionp = apply)))

; end file ------------------------------------------------------------------
```

D.7 hinv1.lisp

```
; hinv1.lisp

; I used the following command to certify this book:
; (ld "defpkg.lisp")
; (certify-book "hinv1" 1)

(in-package "S")

(include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
(include-book "translations")

;----------------------------------------------------------------------
;
; HInv1 ==
;
;  /\ input        in [Proc -> Inputs]
;  /\ output       in [Proc -> Inputs cup { NotAnInput } ]
;  /\ disk         in [Disk -> [Proc -> DiskBlock]]
;  /\ phase        in [Proc -> 0..3]
;  /\ dblock       in [Proc -> DiskBlock]
;  /\ disksWritten in [Proc -> SUBSET Disk]
;  /\ blocksRead   in [Proc -> [Disk ->
;                               SUBSET[block : DiskBlock, proc : Proc]]]
;  /\ allInput     in SUBSET Inputs
;  /\ chosen       in Inputs cup {NotAnInput}
;
;----------------------------------------------------------------------

(defstate hinv1-with-vars (allinput chosen)
  (and (mem input        (all-fns (proc) (inputs)))
       (mem output       (all-fns (proc)
                                  (hide (union (inputs) (brace (notaninput))))))
```

```
       (mem disk         (all-fns (disk)
                                  (all-fns (proc) (diskblock))))
       (mem phase        (all-fns (proc) (nats 3)))
       (mem dblock       (all-fns (proc) (diskblock)))
       (mem diskswritten (all-fns (proc) (powerset (disk))))
       (mem blocksread   (all-fns (proc)
                                  (all-fns (disk) (powerset (blockproc)))))
       (mem allinput     (powerset (inputs)))
       (mem chosen       (hide (union (inputs) (brace (notaninput)))))))

(defmacro hinv1 ()
  '(hinv1-with-vars allinput chosen))

(add-macro-alias hinv1 _hinv1-with-vars)

; End of file ------------------------------------------------------------------
```

D.8 hinv2-exports.lisp

```
; hinv2-exports.lisp

(in-package "S")

; I used the following command to certify this book:
; (ld "defpkg.lisp")
; (certify-book "hinv2-exports" 1)

(include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
(include-book "hinv2")

(local
 (progn

(defthm
  forall-hinv2-2-mem-sane-version
  (implies
   (and (forall-hinv2-2 proc diskswritten
                        phase disk dblock blocksread)
        (mem p proc))
   (forall-hinv2-2-1 (disk)
                     p diskswritten
                     phase disk dblock blocksread))
```

```
  :hints (("goal" :use forall-hinv2-2-mem)))

(defthm
  forall-hinv2-2-1-mem-sane-version
  (implies (and (forall-hinv2-2-1 disk-dom p diskswritten
                                  phase disk dblock blocksread)
                (mem d disk-dom))
           (and
            (implies (mem d (apply diskswritten p))
                     (and (mem (apply phase p) (brace 1 2))
                          (= (apply-m disk d p)
                             (apply dblock p))))
            (implies (mem (apply phase p) (brace 1 2))
                     (and (implies (not (= (apply-m blocksread p d) nil))
                                   (mem d (apply diskswritten p)))
                          (not (hasread p d p blocksread))))))
  :hints (("goal" :use forall-hinv2-2-1-mem
           :in-theory '(forall-hinv2-2-1-predicate))))

(defthm
  forall-hinv2-2-1-mem-sane-version-with-hide-and-disk
  (implies (and (forall-hinv2-2-1 (disk) p diskswritten
                                  phase disk dblock blocksread)
                (mem d (disk)))
           (and
            (implies (mem d (apply diskswritten p))
                     (and (mem (apply phase p) (hide (brace 1 2)))
                          (= (apply-m disk d p)
                             (apply dblock p))))
            (implies (mem (apply phase p) (hide (brace 1 2)))
                     (and (implies (not (= (apply-m blocksread p d) nil))
                                   (mem d (apply diskswritten p)))
                          (not (hasread p d p blocksread))))))
  :hints (("Goal" :use (:instance forall-hinv2-2-1-mem-sane-version
                                  (disk-dom (disk)))
           :expand ((hide (brace 1 2)))
           :in-theory nil)))

)) ; end local progn

(defthm
     hinv2-lemma2
     (implies (and (hinv2)
                   (mem p (proc))
                   (mem d (disk))
```

```
                  (mem (apply phase p) (hide (brace 1 2)))
                  (not (= (apply-m blocksread p d) nil)))
             (mem d (apply diskswritten p)))
    :hints (("goal" :use (forall-hinv2-2-mem-sane-version
             forall-hinv2-2-1-mem-sane-version-with-hide-and-disk)
             :in-theory (enable hinv2))))

(defthm hinv2-lemma
  (implies (and (hinv2)
                (mem p (proc))
                (mem d (disk))
                (mem d (apply diskswritten p)))
           (and (mem (apply phase p) (hide (brace 1 2)))
                (= (apply-m disk d p) (apply dblock p))))
    :hints (("Goal" :use (forall-hinv2-2-mem-sane-version
             forall-hinv2-2-1-mem-sane-version-with-hide-and-disk)
             :in-theory (enable hinv2))))


(defthm phase1or2read-phase-1
  (implies (and (hinv2)
                (mem p (proc))
                (mem d (disk))
                (mem d (apply diskswritten p)))
           (mem (apply phase p) (hide (brace 1 2))))
    :hints (("Goal" :use (forall-hinv2-2-mem-sane-version
                          forall-hinv2-2-1-mem-sane-version-with-hide-and-disk)
             :in-theory (enable hinv2))))

; End of file ------------------------------------------------------------
```

D.9 hinv2.lisp

```
; hinv2.lisp

; I used the following command to certify this book:
; (ld "defpkg.lisp")
; (certify-book "hinv2" 1)

(in-package "S")

(include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
```

```
(include-book "translations")

; ----------------------------------------------------------------------
;
; HInv2 ==
;  /\ forall p in Proc :
;       forall bk in blocksOf(p) : /\ bk.mbal in Ballot(p) cup {0}
;                                  /\ bk.bal  in Ballot(p) cup {0}
;                                  /\ (bk.bal=0) === (bk.inp = NotAnInput)
;                                  /\ bk.mbal >= bk.bal
;
;  /\ forall p in Proc, d in Disk :
;       /\ (d in disksWritten[p]) => /\ phase[p] in {1,2}
;                                    /\ disk[d][p] = dblock[p]
;       /\ (phase[p] in {1,2}) => /\ (blocksRead[p][d] # {}) =>
;                                       (d in disksWritten[p])
;                                 /\ -hasRead(p,d,p)
;  /\ forall p in Proc :
;       /\ (phase[p] = 0) => /\ dblock[p] = InitDB
;                            /\ disksWritten[p] = {}
;                            /\ forall d in Disk :
;                                forall br in blocksRead[p][d] :
;                                  /\ br.proc = p
;                                  /\ br.block = disk[d][p]
;
;       /\ (phase[p] # 0) => /\ dblock[p].mbal in Ballot(p)
;                            /\ dblock[p].bal in Ballot(p) cup {0}
;                            /\ forall d in Disk:
;                                forall br in blocksRead[p][d] :
;                                  br.block.mbal < dblock[p].mbal
;       /\ (phase[p] in {2,3}) => (dblock[p].bal = dblock[p].mbal)
;       /\ output[p] = IF phase[p] = 3 THEN dblock[p].inp ELSE NotAnInput
;
;  /\ chosen in allInput cup {NotAnInput}
;  /\ forall p in Proc :
;      /\ input[p] in allInput
;      /\ (chosen = NotAnInput) => (output[p] = NotAnInput)
;
; ----------------------------------------------------------------------

(defall forall-hinv2-1-1 (blocksof-p p)
  :forall bk :in blocksof-p
  :holds (and (mem (apply bk "mbal") (union (ballot p) (brace 0)))
              (mem (apply bk "bal") (union (ballot p) (brace 0)))
              (iff (= (apply bk "bal") 0)
```

```
                  (= (apply bk "inp") (notaninput)))))

(defall forall-hinv2-1 (proc dblock disk blocksread)
  :forall p :in proc
  :holds (forall-hinv2-1-1 (blocksof p dblock disk blocksread) p))

(defall forall-hinv2-2-1 (disk-dom p diskswritten phase disk dblock blocksread)
  :forall d :in disk-dom
  :holds (and (implies (mem d (apply diskswritten p))
                       (and (mem (apply phase p) (brace 1 2))
                            (= (apply-m disk d p) (apply dblock p))))
              (implies (mem (apply phase p) (brace 1 2))
                       (and (implies (not (= (apply-m blocksread p d) nil))
                                     (mem d (apply diskswritten p)))
                            (not (hasread p d p blocksread))))))

; Abbreviation to add: (not (= s nil)) ---> s

(defall forall-hinv2-2 (proc diskswritten phase disk dblock blocksread)
  :forall p :in proc
  :holds (forall-hinv2-2-1 (disk) p diskswritten phase disk dblock blocksread))

(defall forall-hinv2-3-1-1 (blocksread-p-d p d disk)
  :forall br :in blocksread-p-d
  :holds (and (= (apply br "proc") p)
              (= (apply br "block") (apply-m disk d p))))

(defall forall-hinv2-3-1 (disk-dom blocksread p disk)
  :forall d :in disk-dom
  :holds (forall-hinv2-3-1-1 (apply-m blocksread p d) p d disk))

(defall forall-hinv2-3-2-1 (blocksread-p-d p dblock)
  :forall br :in blocksread-p-d
  :holds (< (apply-m br "block" "mbal")
            (apply-m dblock p "mbal")))

(defall forall-hinv2-3-2 (disk-dom blocksread p dblock)
  :forall d :in disk-dom
  :holds (forall-hinv2-3-2-1 (apply-m blocksread p d) p dblock))

(defall forall-hinv2-3 (proc phase dblock diskswritten blocksread disk output)
  :forall p :in proc
  :holds (and (implies (= (apply phase p) 0)
                       (and (= (apply dblock p) (initdb))
                            (= (apply diskswritten p) nil)
```

```
                    (forall-hinv2-3-1 (disk) blocksread p disk)))
          (implies (not (= (apply phase p) 0))
                   (and (mem (apply-m dblock p "mbal")
                             (ballot p))
                        (mem (apply-m dblock p "bal")
                             (union (ballot p) (brace 0)))
                        (forall-hinv2-3-2 (disk) blocksread p dblock)))
          (implies (mem (apply phase p) (brace 2 3))
                   (= (apply-m dblock p "bal")
                      (apply-m dblock p "mbal")))
          (= (apply output p)
             (if (= (apply phase p) 3)
                 (apply-m dblock p "inp")
               (notaninput)))))

(defall forall-hinv2-4 (proc input allinput chosen output)
  :forall p :in proc
  :holds (and (mem (apply input p) allinput)
              (implies (= chosen (notaninput))
                       (= (apply output p) (notaninput)))))

(defstate hinv2-with-vars (allinput chosen)
  (and (forall-hinv2-1 (proc) dblock disk blocksread)
       (forall-hinv2-2 (proc) diskswritten phase disk dblock blocksread)
       (forall-hinv2-3 (proc) phase dblock diskswritten blocksread disk output)
       (mem chosen (union allinput (brace (notaninput))))
       (forall-hinv2-4 (proc) input allinput chosen output)))

(defmacro hinv2 () '(hinv2-with-vars allinput chosen))
(add-macro-alias hinv2 _hinv2-with-vars)

; End of file ------------------------------------------------------------------
```

D.10 hinv3.lisp

```
; hinv3.lisp

; I used the following command to certify this book:
; (ld "defpkg.lisp")
; (certify-book "hinv3" 1)

(in-package "S")
```

```
(include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
(include-book "translations")

; ------------------------------------------------------------------------
;
; HInv3 == :forall: p,q :in: Proc, d :in: Disk :
;             /\ phase[p] :in: {1,2}
;             /\ phase[q] :in: {1,2}
;             /\ hasRead(p,d,q)
;             /\ hasRead(q,d,p)
;
;             => \/ [block |-> dblock[q], proc |-> q] :in: blocksRead[p][d]
;                \/ [block |-> dblock[p], proc |-> p] :in: blocksRead[q][d]
;
; ------------------------------------------------------------------------

(defun hinv3.l (phase p d q blocksread)
  (and (mem (apply phase p) (hide (brace 1 2)))
       (mem (apply phase q) (hide (brace 1 2)))
       (hasread p d q blocksread)
       (hasread q d p blocksread)))

(defun hinv3.r (dblock blocksread p d q)
  (or (mem (func ("block" (apply dblock q))
                 ("proc" q))
           (apply-m blocksread p d))
      (mem (func ("block" (apply dblock p))
                 ("proc" p))
           (apply-m blocksread q d))))

; These congruences are proven for purposes of forall-hinv3-1-1-1
; below.

(defcong = equal (hinv3.r dblock blocksread p d q) 1)
(defcong = equal (hinv3.r dblock blocksread p d q) 2)
(defcong = equal (hinv3.r dblock blocksread p d q) 3)
(defcong = equal (hinv3.r dblock blocksread p d q) 4)
(defcong = equal (hinv3.r dblock blocksread p d q) 5)

(defcong = equal (hinv3.l phase p d q blocksread) 1)
(defcong = equal (hinv3.l phase p d q blocksread) 2)
(defcong = equal (hinv3.l phase p d q blocksread) 3)
(defcong = equal (hinv3.l phase p d q blocksread) 4)
(defcong = equal (hinv3.l phase p d q blocksread) 5)
```

```
(in-theory (disable hinv3.l hinv3.r))

(defall forall-hinv3-1-1-1 (disk-dom p q phase dblock blocksread)
  :forall d :in disk-dom
  :holds (implies (hinv3.l phase p d q blocksread)
                  (hinv3.r dblock blocksread p d q)))

(defall forall-hinv3-1-1 (proc p phase dblock blocksread)
  :forall q :in proc
  :holds (forall-hinv3-1-1-1 (disk) p q phase dblock blocksread))

(defall forall-hinv3-1 (proc phase dblock blocksread)
  :forall p :in proc
  :holds (forall-hinv3-1-1 (proc) p phase dblock blocksread))

(defun hinv3-with-vars (phase dblock blocksread)
  (forall-hinv3-1 (proc) phase dblock blocksread))

(defmacro hinv3 ()
  '(hinv3-with-vars phase dblock blocksread))

(defmacro hinv3-n ()
  '(hinv3-with-vars phase-n dblock-n blocksread-n))

(add-macro-alias hinv3 hinv3-with-vars)

; End of file ------------------------------------------------------------------
```

D.11 newdefmap.lisp

```
; newdefmap.lisp

(in-package "S")

; I used these commands to certify this book:
; (include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
; (certify-book "newdefmap" 1)


; This is a modified version of defmap, in which I put the :such-that
; predicate inside a black box to prevent the predicate from
```

```
; interfering in the success of proofs, and so that proofs go
; faster. I also add a keyword to omit a theorem that fails if the
; :such-that predicate has an outermost conjunct of the form
; (= <variable> <term>).

; Initial i argument to newdefmap-pred-congruences should be 1.
(defun newdefmap-pred-congruences (vars call i)
  (cond
   ((endp vars) nil)
   (t (cons `(defcong = equal ,call ,i)
            (newdefmap-pred-congruences (cdr vars) call (+ 1 i))))))

; Initial i argument to newdefmap-local-congruences should be 1.
(defun newdefmap-local-congruences (vars call sloc i)
  (cond
   ((endp vars) nil)
   (t (cons (if (equal sloc i)
                `(defx :strategy :congruence ,call ,i :method :subsetp)
              `(defx :strategy :congruence ,call ,i))
            (newdefmap-local-congruences (cdr vars) call sloc (+ 1 i))))))

(defun compute-hint (name name-local theorem-name-local)
  (declare (xargs :mode :program))
  `(:hints (("Goal" :use ((:instance PRED-EQUALS-BODY)
                          (:instance ,theorem-name-local))
             :in-theory '((:rewrite ,(packn-in-pkg
                                      (list name "-EQUALS-" name-local)
                                      'newdefmap)))))))

; Initial i argument to newdefmap-congruences should be 1.
(defun newdefmap-congruences (name name-local vars call i)
  (declare (xargs :mode :program))
  (cond
   ((endp vars) nil)
   (t (cons `(defcong = = ,call ,i
              ,@(compute-hint name
                              name-local
                              (packn-in-pkg
                               (list "=-IMPLIES-=-" name-local "-"
                                     (string (code-char (+ 48 i))))
                               'newdefmap)))
            (newdefmap-congruences name name-local (cdr vars) call (+ 1 i))))))

(defmacro newdefmap (name vars
                          &key
```

```
                        (for 'nil forp)
                        (in 'nil inp)
                        (such-that 'nil such-thatp)
                        (map 'nil mapp)
                        (mem-corollary 't))

 (cond
  ((not (and (symbolp name)
             (acl2::symbol-listp vars)
             forp
             (symbolp for)
             (not (acl2::member-equal for vars))
             inp
             (symbolp in)
             (acl2::member-equal in vars)
             (or (and such-thatp (not mapp))
                 (and (not such-thatp) mapp))))
   `(acl2::er acl2::soft 'newdefmap "Not documented."))

  (such-thatp
   (let* ((x for)
          (s in)
          (sloc (- (length vars) (length (member s vars))))
          (body such-that)
          (s1 (genname1 s 1 (cons x vars)))
          (call `(,name ,@vars))
          (rcall `(,name ,@(put-nth `(scdr ,s) sloc vars)))
          (pred-name (packn-in-pkg (list name "-PRED") 'newdefmap))
          (name-local (packn-in-pkg (list name "-LOCAL") 'newdefmapJ))
          (call-local `(,name-local ,@vars))
          (pred-call `(,pred-name ,@(substitute x s vars))))

 `(encapsulate
   nil

   (local
    (progn
      (defun ,pred-name
        ,(substitute x s vars)
        ,body)

      ,@(newdefmap-pred-congruences (substitute x s vars) pred-call 1)

      (in-theory (disable ,pred-name))
```

```
                                  ; We define name-local in terms of pred-name.

(defun ,name-local (,@vars)
  (if (ur-elementp ,s)
      nil
    (let ((,x (scar ,s)))
      (if ,pred-call
          (scons (scar ,s) (,name-local ,@(put-nth `(scdr ,s)
                                                   sloc vars)))
        (,name-local ,@(put-nth `(scdr ,s) sloc vars))))))

(defthm ,(packn-in-pkg (list "SETP-" name-local) 'newdefmap)
  (setp (,name-local ,@vars)))

(defthm ,(packn-in-pkg (list "UR-ELEMENTP-" name-local) 'newdefmap)
  (equal (ur-elementp ,call-local)
         (equal ,call-local nil)))


(defthm ,(packn-in-pkg (list "MEM-" name-local) 'newdefmap)
  (equal (mem ,x ,call-local)
         (and ,pred-call         ; we write it this way in case body
              (mem ,x ,s)))      ; is not Boolean!
  :otf-flg t)

(defthm ,(packn-in-pkg (list "SUBSETP-" name-local) 'newdefmap)
  (subsetp ,call-local ,s))

,@(newdefmap-local-congruences vars call-local (+ sloc 1) 1)

,@(if mem-corollary
      `((defthm ,(packn-in-pkg (list "MEM-" name-local "-CORROLLARY")
                               'newdefmap)
          (implies (and (subsetp ,s1 ,call-local)
                        (mem ,x ,s1))
                   ,pred-call)))
    nil)

(defthm ,(packn-in-pkg (list "CARDINALITY-" name-local) 'newdefmap)
  (<= (cardinality ,call-local)
      (cardinality ,s))
  :rule-classes :linear)

(defthm ,(packn-in-pkg (list "UNION-" name-local) 'newdefmap)
  (= (,name-local ,@(put-nth `(union ,s1 ,s) sloc vars))
```

```
          (union (,name-local ,@(put-nth s1 sloc vars))
                 ,call-local)))

    (defthm ,(packn-in-pkg (list "INTERSECTION-" name-local) 'newdefmap)
      (= (,name-local ,@(put-nth `(intersection ,s1 ,s) sloc vars))
         (intersection (,name-local ,@(put-nth s1 sloc vars))
                       ,call-local)))

    (defthm ,(packn-in-pkg (list "MEM-CHOOSE-" name-local "-1") 'newdefmap)
      (iff (mem (choose ,call-local)
                ,call-local)
           (not (ur-elementp ,call-local))))

    (defthm ,(packn-in-pkg (list "MEM-CHOOSE-" name-local "-2") 'newdefmap)
      (implies (not (ur-elementp ,call-local))
               (mem (choose ,call-local)
                    ,s)))))

                                        ; Now comes the real function.
(defun ,name (,@vars)
  (if (ur-elementp ,s)
      nil
    (let ((,x (scar ,s)))
      (if ,body
          (scons (scar ,s) ,rcall)
        ,rcall))))

; Now, we show that foo-local and foo are equal. Thus, any
; theorems that held for foo-local trivially hold for foo.
(local
 (defthm ,(packn-in-pkg (list name "-EQUALS-" name-local) 'newdefmap)
   (equal (,name ,@vars) (,name-local ,@vars))
   :hints (("Goal" :in-theory (enable ,pred-name)))))

; We also show that pred-call and body are equal.
(local
 (defthm PRED-EQUALS-BODY
   (equal ,pred-call ,body)
   :hints (("Goal" :in-theory (enable ,pred-name)))))

(defthm ,(packn-in-pkg (list "SETP-" name) 'newdefmap)
  (setp ,call)
  ,@(compute-hint name name-local
                  (packn-in-pkg (list "SETP-" name-local) 'newdefmap)))
```

```
(defthm ,(packn-in-pkg (list "UR-ELEMENTP-" name) 'newdefmap)
  (equal (ur-elementp ,call)
         (equal ,call nil))
  ,@(compute-hint name name-local
                  (packn-in-pkg (list "UR-ELEMENTP-" name-local) 'newdefmap)))


(defthm ,(packn-in-pkg (list "MEM-" name) 'newdefmap)
  (equal (mem ,x ,call)
         (and ,body
              (mem ,x ,s)))
  ,@(compute-hint name name-local
                  (packn-in-pkg (list "MEM-" name-local) 'newdefmap)))


(defthm ,(packn-in-pkg (list "SUBSETP-" name) 'newdefmap)
  (subsetp ,call ,s)
  ,@(compute-hint name name-local
                  (packn-in-pkg (list "SUBSETP-" name-local) 'newdefmap)))

,@(newdefmap-congruences name name-local vars call 1)

,@(if mem-corollary
      `((defthm ,(packn-in-pkg (list "MEM-" name "-CORROLLARY")
                               'newdefmap)
          (implies (and (subsetp ,s1 ,call)
                        (mem ,x ,s1))
                   ,body)
          ,@(compute-hint name name-local
                          (packn-in-pkg (list "MEM-" name-local "-CORROLLARY")
                                        'newdefmap))))
    nil)

(defthm ,(packn-in-pkg (list "CARDINALITY-" name) 'newdefmap)
  (<= (cardinality ,call)
      (cardinality ,s))
  :rule-classes :linear
  ,@(compute-hint name name-local
                  (packn-in-pkg (list "CARDINALITY-" name-local) 'newdefmap)))


(defthm ,(packn-in-pkg (list "UNION-" name) 'newdefmap)
  (= (,name ,@(put-nth `(union ,s1 ,s) sloc vars))
     (union (,name ,@(put-nth s1 sloc vars))
```

```
            ,call))
  ,@(compute-hint name name-local
                  (packn-in-pkg (list "UNION-" name-local) 'newdefmap)))


(defthm ,(packn-in-pkg (list "INTERSECTION-" name) 'newdefmap)
  (= (,name ,@(put-nth `(intersection ,s1 ,s) sloc vars))
     (intersection (,name ,@(put-nth s1 sloc vars))
                   ,call))
  ,@(compute-hint name name-local
                  (packn-in-pkg (list "INTERSECTION-" name-local)
                                'newdefmap)))


(defthm ,(packn-in-pkg (list "MEM-CHOOSE-" name "-1") 'newdefmap)
  (iff (mem (choose ,call)
            ,call)
       (not (ur-elementp ,call)))
  ,@(compute-hint name name-local
                  (packn-in-pkg  (list "MEM-CHOOSE-" name-local "-1")
                                 'newdefmap)))

(defthm ,(packn-in-pkg (list "MEM-CHOOSE-" name "-2") 'newdefmap)
  (implies (not (ur-elementp ,call))
           (mem (choose ,call)
                ,s))
  ,@(compute-hint name name-local
                  (packn-in-pkg  (list "MEM-CHOOSE-" name-local "-2")
                                 'newdefmap)))


; I disable the exported function because I found out that it's
; very expensive have it enabled, and the theorems proved should
; suffice to reason about it. The same holds for defall and
; defexists.
(in-theory (disable ,name)))))

(t ;;; :map

 (let* ((x for)
        (s in)
        (sloc (- (length vars) (length (member s vars))))
        (body map)
        (fx (genname1 x 1 (cons x vars)))
        (s1 (genname1 s 1 (cons fx (cons x vars))))
```

```
          (call `(,name ,@vars))
          (rcall `(,name ,@(put-nth `(scdr ,s) sloc vars))))
      `(encapsulate
        nil
        (defun ,name (,@vars)
          (if (ur-elementp ,s)
              nil
            (let ((,x (scar ,s)))
              (scons ,body ,rcall))))

        (defthm ,(packn-in-pkg (list "SETP-" name) 'newdefmap)
          (setp ,call))

        (defthm ,(packn-in-pkg (list "UR-ELEMENTP-" name) 'newdefmap)
          (equal (ur-elementp ,call)
                 (ur-elementp ,s)))

        (defthm ,(packn-in-pkg (list "WEAK-MEM-" name) 'newdefmap)
          (implies (and (mem ,x ,s)
                        (= ,fx ,body))
                   (mem ,fx ,call)))


        (defthm ,(packn-in-pkg (list "SUBSETP-" name) 'newdefmap)
          (implies (subsetp ,s1 ,s)
                   (subsetp (,name ,@(put-nth s1 sloc vars))
                            ,call)))

        ,@(defmap-congruences vars call (+ sloc 1) 1)

        (defthm ,(packn-in-pkg (list "CARDINALITY-" name) 'newdefmap)
          (<= (cardinality ,call)
              (cardinality ,s))
          :rule-classes :linear)

        (defthm ,(packn-in-pkg (list "UNION-" name) 'newdefmap)
          (= (,name ,@(put-nth `(union ,s1 ,s) sloc vars))
             (union (,name ,@(put-nth s1 sloc vars))
                    ,call)))

; This comment is J's.
; Once I thought that
; (image (intersection s1 s)) = (intersection (image s1) (image s))
; But this is wrong.  Consider
; s1 = {(0 . 1) (0 . 2)}
```

```
; s  = {(1 . 1) (1 . 2)}
; let (body e) = (cdr e) (or (tl e) if the elements are pairps)
; Then the lhs is nil because the two sets are disjoint, but the
; rhs is {1 2}.

        (defthm ,(packn-in-pkg (list "INTERSECTION-" name) 'newdefmap)
          (subsetp (,name ,@(put-nth `(intersection ,s1 ,s) sloc vars))
                   (intersection (,name ,@(put-nth s1 sloc vars))
                                 ,call)))

    ; I disable the exported function because I found out that it's
    ; very expensive have it enabled, and the theorems proved should
    ; suffice to reason about it. The same holds for defall and
    ; defexists.

        (in-theory (disable ,name))

        )))))

; End of file ------------------------------------------------------------------
```

D.12 newpowerset.lisp

```
; newpowerset.lisp

; To certify this book: (ld "defpkg.lisp") (certify-book
; "powerset-examples" 1)

(in-package "S")

; I used these commands to certify this book:
; (include-book "/home/pacheco/acl2-sources/books/finite-set-theory/carlos")
; (certify-book "newpowerset" 1)

; Here is the definition of powerset.
(defmap scons-to-every (e s)
  :for x :in s :map (scons e x))

(defun powerset (s)
  (cond ((ur-elementp s) (brace nil))
        (t (union (powerset (scdr s))
                  (scons-to-every (scar s)
```

```
                                    (powerset (scdr s)))))))
; Powerset builds a set.
(defthm setp-powerset
  (setp (powerset s)))

; In fact, it builds a set of sets.  We have to define that concept
; and prove that it admits = as a congruence.

(defun set-of-setsp (p)
  (if (ur-elementp p)
      t
    (and (setp (scar p))
         (set-of-setsp (scdr p)))))

; Since set-of-setsp is a predicate, we use the canonicalize method.
(defx :strategy :congruence (set-of-setsp p) 1 :method :canonicalize)

; Powerset builds a set of sets.
(defthm set-of-setsp-powerset
  (set-of-setsp (powerset b)))

; Here is the fundamental fact about membership in scons-to-every.
(defthm mem-scons-to-every
  (implies (and (setp p)
                (set-of-setsp p)
                (setp s1))
           (iff (mem s1 (scons-to-every e p))
                (and (mem e s1)
                     (or (mem s1 p)
                         (mem (diff s1 (brace e)) p))))))

; The following function is used to tell ACL2 how to induct in the
; next theorem.  It says: induct on b and assume two inductive hypotheses.
(defun induction-hint (a b)
  (if (ur-elementp b)
      (list a b)
    (list (induction-hint a (scdr b))                              ; hyp 1
          (induction-hint (diff a (brace (scar b))) (scdr b)))))   ; hyp 2

; The powerset contains precisely the subsets.
(defthm powerset-property
 (implies (setp e)
          (iff (mem e (powerset s))
               (subsetp e s)))
 :hints (("Goal" :induct (induction-hint e s))))
```

```
; The next lemma is needed for the final defx command.
(defthm subsetp-scons-to-every-powerset
  (implies (and (set-of-setsp s)
                (subsetp s (powerset b))
                (mem e b))
           (subsetp (scons-to-every e s)
                    (powerset b)))
  :hints (("Goal" :induct (scons-to-every e s))))

; This command establishes that powerset admits = as a congruence.
(defx :strategy :congruence (powerset s) 1 :method :subsetp)

; ----------------------------------------------------------------------
; Stuff below added by Carlos.

; Powerset's key property established (theorem powerset-property), we
; disable its definition.
(in-theory (disable powerset))

; This encapsulate exports equal-mem-powerset. I am unhappy with the
; clunkiness of the events. A good exercise would be to obtain better
; proofs.
(encapsulate nil

(local (defthm mem-set-of-setsp
         (implies (and (set-of-setsp s)
                       (mem e s))
                  (setp e))
         :rule-classes nil))

(local (defthm mem-powerset-setp
         (implies (mem e (powerset s))
                  (setp e))
         :hints (("Goal"
                  :use ((:instance mem-set-of-setsp (s (powerset s)))
                        (:instance set-of-setsp-powerset (b s)))
                  :in-theory nil))
         :rule-classes nil))

(local (defthm rewrite-equal-to-iff
         (implies (and (booleanp (mem e (powerset s)))
                       (booleanp (and (setp e) (subsetp e s))))
                  (equal
                   (equal (mem e (powerset s)) (and (setp e) (subsetp e s)))
```

```
                  (iff (mem e (powerset s)) (and (setp e) (subsetp e s)))))))

(local (defthm iff-mem-powerset
         (iff (mem e (powerset s))
              (and (setp e)
                   (subsetp e s)))
         :hints (("Goal"
                  :use ((:instance mem-powerset-setp)
                        (:instance powerset-property))))))

(defthm equal-mem-powerset
  (equal (mem e (powerset s))
         (and (setp e)
              (subsetp e s)))
  :instructions ((rewrite rewrite-equal-to-iff)
                 (dv 1)
                 (rewrite iff-mem-powerset)
                 top
                 s)))

; Now, some obvious theorems.

(defthm union-powerset
  (implies (and (mem a (powerset s)) (mem b (powerset s)))
                (mem (union a b) (powerset s))))

(defthm intersection-powerset
  (implies (and (mem a (powerset s)) (mem b (powerset s)))
                (mem (intersection a b) (powerset s))))

(defthm scons-powerset
  (implies (and (mem a s) (mem b (powerset s)))
                (mem (scons a b) (powerset s))))

(defthm mem-nil-powerset
  (mem nil (powerset s)))

(defthm weak-powerset-property
  (implies (mem e (powerset s))
           (subsetp e s)))

; Three theorems in this file represent different approaches to
; handling powerset reasoning. The user should choose the approach
; desired by enabling the corresponding theorem.
```

```
(in-theory (disable powerset-property))
(in-theory (disable equal-mem-powerset))
(in-theory (disable weak-powerset-property))

; End of file ------------------------------------------------------------------
```

D.13 tla-translation-macros.lisp

```
; tla-translation-macros.lisp

; I used the following command to certify this book:
; (ld "defpkg.lisp")
; (certify-book "tla-translation-macros" 1)

(in-package "S")

; ------------------------------------------------------------------------------
; Unchanged

(defmacro unchanged (&rest args)
  (cond ((endp args) t)
        (t `(and (= ,(intern-in-package-of-symbol
                      (coerce (append (coerce (symbol-name
                                               (car args))
                                              'list)
                                      (coerce "-N" 'list))
                              'string)
                      'unchanged)
                    ,(car args))
                (unchanged ,@(cdr args))))))

; ------------------------------------------------------------------------------
; Apply-m (m for Multiple arguments)

; Apply-m must be provided at least two arguments. The first argument
; is always the function being applied to.

(defmacro apply-m (&rest args)
  (cond ((equal (length args) 2) `(apply ,(car args) ,(cadr args)))
        (t `(apply (apply-m ,@(butlast args 1)) ,@(last args)))))

; ------------------------------------------------------------------------------
```

```
; Except-m (m for Multiple arguments)

; (except-m f a b c) corresponds to [f EXCEPT ![a][b] = c]
; (except-m f a b c) becomes (except f a (except (apply f a) b c))

; Except-m must be provided at least three arguments. The first
; argument is always the function being operated on, and the last
; argument is the new value introduced.

; We follow the (except-m f a b c) example in binding variables. This
; binding is done purely for purposes of clarity in the code.

(defmacro except-m (&rest args)
  (let ((f (car args))
        (a (cadr args))
        (b (caddr args))
        (c (cdddr args)))
    (cond ((equal (length args) 3) `(except ,f ,a ,b))
          (t `(except ,f ,a (except-m (apply ,f ,a) ,b ,@c))))))

; ----------------------------------------------------------------------------
; Except-and

; (except-and f (a b c) (x y z)) corresponds to [f EXCEPT ![a][b] = c
;                                                        ![x][y] = z]

; (except-and f (a b c) (x y z)) translates to

; (except (except f x (except (apply f x) y z))
;         a
;         (except (apply (except f x (except (apply f x) y z))
;                        a)
;                 b c))

; Except-and must be provided at least two arguments.

; I'll better explain this macro later.

(defmacro except-and (f &rest args)
  (cond ((equal (length args) 1) `(except-m ,f ,@(car args)))
        (t `(except-m (except-and ,f ,@(cdr args)) ,@(car args)))))

; ----------------------------------------------------------------------------
; deftla (a first shot)
```

```
; End of this file ------------------------------------------------------------
```

D.14 translations.lisp

```
; translations.lisp

; Disk Paxos translations
; ======================

; I used these commands to certify this book:
; (include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
; (certify-book "translations" 1)

(in-package "S")

(include-book "additions")
(include-book "newdefmap")
(include-book "/home/pacheco/acl2-sources/books/finite-set-theory/records")

;-----------------------------------------------------------------------------
;
;  CONSTANT N, Inputs
;  ASSUME (N \in Nat) /\ (N > 0)
;
;-----------------------------------------------------------------------------

(encapsulate ((n1 () t))

             (local (defun n1 () 1))

             (defthm n1-constraint
               (and (integerp (n1))
                    (< 0 (n1)))
               :rule-classes :type-prescription))

(defstub inputs () t)

;-----------------------------------------------------------------------------
;
;   Proc == 1..N
```

```
;
;---------------------------------------------------------------------

(defun Proc () (dot-dot 1 (n1)))
(in-theory (disable (:executable-counterpart Proc)))
(in-theory (disable proc))

;---------------------------------------------------------------------
;
;   NotAnInput == CHOOSE c : c notin Inputs
;
;---------------------------------------------------------------------

(encapsulate ((notaninput () t))
             (local (defun notaninput () (inputs)))
             (local (defthm notaninput-helper
                      (equal (acl2::hide (notaninput))
                             (inputs))
                      :hints (("Goal" :expand (acl2::hide (notaninput))
                                      :in-theory
                                      (disable
                                       (:executable-counterpart notaninput))))))

             (defthm notaninput-constraint
               (not (mem (notaninput) (inputs)))))

;---------------------------------------------------------------------
;
; CONSTANTS Ballot(), Disk, IsMajority()
;
; ASSUME /\ \A p \in Proc : Ballot(p) \subset {n \in Nat : n > 0}         |
;        /\ \A q \in Proc \ p : Ballot(p) \cap Ballot(q) = {}
;        /\ \A S,T \in SUBSET Disk :                                      |
;                  IsMajority(S) /\ IsMajority(T) => (S \cap T # {})
;
;---------------------------------------------------------------------

(encapsulate ((ballot (p) t))
             (local (defun ballot (p) (declare (ignore p)) nil))

             ; There is more than one way to express the first
             ; constraint.

;            (local (defall forall-p-in-proc (proc e)
;                     :forall p :in proc
```

```
;                    :holds (implies (mem e (ballot p))
;                                    (and (integerp e) (> e 0)))))

;             (local (defall forall-p-in-proc-minus-p (proc-minus-p p)
;                      :forall q :in proc-minus-p
;                      :holds (= (intersection (ballot p) (ballot q)) nil)))

              ; This is the rule we really want. And it doesn't depend
              ; on the defalls above. Perhaps an additional theorem to
              ; prove for

              ; (defall (s ...) :forall x :in s :holds (implies p q))

              ; is (implies (and (mem x s) p) q). But do we always want
              ; to produce such a theorem? And when do we want to
              ; forward chain or not?

              (defthm ballot-is-set-of-nats
                (implies (and (mem b (ballot p))
                              (mem p (Proc)))
                         (and (integerp b)
                              (> b 0)))
                :rule-classes ((:forward-chaining
                                :trigger-terms ((mem b (ballot p))))))

              (defcong = equal (ballot p) 1)

              ; Here might be another example of TLA statements that
              ; might be better expressed otherwise in ACL2. I suspect
              ; that this theorem might be used to prove a != b, given
              ; that a = (ballot p), b = (ballot q), and p != q. But
              ; the way the theorem is stated, I'm not sure ACL2 will
              ; be able to figure things out (the vagueness of this
              ; statement reflects my own uncertainty of what I'm
              ; saying!)

              (defthm ballot-partitions-nats
                (implies (mem q (diff (proc) (brace p)))
                         (ur-elementp (intersection (ballot p) (ballot q))))))

; this constraint is stronger than paxos: it applies to any s and t,
; whereas the Paxos statement applies only to elements of SUBSET Disk.

(encapsulate ((disk () t)
              (ismajority (s) t))
```

```
             (local (defstub disk () t))
             (local (defun ismajority (s) (declare (ignore s)) nil))
             (defcong = equal (ismajority s) 1)

             ; Once again, I deviate from the TLA statement.
             (defthm is-majority-thm1
               (implies (and (ismajority s)
                             (ismajority s2)
                             (subsetp s (disk))
                             (subsetp s2 (disk)))
                        (not (ur-elementp (intersection s s2))))))

; The translation of the above ASSUME statement repeatedly raises the
; question: "how do we get from a TLA assumption to an effective ACL2
; rule?"


;------------------------------------------------------------------------
;
; DiskBlock == [ mbal : (UNION {Ballot(p) : p \in Proc }) \cup {0},
;                bal  : (UNION {Ballot(p) : p \in Proc }) \cup {0},
;                inp  : Inputs \cup {NotAnInput}
;              ]
;
;------------------------------------------------------------------------

(newdefmap diskblock-map1 (proc) :for p :in proc :map (ballot p))
(in-theory (disable (:executable-counterpart diskblock-map1)))

(defrec diskblock
  ("mbal" (hide (union (union* (diskblock-map1 (proc)))
                       (brace 0))))
  ("bal" (hide (union (union* (diskblock-map1 (proc)))
                      (brace 0))))
  ("inp" (hide (union (inputs) (brace (notaninput))))))

(in-theory (disable diskblock-property))

(defthm diskblock-def
  (iff (mem db (diskblock))
       (and (functionp db)
            (= (domain db) (brace "mbal" "bal" "inp"))
            (mem (apply db "mbal")
                 (hide (union (union* (diskblock-map1 (proc)))
                              (brace 0)))))
```

```
            (mem (apply db "bal")
                 (hide (union (union* (diskblock-map1 (proc)))
                              (brace 0))))
            (mem (apply db "inp")
                 (hide (union (inputs) (brace (notaninput)))))))
  :hints (("Goal" :use diskblock-property
           :in-theory (disable diskblock-property))))

(defthm mem-ballot
  (implies (and (mem e (ballot p))
                (mem p (proc)))
           (mem e (union* (diskblock-map1 (proc))))))

(defthm hide-lemma-1
  (implies (= e 0)
           (mem e (hide (union (union* (diskblock-map1 (proc))) (brace 0)))))
  :hints (("Goal" :expand (hide (union (union* (diskblock-map1 (proc))) (brace 0))))))

(defthm hide-lemma-2
  (implies (mem e (union* (diskblock-map1 (proc))))
           (mem e (hide (union (union* (diskblock-map1 (proc))) (brace 0)))))
  :hints (("Goal" :expand (hide (union (union* (diskblock-map1 (proc))) (brace 0))))))

(defthm hide-lemma-3
  (implies (mem e (inputs))
           (mem e (hide (union (inputs) (brace (notaninput))))))
  :hints (("Goal" :expand (hide (union (inputs) (brace (notaninput)))))))

(defthm hide-lemma-4
  (implies (= e (notaninput))
           (mem e (hide (union (inputs) (brace (notaninput))))))
  :hints (("Goal" :expand (hide (union (inputs) (brace (notaninput)))))))

; Instead of proving all the following "nil" rules, I should establish
; a discipline of, whenever explicitly declaring a diskblock record,
; using a prescribed order for entering the fields.

(defthm diskblock-nil-1
  (implies (and (mem mb (hide (union (union* (diskblock-map1 (proc)))
                                     (brace 0))))
                (mem b (hide (union (union* (diskblock-map1 (proc)))
                                                (brace 0))))
                (mem i (hide (union (inputs) (brace (notaninput))))))
           (mem (except (except (except nil "mbal" mb) "bal" b) "inp" i)
                (diskblock))))
```

```
(defthm diskblock-nil-2
  (implies (and (mem mb (hide (union (union* (diskblock-map1 (proc)))
                                     (brace 0))))
                (mem b (hide (union (union* (diskblock-map1 (proc)))
                                                (brace 0))))
                (mem i (hide (union (inputs) (brace (notaninput))))))
           (mem (except (except (except nil "mbal" mb) "inp" i) "bal" b)
                (diskblock))))

(defthm diskblock-nil-3
  (implies (and (mem mb (hide (union (union* (diskblock-map1 (proc)))
                                     (brace 0))))
                (mem b (hide (union (union* (diskblock-map1 (proc)))
                                                (brace 0))))
                (mem i (hide (union (inputs) (brace (notaninput))))))
           (mem (except (except (except nil "bal" b) "mbal" mb) "inp" i)
                (diskblock))))

(defthm diskblock-nil-4
  (implies (and (mem mb (hide (union (union* (diskblock-map1 (proc)))
                                     (brace 0))))
                (mem b (hide (union (union* (diskblock-map1 (proc)))
                                                (brace 0))))
                (mem i (hide (union (inputs) (brace (notaninput))))))
           (mem (except (except (except nil "bal" b) "inp" i) "mbal" mb)
                (diskblock))))

(defthm diskblock-nil-5
  (implies (and (mem mb (hide (union (union* (diskblock-map1 (proc)))
                                     (brace 0))))
                (mem b (hide (union (union* (diskblock-map1 (proc)))
                                                (brace 0))))
                (mem i (hide (union (inputs) (brace (notaninput))))))
           (mem (except (except (except nil "inp" i) "bal" b) "mbal" mb)
                (diskblock))))

(defthm diskblock-nil-6
  (implies (and (mem mb (hide (union (union* (diskblock-map1 (proc)))
                                     (brace 0))))
                (mem b (hide (union (union* (diskblock-map1 (proc)))
                                                (brace 0))))
                (mem i (hide (union (inputs) (brace (notaninput))))))
           (mem (except (except (except nil "inp" i) "mbal" mb) "bal" b)
                (diskblock))))
```

```
(defthm diskblock-except-inp
  (implies (and (mem f (diskblock))
                (mem e (hide (union (inputs) (brace (notaninput))))))
           (mem (except f "inp" e) (diskblock)))
  :hints (("Goal" :expand (hide (union (inputs) (brace (notaninput)))))))

(defthm diskblock-except-bal
  (implies (and (mem f (diskblock))
                (mem e (hide (union (union* (diskblock-map1 (proc)))
                                    (brace 0)))))
           (mem (except f "bal" e) (diskblock)))
  :hints (("Goal" :expand (hide (union (union* (diskblock-map1 (proc)))
                                       (brace 0))))))

(defthm diskblock-except-mbal
  (implies (and (mem f (diskblock))
                (mem e (hide (union (union* (diskblock-map1 (proc)))
                                    (brace 0)))))
           (mem (except f "mbal" e) (diskblock)))
  :hints (("Goal" :expand (hide (union (union* (diskblock-map1 (proc)))
                                       (brace 0))))))

; I cannot introduce these rules in my defmaps. See "-IMPLIES". So I
; have to do them one by one as I need them. But this is silly.

(defthm silly1
  (implies (mem s (diskblock))
           (mem (apply s "inp")
                (hide (union (inputs)
                             (brace (notaninput)))))))
  :hints (("goal" :expand (hide (union (inputs) (brace (notaninput)))))))

(defthm silly2
  (implies (mem s (diskblock))
           (mem (apply s "mbal")
                (hide (union (union* (diskblock-map1 (proc)))
                             (brace 0)))))
  :hints (("goal" :expand (hide (union (union* (diskblock-map1 (proc)))
                                       (brace 0))))))

(defthm silly3
  (implies (mem s (diskblock))
           (mem (apply s "bal")
                (hide (union (union* (diskblock-map1 (proc)))
```

```
                              (brace 0)))))
  :hints (("goal" :expand (hide (union (union* (diskblock-map1 (proc)))
                                       (brace 0))))))

(defthm silly4
  (implies (and (subsetp s (diskblock))
                (not (ur-elementp s)))
           (mem (choose s) (diskblock)))
  :hints (("goal" :use (:instance mem-subsetp
                                  (e (choose s))
                                  (a s)
                                  (b (diskblock))))))

(in-theory (disable diskblock-def))

; blockproc ------------------------------------------------------------------

(defrec blockproc
  ("block" (diskblock))
  ("proc" (proc)))

(defthm blockproc-except-block
  (implies (and (mem f (blockproc))
                (mem y (diskblock)))
           (mem (except f "block" y) (blockproc))))

(defthm blockproc-except-proc
  (implies (and (mem f (blockproc))
                (mem y (proc)))
           (mem (except f "proc" y) (blockproc))))

(defthm blockproc-apply-block
  (implies (mem f (blockproc))
           (mem (apply f "block") (diskblock))))

(defthm blockproc-apply-proc
  (implies (mem f (blockproc))
           (mem (apply f "proc") (proc))))

(defthm blockproc-nil-subsetp
  (subsetp nil (blockproc)))

(defthm block-proc-nil-1
  (implies (and (mem p (proc))
                (mem d (diskblock)))
```

```
           (mem (except (except nil "proc" p) "block" d)
                (blockproc))))

(defthm block-proc-nil-2
  (implies (and (mem p (proc))
                (mem d (diskblock)))
           (mem (except (except nil "block" d) "proc" p)
                (blockproc))))

(in-theory (disable blockproc-property))

;----------------------------------------------------------------------

(include-book "newpowerset")
(include-book "additions")
(include-book "choose-max")
(include-book "defall")
(include-book "defexists")
(include-book "tla-translation-macros")

;----------------------------------------------------------------------
; Defaction and Defstate

(defmacro defaction (name args body)
  `(progn
     (defun ,(packn-in-pkg (list "_" name) 'defaction) (,@args input input-n
                    output output-n disk disk-n phase
                    phase-n dblock dblock-n diskswritten
                    diskswritten-n blocksread blocksread-n)
       ,body)

     (defmacro ,name ,args
        (list (quote ,(packn-in-pkg (list "_" name) 'defaction))
              ,@args 'input 'input-n
              'output 'output-n 'disk 'disk-n 'phase
              'phase-n 'dblock 'dblock-n 'diskswritten
              'diskswritten-n 'blocksread 'blocksread-n))

     (add-macro-alias ,name ,(packn-in-pkg (list "_" name) 'defaction))))

(defmacro defstate (name args body)
  `(progn
     (defun ,(packn-in-pkg (list "_" name) 'defaction) (,@args input output disk phase dbloc
                    blocksread)
       ,body)
```

```
     (defmacro ,name ,args
       (list (quote ,(packn-in-pkg (list "_" name) 'defaction))
             ,@args 'input 'output 'disk 'phase
             'dblock 'diskswritten 'blocksread))

     (defmacro ,(packn-in-pkg (list name "-N") 'defaction) ,args
       (list  (quote ,(packn-in-pkg (list "_" name) 'defaction))
              ,@args 'input-n 'output-n 'disk-n 'phase-n
              'dblock-n 'diskswritten-n 'blocksread-n))

     (add-macro-alias ,name ,(packn-in-pkg (list "_" name) 'defaction))))

;------------------------------------------------------------------------
;
; hasRead(p,d,q) == exists br in blocksRead[p][d] : br.proc = q
;
;------------------------------------------------------------------------

(defexists exists-hasread (blocksread-p-d q)
  :exists br :in blocksread-p-d :such-that (= (apply br "proc") q))

(defun hasread (p d q blocksread)
  (exists-hasread (apply-m blocksread p d) q))

(defcong = equal (hasread p d q blocksread) 1)
(defcong = equal (hasread p d q blocksread) 2)
(defcong = equal (hasread p d q blocksread) 3)
(defcong = equal (hasread p d q blocksread) 4)

;------------------------------------------------------------------------
;
; allBlocksRead(p) ==
;
;   LET allRdBlks == UNION { blocksRead[p][d] : d in Disk }
;   IN  { br.block : br in allRdBlks }
;
;------------------------------------------------------------------------


(newdefmap allblocksread-map-2 (allrdblks)
  :for br :in allrdblks :map (apply br "block"))

(newdefmap allblocksread-map-3 (disk p blocksread)
  :for d :in disk :map (apply-m blocksread p d))
```

```
(defun allblocksread (p blocksread)
  (allblocksread-map-2 (union* (allblocksread-map-3 (disk) p blocksread))))

(defcong = = (allblocksread p blocksread) 1)
(defcong = = (allblocksread p blocksread) 2)


;----------------------------------------------------------------------
;
; InitDB == [ mbal /-> 0 , bal /-> 0 , inp /-> NotAnInput ]
;
;----------------------------------------------------------------------

(defun InitDB ()
  (func ("mbal" 0) ("bal" 0) ("inp" (NotAnInput))))

(in-theory (disable (:executable-counterpart initdb)))

;----------------------------------------------------------------------
;
; InitializePhase(p) ==
;   /\ disksWritten' = [disksWritten EXCEPT ![p] = {}]
;   /\ blocksRead'   = [blocksRead   EXCEPT ![p] = [d \in Disk /-> {}]]
;
;----------------------------------------------------------------------

(defmap-fn map-to-nil (dom)
  :for x :in dom :map nil)

(defthm apply-map-to-nil
  (= (apply (map-to-nil d) x) nil))

(defthm range-map-to-nil
  (subsetp (range (map-to-nil d))
           (powerset x)))

(defthm map-to-nil-all-fns-powerset
  (mem (map-to-nil s)
       (all-fns s (powerset x)))
  :hints (("Goal" :in-theory (enable all-fns-def))))

(defun _initializephase (p
                         disksWritten diskswritten-n
                         blocksRead blocksread-n)
```

```
  (and (= diskswritten-n (except disksWritten p nil))
       (= blocksread-n (except blocksRead p (map-to-nil (disk))))))

(defcong = equal (_initializephase p a s d f) 1)
(defcong = equal (_initializephase p a s d f) 2)
(defcong = equal (_initializephase p a s d f) 3)
(defcong = equal (_initializephase p a s d f) 4)

(defmacro initializephase (p)
  `(_initializephase ,p disksWritten diskswritten-n
                     blocksRead blocksread-n))

(add-macro-alias initializephase _initializephase)

;----------------------------------------------------------------------------
;
; StartBallot(p) ==
;   /\ phase[p] :in: {1,2}
;   /\ phase' = [phase EXCEPT ![p] = 1]
;   /\ :exists: b :in: Ballot(p) :
;          /\ b > dblock[p].mbal
;          /\ dblock' = [dblock EXCEPT ![p].mbal = b]
;   /\ InitializePhase(p)
;   /\ UNCHANGED <input,output,disk>
;
;----------------------------------------------------------------------------

(defexists exists-startballot (ballot-p dblock dblock-n p)
  :exists b :in ballot-p
  :such-that (and (> b (apply-m dblock p "mbal"))
                  (= dblock-n (except dblock p
                                      (except (apply dblock p) "mbal" b))))

  ; mem-corollary nil must be used when the :such-that term is of the form
  ; (and ... (= <variable> <term>)). For more information, see newdefmap.lisp.

  :mem-corollary nil)

(defaction startballot (p b-witness)
  (and (mem (apply phase p) (brace 1 2))
       (= phase-n (except phase p 1))
       (mem b-witness (ballot p))
       (> b-witness (apply-m dblock p "mbal"))
       (= dblock-n (except dblock p
```

```
                          (except (apply dblock p) "mbal" b-witness)))
;       (exists-startballot (ballot p) dblock dblock-n p)
       (initializephase p)
       (unchanged input output disk)))

;---------------------------------------------------------------------------
;
; Phase1or2Write(p,d) ==
;  /\ phase[p] :in: {1,2}
;  /\ disk' = [disk EXCEPT ![d][p] = dblock[p]]
;  /\ disksWritten' = [disksWritten EXCEPT ![p] = @ :cup: {d}]
;  /\ UNCHANGED <input,output,phase,dblock,blocksRead>
;
;---------------------------------------------------------------------------

(defaction phase1or2write (p d)
  (and (mem (apply phase p) (brace 1 2))
       (= disk-n (except-m disk d p (apply dblock p)))
       (= diskswritten-n
          (except diskswritten p (union (apply diskswritten p) (brace d))))
       (unchanged input output phase dblock blocksread)))

;---------------------------------------------------------------------------
;
; Phase1or2Read(p,d,q) ==
;  /\ d :in: disksWritten[p]
;  /\ IF disk[d][q].mbal < dblock[p].mbal
;      THEN /\ blocksRead' =
;                [blocksRead EXCEPT ![p][d] =
;                 @ :cup: { [block /-> disk[d][q], proc /-> q] } ]
;           /\ UNCHANGED <input,output,disk,phase,dblock,disksWritten>
;      ELSE
;       StartBallot(p)
;
;---------------------------------------------------------------------------

(defaction phase1or2read (p d q b-witness)
  (and (mem d (apply diskswritten p))
       (if (< (apply-m disk d q "mbal") (apply-m dblock p "mbal"))
           (and (= blocksread-n
                   (except-m blocksread
                             p
                             d
                             (union (apply-m blocksread p d)
                                    (brace (func ("block"
```

```
                                                    (apply-m disk d q))
                                                   ("proc" q))))))
                (unchanged input output disk phase dblock diskswritten))
         (startballot p b-witness))))

;-----------------------------------------------------------------------
;
; Phase0Read(p,d) ==
;  /\ phase[p] = 0
;  /\ blocksRead' = [blocksRead EXCEPT
;                      ![p][d] = @ :cup: { [ block /-> disk[d][p],
;                                            proc /-> p ] } ]
;  /\ UNCHANGED <input,output,disk,phase,dblock,disksWritten>
;
;-----------------------------------------------------------------------

(defaction phase0read (p d)
  (and (= (apply phase p) 0)
       (= blocksread-n
          (except-m blocksread
                    p
                    d
                    (union (apply-m blocksread p d)
                           (brace (func ("block" (apply-m disk d p))
                                        ("proc" p))))))
       (unchanged input output disk phase dblock diskswritten)))

;-----------------------------------------------------------------------
;
; Fail(p) ==
;   /\ exists ip in Inputs : input' = [input EXCEPT ![p] = ip]
;   /\ phase'  = [phase  EXCEPT ![p] = 0 ]
;   /\ dblock' = [dblock EXCEPT ![p] = InitDB]
;   /\ output' = [output EXCEPT ![p] = NotAnInput]
;   /\ InitializePhase(p)
;   /\ UNCHANGED disk
;
;-----------------------------------------------------------------------

(defexists exists-fail-1 (inputs input input-n p)
  :exists ip :in inputs :such-that (= input-n (except input p ip))
  :mem-corollary nil)

(defaction fail (p ip-witness1)
  (and (mem ip-witness1 (inputs))
```

```
       (= input-n (except input p ip-witness1))
       (= phase-n  (except phase p 0))
       (= dblock-n (except dblock p (initdb)))
       (= output-n (except output p (notaninput)))
       (initializephase p)
       (unchanged disk)))

;---------------------------------------------------------------------------
;
; EndPhase0(p) ==
;
;   /\ phase[p] = 0
;   /\ isMajority( { d in Disk : hasRead(p,d,p) } )
;   /\ exists b in Ballot(p) :
;        /\ forall r in allBlocksRead(p) : b > r.mbal
;        /\ dblock' = [ dblock EXCEPT
;                         ![p] = [ (CHOOSE r in allBlocksRead(p) :
;                                     forall s in allBlocksRead(p) :
;                                       r.bal >= s.bal)
;                                  EXCEPT !.mbal = b ] ]
;   /\ InitializePhase(p)
;   /\ phase' = [ phase EXCEPT ![p] = 1 ]
;   /\ UNCHANGED <input,output,disk>
;
;---------------------------------------------------------------------------

(newdefmap ep0-map1 (disk p blocksread)
  :for d :in disk
 :such-that (hasread p d p blocksread))

(defall forall-endphase0-1 (allblocksread-p b)
  :forall r :in allblocksread-p
  :holds (> b (apply r "mbal")))

(defexists exists-endphase0-1 (ballot-p p blocksread dblock-n dblock)
  :exists b :in ballot-p
  :such-that
  (and (forall-endphase0-1 (allblocksread p blocksread) b)
       (= dblock-n
          (except dblock p
                  (except (choose-max-bal (allblocksread p blocksread))
                          "mbal" b))))
  :mem-corollary nil)

(defaction endphase0 (p b-wit)
```

```
  (and (= (apply phase p) 0)
       (ismajority (ep0-map1 (disk) p blocksread))
       (mem b-wit (ballot p))
       (forall-endphase0-1 (allblocksread p blocksread) b-wit)
       (= dblock-n
          (except dblock p
                  (except (choose-max-bal (allblocksread p blocksread))
                          "mbal" b-wit)))
       (initializephase p)
       (= phase-n (except phase p 1))
       (unchanged input output disk)))

;-----------------------------------------------------------------------
;
; EndPhase1or2(p) ==
;
;   /\ IsMajority({ d in disksWritten[p] :
;                   forall q in Proc \ {p} : hasRead(p,d,q)})
;   /\ \/ /\ phase[p] = 1
;         /\ dblock' =
;             [dblock EXCEPT
;               ![p].bal = dblock[p].mbal,
;               ![p].inp =
;                 LET blocksSeen == allBlocksRead(p) cup { dblock[p] }
;                     nonInitBlks ==
;                       { bs in blocksSeen : bs.inp # NotAnInput }
;                     maxBlk ==
;                       CHOOSE b in nonInitBlks :
;                          forall c in nonInitBlks : b.bal >= c.bal
;                 IN
;                   IF nonInitBlks = {} THEN input[p]
;                                       ELSE maxBlk.inp ]
;         /\ UNCHANGED output
;
;      \/ /\ phase[p] = 2
;         /\ output' = [output EXCEPT ![p] = dblock[p].inp]
;         /\ UNCHANGED dblock
;
;   /\ phase' = [phase EXCEPT ![p] = @ + 1]
;   /\ InitializePhase(p)
;   /\ UNCHANGED <input, disk>
;
;-----------------------------------------------------------------------

(defall forall-ep12-1 (proc-minus-p p d blocksread)
```

```
  :forall q :in proc-minus-p
  :holds (hasread p d q blocksread))

(defexists exists-ep12-1 (blocksseen)
  :exists bs :in blocksseen
  :such-that (not (= (apply bs "inp") (notaninput))))

(newdefmap map-ep12-1 (diskswritten-p p blocksread)
  :for d :in diskswritten-p
  :such-that (forall-ep12-1 (diff (proc) (brace p)) p d blocksread))

(newdefmap noninitblks (blocksseen)
  :for bs :in blocksseen
  :such-that (not (= (apply bs "inp") (notaninput))))

(defaction endphase1or2 (p)
  (and (ismajority (map-ep12-1 (apply diskswritten p) p blocksread))
       (or (and (= (apply phase p) 1)
                (= dblock-n
                   (except-and dblock
                               (p "bal" (apply-m dblock p "mbal"))
                               (p "inp" (if (=  (noninitblks
                                                 (union (allblocksread p
                                                                       blocksread)
                                                        (brace
                                                         (apply dblock p))))
                                                nil)
                                            (apply input p)
                                           (apply (choose-max-bal
                                                   (noninitblks
                                                    (union (allblocksread p
                                                                          blocksread)
                                                           (brace
                                                            (apply dblock p)))))
                                                  "inp")))))
                (unchanged output))
           (and (= (apply phase p) 2)
                (= output-n (except output p (apply-m dblock p "inp")))
                (unchanged dblock)))
       (= phase-n (except phase p (+ (apply phase p) 1)))
       (initializephase p)
       (unchanged input disk)))

;-----------------------------------------------------------------------
;
```

```
; MajoritySet == { D in SUBSET Disk : IsMajority(D) }
;
; blocksOf(p) ==
;
;  LET rdBy(q,d) == {br in blocksRead[q][d] : br.proc = p }
;  IN { dblock[p] } cup { disk[d][p] : d in Disk }
;                 cup { br.block : br in UNION {rdBy(q,d) : q in Proc,
;                                                          d in Disk }}
;
; allBlocks == UNION { blocksOf(p) : p in Proc
;
;-----------------------------------------------------------------------

(newdefmap majorityset (powset-disk)
  :for d :in powset-disk
  :such-that (ismajority d))

; Translation problem. In spec, we have rdBy(q,d). But here, we can't
; pass q and d as parameters, they have to be given in the call.
(newdefmap rdby (p blocksread-q-d)
  :for br :in blocksread-q-d
  :such-that (= (apply br "proc") p))

(newdefmap rdby-proc-disk-1-1 (disk-dom p q blocksread)
  :for d :in disk-dom
  :map (rdby p (apply-m blocksread q d)))

(newdefmap rdby-proc-disk-1 (proc-dom disk-dom p blocksread)
  :for q :in proc-dom
  :map (rdby-proc-disk-1-1 disk-dom p q blocksread))

(newdefmap br-block (union-rdby-proc-disk-1)
  :for br :in union-rdby-proc-disk-1
  :map (apply br "block"))

(newdefmap disk-d-p (disk-dom disk p)
  :for d :in disk-dom :map (apply-m disk d p))

(defun blocksof (p dblock disk blocksread)
  (union (apply dblock p)
         (union (disk-d-p (disk) disk p)
                (br-block (union* (rdby-proc-disk-1 (proc) (disk) p blocksread))))))

(defcong = = (blocksof p dblock disk blocksread) 1)
(defcong = = (blocksof p dblock disk blocksread) 2)
```

```
(defcong = = (blocksof p dblock disk blocksread) 3)
(defcong = = (blocksof p dblock disk blocksread) 4)

(newdefmap blocks-of-p-map (proc-dom dblock disk blocksread)
  :for p :in proc-dom
  :map (blocksof p dblock disk blocksread))

(defun allblocks (dblock disk blocksread)
  (union* (blocks-of-p-map (proc) dblock disk blocksread)))


;---------------------------------------------------------------------
;
; Next ==
;
;   exists p in Proc :
;       \/ StartBallot(p)
;       \/ exists d in Disk : \/ Phase0Read(p,d)
;                             \/ Phase1or2Write(p,d)
;                             \/ exists q in Proc \ {p} :
;                                  Phase1or2Read(p,d,q)
;       \/ EndPhase1or2(p)
;       \/ Fail(p)
;       \/ EndPhase0(p)
;
;---------------------------------------------------------------------

(defaction next (p              ; witness for next
                 d              ; witness for next
                 q              ; witness for next
                 b-witness1     ; witness for startballot
                 ip-witness1    ; witness for fail
                 b-witness)     ; witness for endphase0

  (and (mem p (proc))
       (or (startballot p b-witness1)
           (and (mem d (disk))
                (or (phase0read p d)
                    (phase1or2write p d)
                    (and (mem q (diff (proc) (brace p)))
                         (phase1or2read p d q b-witness1))))
           (endphase1or2 p)
           (fail p ip-witness1)
           (endphase0 p b-witness))))
```

```
;-----------------------------------------------------------------------
;
; HNext ==
;
;  /\ Next
;  /\ chosen' = LET hasOutput(p) == output'[p] # NotAnInput
;               IN IF \/ chosen # NotAnInput
;                     \/ forall p in Proc : -hasOutput(p)
;                  THEN chosen
;                  ELSE output'[CHOOSE p in Proc : hasOutput(p)]
;  /\ allInput' = allInput cup { input'[p] : p in Proc }
;
;-----------------------------------------------------------------------

(defall forall-hnext (proc output-n)
  :forall p :in proc :holds (= (apply output-n p) (notaninput)))

(newdefmap map-hnext (proc output-n)
  :for p :in proc :such-that (not (= (apply output-n p) (notaninput))))

(defthm hnext-forall-map-property-
  (implies (not (forall-hnext proc output-n))
           (not (ur-elementp (map-hnext proc output-n))))
  :hints (("Goal" :in-theory
           (enable forall-hnext map-hnext forall-hnext-predicate))))

(newdefmap map2-hnext (proc input-n)
  :for p :in proc :map (apply input-n p))

(defun _chosen-allinput-action (chosen chosen-n
                                allinput allinput-n
                                input-n
                                output-n)
  (and (= chosen-n (if (or (not (= chosen (notaninput)))
                           (forall-hnext (proc) output-n))
                       chosen
                     (apply output-n
                            (choose (map-hnext (proc) output-n)))))
       (= allinput-n (union allinput (map2-hnext (proc) input-n)))))

(defmacro chosen-allinput-action ()
  '(_chosen-allinput-action chosen chosen-n allinput allinput-n
                            input-n output-n))

(add-macro-alias chosen-allinput-action _chosen-allinput-action)
```

```
(defaction hnext-with-vars (p d q b-witness1 ip-witness1 b-witness
                            chosen chosen-n allinput allinput-n)
  (and (next p d q b-witness1 ip-witness1 b-witness)
       (chosen-allinput-action)))

(defmacro hnext (p d q b-witness1 ip-witness1 b-witness)
  `(hnext-with-vars ,p ,d ,q ,b-witness1 ,ip-witness1 ,b-witness
                    chosen chosen-n allinput allinput-n))

(add-macro-alias hnext _hnext-with-vars)

; what about the other arguments for the following functions?
(defcong = equal (startballot p b-witness) 1)
(defcong = equal (phase0read p d) 1)
(defcong = equal (phase0read p d) 2)
(defcong = equal (phase1or2read p d q b-witness) 1)
(defcong = equal (phase1or2read p d q b-witness) 2)
(defcong = equal (phase1or2read p d q b-witness) 3)
(defcong = equal (endphase0 p b-wit) 1)
(defcong = equal (endphase1or2 p) 1)
(defcong = equal (phase1or2write p d) 1)
(defcong = equal (phase1or2write p d) 2)
(defcong = equal (fail p ip-witness1) 1)

; End of file ------------------------------------------------------------
```

D.15 i2a/i2a.lisp

```
; i2a.lisp ---------------------------------------------------------------------

; (include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
; (certify-book "i2a" 1)

(in-package "S")

(include-book "i2a-1")
(include-book "../common-all")

; LEMMA I2a. Hinv1 /\ HNext => HInv1'

(disable-all-actions)
```

```
(in-theory (disable hinv1))

(in-theory (enable hnext next))

(defthm i2a
  (implies (and (hnext p d q b-witness1 ip-witness1 b-witness) (hinv1))
           (hinv1-n)))

; End of file ------------------------------------------------------------
```

D.16 i2a/i2a-1.lisp

```
; i2a-1.lisp

; I used the following command to certify this book:
; (include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
; (certify-book "i2a-1" 1)

(in-package "S")

(include-book "../translations")
(include-book "../hinv1")

; next time you certify all-fns, make sure to disable this theorem.
(in-theory (disable main))

; i2a-1.lisp -------------------------------------------------------------

; I forgot to include this one in hinv1.lisp. This should be
; automatically generated once I have a better version of defaction.
(defmacro hinv1-n ()
  '(hinv1-with-vars-n allinput-n chosen-n))


(encapsulate
 nil

(local
(defthm map2-hnext-subsetp-1
  (implies (not (ur-elementp proc))
           (iff (mem x (map2-hnext proc input-n))
                (or (= x (apply input-n (scar proc)))
```

```
                    (mem x (map2-hnext (scdr proc) input-n)))))
  :hints (("Goal" :induct (map2-hnext proc input-n)
           :in-theory (enable map2-hnext)))))

(local
(defthm map2-hnext-subsetp-2
  (implies (and (mem input-n (all-fns proc2 inputs))
                (subsetp proc proc2)
                (mem x (map2-hnext proc input-n)))
           (mem x inputs))
  :hints (("Goal" :induct (map2-hnext proc input-n)
           :in-theory (enable map2-hnext)))))

(local
(defthm map2-hnext-subsetp-3
  (implies (and (mem input-n (all-fns (proc) (inputs)))
                (mem x (map2-hnext (proc) input-n)))
           (mem x (inputs)))
  :hints (("Goal" :use ((:instance map2-hnext-subsetp-2 (proc2 (proc)) (proc (proc))
                                   (inputs (inputs)))
                        (:instance subsetp-x-x (x (proc))))))))

(local
(defx map2-hnext-subsetp-4
  (implies (mem input-n (all-fns (proc) (inputs)))
           (subsetp (map2-hnext (proc) input-n) (inputs)))
  :strategy subset-relation))

(defthm map2-hnext-subsetp
  (implies (mem input-n (all-fns (proc) (inputs)))
           (mem (map2-hnext (proc) input-n) (powerset (inputs))))
  :hints (("Goal" :in-theory (enable powerset-property)))))

; (enable-all-actions)
(local (in-theory (enable hinv1)))

(defthm hinv1-startballot
  (implies
   (and (startballot p b-witness)
        (chosen-allinput-action)
        (hinv1)
        (mem p (proc)))
   (hinv1-n)))

(defthm hinv1-phase1or2write
```

```
  (implies (and (phase1or2write p d)
                (chosen-allinput-action)
                (hinv1)
                (mem p (proc))
                (mem d (disk)))
           (hinv1-n)))

(defthm hinv1-phase1or2read
  (implies (and (phase1or2read p d q b-witness)
                (chosen-allinput-action)
                (hinv1)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk)))
           (hinv1-n)))

(defthm hinv1-phase0read
  (implies (and (phase0read p d)
                (chosen-allinput-action)
                (hinv1)
                (mem p (proc))
                (mem d (disk)))
           (hinv1-n)))

(defthm hinv1-fail
  (implies (and (fail p ip-witness1)
                (chosen-allinput-action)
                (hinv1)
                (mem p (proc)))
           (hinv1-n)))

(encapsulate
 nil

(local (defthm allblocksread-map-3-subsetp-1
  (implies (not (ur-elementp disk))
           (iff (mem x (allblocksread-map-3 disk p blocksread))
                (or (= x (apply-m blocksread p (scar disk)))
                    (mem x (allblocksread-map-3 (scdr disk) p blocksread)))))
  :hints (("Goal" :induct (allblocksread-map-3 disk p blocksread)
           :in-theory (enable allblocksread-map-3)))))

(local (defthm allblocksread-map-3-subsetp-2
  (implies (and (mem blocksread (all-fns proc (all-fns disk2 powerset-blockproc)))
                (subsetp disk disk2)
```

```
               (mem p proc)
               (mem x (allblocksread-map-3 disk p blocksread)))
          (mem x powerset-blockproc))
  :hints (("Goal" :induct (allblocksread-map-3 disk p blocksread)
           :in-theory (enable allblocksread-map-3)))))

(local (defthm allblocksread-map-3-subsetp-3
  (implies (and (mem blocksread (all-fns (proc) (all-fns (disk) (powerset (blockproc)))))
               (mem x (allblocksread-map-3 (disk) p blocksread))
               (mem p (proc)))
          (mem x (powerset (blockproc))))
  :hints (("Goal" :use ((:instance allblocksread-map-3-subsetp-2
                                   (disk2 (disk)) (disk (disk)) (proc (proc))
                                   (powerset-blockproc (powerset (blockproc))))
                        (:instance subsetp-x-x (x (disk))))))))

(defx allblocksread-map-3-subsetp
  (implies (and (mem blocksread (all-fns (proc) (all-fns (disk) (powerset (blockproc)))))
               (mem p (proc)))
           (subsetp (allblocksread-map-3 (disk) p blocksread)
                    (powerset (blockproc))))
  :strategy subset-relation)
) ; end of encapsulate

; I like this theorem.
(defthm union*-powerset
  (implies (subsetp s (powerset a))
           (subsetp (union* s) a))
  :hints (("Goal" :in-theory (enable union* powerset-property))
          ("Subgoal *1/3.1" :in-theory (enable weak-powerset-property))))

(encapsulate
 nil

 (local
  (defthm subsetp-allblocksread-map-2-diskblock-1
    (implies (not (ur-elementp dom))
             (iff (mem x (allblocksread-map-2 dom))
                  (or (= x (apply (scar dom) "block"))
                      (mem x (allblocksread-map-2 (scdr dom))))))
    :hints (("Goal" :induct (allblocksread-map-2 dom)
             :in-theory (enable allblocksread-map-2)))))

(local
 (defthm subsetp-allblocksread-map-2-diskblock-2
```

```
   (implies (and (subsetp dom (blockproc))
                 (mem x (allblocksread-map-2 dom)))
            (mem x (diskblock)))
   :hints (("Goal" :induct (allblocksread-map-2 dom)
            :in-theory (enable allblocksread-map-2)))))


(defx subsetp-allblocksread-map-2-diskblock
  (implies (subsetp dom (blockproc))
           (subsetp (allblocksread-map-2 dom) (diskblock)))
  :strategy subset-relation)

) ; end of encapsulate

(defthm subsetp-choose-max-bal
  (implies (and (subsetp s s2)
                (not (ur-elementp s)))
           (mem (choose-max-bal s) s2))
  :hints (("Goal" :use (:instance mem-subsetp (e (choose-max-bal s))
                                  (a s) (b s2)))))

(defthm subsetp-noninitblks-2
  (implies (subsetp s s2)
           (subsetp (noninitblks s) s2))
  :hints (("Goal" :in-theory (enable noninitblks))))

(defthm goodorbadrule?
  (implies (setp s)
           (iff (= s nil) (ur-elementp s))))

; (defthm goodorbad2
;   (implies (setp x)
;           (iff (not (ur-elementp x)) x)))

(defthm subsetp-nil-x
  (subsetp nil x))

(defthm hinv1-endphase1or2
  (implies (and (endphase1or2 p)
                (chosen-allinput-action)
                (hinv1)
                (mem p (proc)))
           (hinv1-n)))

; add to set theory?
```

```
(defthm intersection-s-s
  (implies (setp s)
           (= (intersection s s) s)))

(local
 (progn

(defthm is-majority-implies-not-ur-elementp
  (implies (and (ismajority s)
                (subsetp s (disk)))
           (not (ur-elementp s)))
  :hints (("Goal" :use ((:instance is-majority-thm1 (s s) (s2 s))
                        (:instance intersection-s-s))
           :in-theory (disable intersection-s-s is-majority-thm1))))

(defthm step1
  (iff (not (ur-elementp (ep0-map1 disk p blocksread)))
           (mem (choose (ep0-map1 disk p blocksread))
                (ep0-map1 disk p blocksread))))

(in-theory (disable mem-choose-ep0-map1-1))

(defthm step2
  (implies (not (ur-elementp (ep0-map1 disk p blocksread)))
           (hasread p (choose (ep0-map1 disk p blocksread)) p blocksread))
  :instructions ((:DV 1) (:REWRITE STEP1)
                 (:REWRITE MEM-EP0-MAP1) top bash))

(defthm step3
  (implies (not (ur-elementp (ep0-map1 disk p blocksread)))
           (exists-hasread
            (apply-m blocksread p (choose (ep0-map1 disk p blocksread))) p))
  :hints (("Goal" :use (:instance step2))))

(defthm step4
  (implies (exists-hasread dom p)
           (not (ur-elementp dom)))
  :hints (("Goal" :in-theory (enable exists-hasread exists-hasread-map)))
  :rule-classes nil)

(defthm step5
  (implies (exists-hasread
            (apply-m blocksread p (choose (ep0-map1 disk p blocksread))) p)
           (not (ur-elementp
                 (apply-m blocksread p (choose (ep0-map1 disk p blocksread))))))
```

```
  :hints (("Goal" :by step4)))


(defthm step5andahalf
  (implies (and (mem d disk)
                (not (= (apply-m blocksread p d)
                 (scar (allblocksread-map-3 disk p blocksread)))))
           (mem (apply-m blocksread p d)
                (scdr (allblocksread-map-3 disk p blocksread))))
 :hints (("Goal" :in-theory (enable allblocksread-map-3))))

(defthm step6
  (implies (and (not (ur-elementp (apply-m blocksread p d)))
                (mem d disk))
           (not (ur-elementp (union* (allblocksread-map-3 disk p blocksread)))))
  :hints (("Goal" :induct (allblocksread-map-3 disk p blocksread)
           :in-theory (enable allblocksread-map-3))))

)) ; end local progn

; yuck.
(defthm
 ismajority-implies-not-ur-elementp-allblocksread
 (implies
  (ismajority (ep0-map1 (disk) p blocksread))
  (not
    (ur-elementp
         (allblocksread-map-2 (union* (allblocksread-map-3 (disk)
                                                           p blocksread))))))
 :instructions
 (:promote (:dv 1)
           (:rewrite ur-elementp-allblocksread-map-2)
           (:rewrite step6
                     ((d (choose (ep0-map1 (disk) p blocksread)))))
           (:change-goal nil t)
           :bash (:dv 1)
           (:rewrite step5)
           (:rewrite step3)
           (:rewrite step1)
           :bash))

(local (in-theory (enable powerset-property)))

(defthm hinv1-endphase0
  (implies (and (endphase0 p b-wit)
```

```
               (chosen-allinput-action)
               (hinv1)
               (mem p (proc)))
          (hinv1-n)))

; End of file ------------------------------------------------------------
```

D.17 i2c/common-i2c.lisp

```
; common-i2c.lisp

; I used the following command to certify this book:
; (ld "defpkg.lisp")
; (certify-book "common-i2c" 1)

(in-package "S")

(include-book "/home/pacheco/acl2-sources/books/finite-set-theory/set-theory")
(include-book "../translations")
(include-book "../hinv1")
(include-book "../hinv2")
(include-book "../hinv3")
(include-book "../common-all")

(disable-all-actions)
(in-theory (disable hinv1 hinv2 hinv3))

; (defthm exists-hasread-nil
;   (equal (exists-hasread nil q) nil))

; The next three in order of usefulness (I think).

; (defexists exists-hasread (blocksread-p-d q)
;   :exists br :in blocksread-p-d :such-that (= (apply br "proc") q))

(defthm not-exists-hasread-nil
  (not (exists-hasread nil q))
  :hints (("Goal" :in-theory (enable exists-hasread
                                     exists-hasread-map))))

(defthm initphase-hasread-2
  (implies (initializephase p2)
```

```
          (not (hasread p2 d q blocksread-n)))
  :hints (("Goal" :in-theory (enable initializephase hasread))))

(defthm initphase-hasread
  (implies (and (initializephase p)
                (= p2 p))
           (not (hasread p2 d q blocksread-n)))
  :hints (("Goal" :use initphase-hasread-2
           :in-theory (disable initphase-hasread-2))))

(defthm wb-initializephase
  (implies (and (initializephase p2)
                (not (= p2 p)))
           (and (= (apply diskswritten-n p) (apply diskswritten p))
                (= (apply blocksread-n p) (apply blocksread p))
                (= (apply-m blocksread-n p d) (apply-m blocksread p d))))
  :hints (("Goal" :in-theory (enable initializephase))))

(defthm initializephase-hasread
  (implies (and (initializephase p2)
                (not (= p2 p)))
           (iff (hasread p d q blocksread-n)
                (hasread p d q blocksread)))
  :hints (("Goal" :in-theory (enable hasread))))


(local
(DEFTHM
  FORALL-HINV3-1-1-MEM-painful
  (IMPLIES
   (AND (FORALL-HINV3-1-1 PROC P PHASE DBLOCK BLOCKSREAD)
        (MEM Q PROC))
   (FORALL-HINV3-1-1-1 (DISK)
                                     P Q PHASE DBLOCK BLOCKSREAD))
  :hints (("Goal" :use (forall-hinv3-1-1-mem
                        forall-hinv3-1-1-predicate)
           :in-theory nil))))

(local
(DEFTHM
  FORALL-HINV3-1-Mem-painful
  (IMPLIES
   (AND (FORALL-HINV3-1 PROC PHASE DBLOCK BLOCKSREAD)
        (MEM P PROC))
   (FORALL-HINV3-1-1 (PROC)
```

```
                    P PHASE DBLOCK BLOCKSREAD))
  :hints (("Goal" :use (forall-hinv3-1-mem
                        forall-hinv3-1-predicate)
           :in-theory nil))))

(local
(DEFTHM
     HINV3-IMPLIES-PARTICULAR0
     (IMPLIES (AND (HINV3)
                   (MEM P (PROC))
                   (MEM Q (PROC))
                   (MEM D (DISK)))
              (FORALL-HINV3-1-1-1-PREDICATE D P Q PHASE DBLOCK BLOCKSREAD))
     :INSTRUCTIONS
     (:PROMOTE (:REWRITE FORALL-HINV3-1-1-1-MEM
                         ((DISK-DOM (DISK))))
               (:REWRITE FORALL-HINV3-1-1-MEM-PAINFUL
                         ((PROC (PROC))))
               (:REWRITE FORALL-HINV3-1-MEM-PAINFUL
                         ((PROC (PROC))))
               :DEMOTE (:DV 1 1)
               :EXPAND :TOP :BASH)))

(defthm hinv3-implies-particular
  (implies (and (hinv3) (mem p (proc)) (mem q (proc)) (mem d (disk))
                (hinv3.l phase p d q blocksread))
           (hinv3.r dblock blocksread p d q))
  :hints (("goal" :use (hinv3-implies-particular0)
           :in-theory '(forall-hinv3-1-1-1-predicate))))

; End of file ------------------------------------------------------------------
```

D.18 i2c/common-p12r.lisp

```
; common-p12r.lisp

; I used the following command to certify this book:
; (ld "defpkg.lisp")
; (certify-book "common-p12r" 1)

(in-package "S")
```

```
(include-book "common-i2c")
(include-book "startballot")

; Follows from startballot
(defthm p12r-trivial-on-else-branch
  (implies (and (hinv1) (hinv2) (hinv3)
                (phase1or2read p2 d2 q2 b-witness)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (not (< (apply-m disk d2 q2 "mbal")
                        (apply-m dblock p2 "mbal")))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable phase1or2read))))

(defthm wb-phase1or2read
  (implies (and (phase1or2read p2 d2 q2 b-witness)
                (not (= p2 p)))
           (and (= (apply input-n p) (apply input p))
                (= (apply output-n p) (apply output p))
                (= (apply-m disk-n d p) (apply-m disk d p))
                (= (apply phase-n p) (apply phase p))
                (= (apply dblock-n p) (apply dblock p))
                (= (apply diskswritten-n p) (apply diskswritten p))
                (= (apply-m blocksread-n p d) (apply-m blocksread p d))
                (= (apply blocksread-n p) (apply blocksread p))))
  :hints (("Goal" :in-theory (enable phase1or2read startballot))))

(defthm wb-phase1or2read-hasread
  (implies (and (phase1or2read p2 d2 q2 b-witness)
                (not (= p2 p)))
           (iff (hasread p d q blocksread-n)
                (hasread p d q blocksread)))
  :hints (("Goal" :in-theory (enable hasread))))

; End of file ------------------------------------------------------------------
```

D.19 i2c/ep0.lisp

```
; ep0.lisp

(in-package "S")
```

```
; I used the following command to certify this book:
; (ld "defpkg.lisp")
; (certify-book "ep0" 1)

(include-book "common-i2c")

(local
 (progn

(defthm wb-endphase0
  (implies (and (endphase0 p2 b-wit)
                (not (= p2 p)))
           (and (= (apply input-n p) (apply input p))
                (= (apply output-n p) (apply output p))
                (= (apply-m disk-n d p) (apply-m disk d p))
                (= (apply phase-n p) (apply phase p))
                (= (apply dblock-n p) (apply dblock p))
                (= (apply diskswritten-n p) (apply diskswritten p))
                (= (apply-m blocksread-n p d) (apply-m blocksread p d))
                (= (apply blocksread-n p) (apply blocksread p))))
  :hints (("Goal" :in-theory (enable endphase0))))

(defthm i2c-ep0-0
  (implies (and (hinv1) (hinv2) (hinv3) (endphase0 p2 b-wit)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (= p2 p)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable hinv3.l hinv3.r endphase0))))

(defthm i2c-ep0-1
  (implies (and (hinv1) (hinv2) (hinv3) (endphase0 p2 b-wit)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (= p2 q)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable hinv3.l hinv3.r endphase0))))

(defthm i2c-ep0-2
  (implies (and (hinv1) (hinv2) (hinv3) (endphase0 p2 b-wit)
                (mem p (proc))
```

```
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hasread p d q blocksread-n)
                (hasread p d q blocksread)))
  :hints (("Goal" :in-theory (enable hasread))))

(defthm i2c-ep0-3
  (implies (and (hinv1) (hinv2) (hinv3) (endphase0 p2 b-wit)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hinv3.l phase-n p d q blocksread-n)
                (hinv3.l phase p d q blocksread)))
  :hints (("Goal" :in-theory (enable hinv3.l))))

(defthm i2c-ep0-4
  (implies (and (hinv1) (hinv2) (hinv3) (endphase0 p2 b-wit)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hinv3.r dblock-n blocksread-n p d q)
                (hinv3.r dblock blocksread p d q)))
  :hints (("Goal" :in-theory (enable hinv3.r))))

)) ; end local progn

(defthm i2c-ep0
  (implies (and (hinv1) (hinv2) (hinv3) (endphase0 p2 b-wit)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :cases ((and (not (= p2 p)) (not (= p2 q)))))))

; End of file ------------------------------------------------------------------
```

D.20 i2c/ep12.lisp

```
; ep12.lisp

; (ld "defpkg.lisp")
; (certify-book "ep12" 1)

(in-package "S")

(include-book "common-i2c")

(local
 (progn


(defthm endphase1or2-hasread
  (implies (and (endphase1or2 p)
                (= p2 p))
           (not (hasread p2 d q blocksread-n)))
  :hints (("Goal" :in-theory (enable endphase1or2))))

(defthm endphase1or2-hasread-2
  (implies (endphase1or2 p2)
           (not (hasread p2 d q blocksread-n)))
  :hints (("Goal" :in-theory (enable endphase1or2))))


(defthm wb-endphase1or2
  (implies (and (endphase1or2 p2)
                (not (= p2 p)))
           (and (= (apply input-n p) (apply input p))
                (= (apply output-n p) (apply output p))
                (= (apply-m disk-n d p) (apply-m disk d p))
                (= (apply phase-n p) (apply phase p))
                (= (apply dblock-n p) (apply dblock p))
                (= (apply diskswritten-n p) (apply diskswritten p))
                (= (apply-m blocksread-n p d) (apply-m blocksread p d))
                (= (apply blocksread-n p) (apply blocksread p))))
  :hints (("Goal" :in-theory (enable endphase1or2))))

(defthm i2c-ep12-0
  (implies (and (hinv1) (hinv2) (hinv3) (endphase1or2 p2)
                (mem p (proc))
                (mem q (proc))
```

```
               (mem d (disk))
               (= p2 p)
               (hinv3.l phase-n p d q blocksread-n))
          (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable hinv3.l hinv3.r endphase0))))

(defthm i2c-ep12-1
  (implies (and (hinv1) (hinv2) (hinv3) (endphase1or2 p2)
               (mem p (proc))
               (mem q (proc))
               (mem d (disk))
               (= p2 q)
               (hinv3.l phase-n p d q blocksread-n))
          (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable hinv3.l hinv3.r endphase0))))

(defthm i2c-ep12-2
  (implies (and (hinv1) (hinv2) (hinv3) (endphase1or2 p2)
               (mem p (proc))
               (mem q (proc))
               (mem d (disk))
               (not (= p2 p))
               (not (= p2 q)))
          (iff (hasread p d q blocksread-n)
               (hasread p d q blocksread)))
  :hints (("Goal" :in-theory (enable hasread))))

(defthm i2c-ep12-3
  (implies (and (hinv1) (hinv2) (hinv3) (endphase1or2 p2)
               (mem p (proc))
               (mem q (proc))
               (mem d (disk))
               (not (= p2 p))
               (not (= p2 q)))
          (iff (hinv3.l phase-n p d q blocksread-n)
               (hinv3.l phase p d q blocksread)))
  :hints (("Goal" :in-theory (enable hinv3.l))))

(defthm i2c-ep12-4
  (implies (and (hinv1) (hinv2) (hinv3) (endphase1or2 p2)
               (mem p (proc))
               (mem q (proc))
               (mem d (disk))
               (not (= p2 p))
               (not (= p2 q)))
```

```
           (iff (hinv3.r dblock-n blocksread-n p d q)
                (hinv3.r dblock blocksread p d q)))
  :hints (("Goal" :in-theory (enable hinv3.r))))

)) ;end local progn

(defthm i2c-ep12
  (implies (and (hinv1) (hinv2) (hinv3) (endphase1or2 p2)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :cases ((and (not (= p2 p)) (not (= p2 q)))))))

; End of file ------------------------------------------------------------------
```

D.21 i2c/fail.lisp

```
; fail.lisp

(in-package "S")

; (ld "defpkg.lisp")
; (certify-book "fail" 1)

(include-book "common-i2c")

(local
 (progn


(defthm fail-hasread
  (implies  (and (fail p ip-witness1)
                 (= p2 p))
            (not (hasread p2 d q blocksread-n)))
  :hints (("Goal" :in-theory (enable fail))))

(defthm fail-hasread-2
  (implies  (fail p2 ip-witness1)
            (not (hasread p2 d q blocksread-n)))
  :hints (("Goal" :in-theory (enable fail))))
```

```
(defthm wb-fail
  (implies (and (fail p2 ip-witness1)
                (not (= p2 p)))
           (and (= (apply input-n p) (apply input p))
                (= (apply output-n p) (apply output p))
                (= (apply-m disk-n d p) (apply-m disk d p))
                (= (apply phase-n p) (apply phase p))
                (= (apply dblock-n p) (apply dblock p))
                (= (apply diskswritten-n p) (apply diskswritten p))
                (= (apply-m blocksread-n p d) (apply-m blocksread p d))
                (= (apply blocksread-n p) (apply blocksread p))))
  :hints (("Goal" :in-theory (enable fail))))

(defthm i2c-fail-0
  (implies (and (hinv1) (hinv2) (hinv3) (fail p2 ip-witness1)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (= p2 p)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable hinv3.l hinv3.r endphase0))))

(defthm i2c-fail-1
  (implies (and (hinv1) (hinv2) (hinv3) (fail p2 ip-witness1)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (= p2 q)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable hinv3.l hinv3.r endphase0))))

(defthm i2c-fail-2
  (implies (and (hinv1) (hinv2) (hinv3) (fail p2 ip-witness1)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hasread p d q blocksread-n)
                (hasread p d q blocksread)))
  :hints (("Goal" :in-theory (enable hasread))))
```

```
(defthm i2c-fail-3
  (implies (and (hinv1) (hinv2) (hinv3) (fail p2 ip-witness1)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hinv3.l phase-n p d q blocksread-n)
                (hinv3.l phase p d q blocksread)))
  :hints (("Goal" :in-theory (enable hinv3.l))))

(defthm i2c-fail-4
  (implies (and (hinv1) (hinv2) (hinv3) (fail p2 ip-witness1)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hinv3.r dblock-n blocksread-n p d q)
                (hinv3.r dblock blocksread p d q)))
  :hints (("Goal" :in-theory (enable hinv3.r))))

)) ;end local progn

(defthm i2c-fail
  (implies (and (hinv1) (hinv2) (hinv3) (fail p2 ip-witness1)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :cases ((and (not (= p2 p)) (not (= p2 q)))))))

; End of file ------------------------------------------------------------------
```

D.22 i2c/i2c.lisp

```
; i2c.lisp

(in-package "S")

; (ld "defpkg.lisp")
```

```
; (certify-book "i2c" 1)

(include-book "startballot")
(include-book "p0r")
(include-book "ep0")
(include-book "p12w")
(include-book "p12r")
(include-book "ep12")
(include-book "fail")

(encapsulate
 ((p (x) t)
  (hyps () t)
  (s () t))
 (local (defun p (x) (declare (ignore x)) t))
 (local (defun hyps () nil))
 (local (defun s () t))
 (defcong = equal (p x) 1)

 (defthm mem-foo (implies (and (hyps)
                               (mem x (s)))
                          (p x)))
)

(defall foo2 (s)
  :forall x :in s :holds (p x))

(defthm foo-main-helper
  (implies
   (and (hyps)
        (subsetp s1 (s)))
   (foo2 s1))
  :hints (("Goal" :in-theory (enable foo2))))

(defthm main-foo (implies (hyps)
                          (foo2 (s))))

(defthm i2c-1
  (implies (and (hinv1) (hinv2) (hinv3) (hnext p2 d2 q2
                                               b-witness1 ip-witness1 b-witness)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
```

```
  :hints (("Goal" :in-theory (enable hnext next))))

(defthm i2c-2
  (implies (and (hinv1) (hinv2) (hinv3) (hnext p2 d2 q2
                                               b-witness1 ip-witness1 b-witness)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk)))
           (forall-hinv3-1-1-1-predicate d p q phase-n dblock-n blocksread-n)))

(in-theory (enable forall-hinv3-1-1-1))

(defthm i2c-3
  (implies (and (hinv1) (hinv2) (hinv3) (hnext p2 d2 q2
                                               b-witness1 ip-witness1 b-witness)
                (mem p (proc))
                (mem q (proc)))
           (forall-hinv3-1-1-1 (disk) p q phase-n dblock-n blocksread-n))
  :hints (("Goal" :by (:functional-instance
                       main-foo
                       (s (lambda () (disk)))
                       (foo2-predicate (lambda (x)
           (forall-hinv3-1-1-1-predicate x p q phase-n dblock-n blocksread-n)))
                       (hyps (lambda ()
 (and (hinv1) (hinv2) (hinv3) (hnext p2 d2 q2
                                     b-witness1 ip-witness1 b-witness)
      (mem p (proc)) (mem q (proc)))))
                       (p (lambda (x)
           (forall-hinv3-1-1-1-predicate x p q phase-n dblock-n blocksread-n)))
                       (foo2 (lambda (dom)
                  (forall-hinv3-1-1-1 dom p q phase-n dblock-n blocksread-n))))))
  :otf-flg t)

(in-theory (enable forall-hinv3-1-1))

(defthm i2c-4
  (implies (and (hinv1) (hinv2) (hinv3) (hnext p2 d2 q2
                                               b-witness1 ip-witness1 b-witness)
                (mem p (proc)))
           (forall-hinv3-1-1 (proc) p phase-n dblock-n blocksread-n))
  :hints (("Goal" :by (:functional-instance
                       main-foo
                       (s (lambda () (proc)))
                       (foo2-predicate (lambda (x)
           (forall-hinv3-1-1-predicate x p phase-n dblock-n blocksread-n)))
```

```
                      (hyps (lambda ()
 (and (hinv1) (hinv2) (hinv3) (hnext p2 d2 q2
                                   b-witness1 ip-witness1 b-witness)
      (mem p (proc)))))
                      (p (lambda (x)
           (forall-hinv3-1-1-predicate x p phase-n dblock-n blocksread-n)))
                      (foo2 (lambda (dom)
                 (forall-hinv3-1-1 dom p phase-n dblock-n blocksread-n))))))
  :otf-flg t)

(in-theory (enable forall-hinv3-1))

(defthm i2c-5
  (implies (and (hinv1) (hinv2) (hinv3) (hnext p2 d2 q2
                                                b-witness1 ip-witness1 b-witness))
           (forall-hinv3-1 (proc) phase-n dblock-n blocksread-n))
  :hints (("Goal" :by (:functional-instance
                      main-foo
                      (s (lambda () (proc)))
                      (foo2-predicate (lambda (x)
           (forall-hinv3-1-predicate x phase-n dblock-n blocksread-n)))
                      (hyps (lambda ()
 (and (hinv1) (hinv2) (hinv3) (hnext p2 d2 q2 b-witness1 ip-witness1 b-witness))))
                      (p (lambda (x)
           (forall-hinv3-1-predicate x phase-n dblock-n blocksread-n)))
                      (foo2 (lambda (dom)
                 (forall-hinv3-1 dom phase-n dblock-n blocksread-n))))))
  :otf-flg t)

(in-theory (enable hinv3))

(defthm i2c
  (implies (and (hinv1) (hinv2) (hinv3)
                (hnext p2 d2 q2 b-witness1 ip-witness1 b-witness))
           (hinv3-n)))

; End of file ------------------------------------------------------------------
```

D.23 i2c/p0r.lisp

```
; p0r.lisp
```

```
(in-package "S")

; (ld "defpkg.lisp")
; (certify-book "p0r" 1)

(include-book "common-i2c")

(local
 (progn

(defthm phase0read-hasread
  (implies (and (phase0read p d)
                (= p2 p))
           (not (mem (apply phase-n p2) (hide (brace 1 2)))))
  :hints (("Goal" :in-theory (enable phase0read))))

(defthm phase0read-hasread-2
  (implies (phase0read p2 d)
           (not (mem (apply phase-n p2) (hide (brace 1 2)))))
  :hints (("Goal" :in-theory (enable phase0read))))

(defthm wb-phase0read
  (implies (and (phase0read p2 d2)
                (not (= p2 p)))
           (and (= (apply input-n p) (apply input p))
                (= (apply output-n p) (apply output p))
                (= (apply-m disk-n d p) (apply-m disk d p))
                (= (apply phase-n p) (apply phase p))
                (= (apply dblock-n p) (apply dblock p))
                (= (apply diskswritten-n p) (apply diskswritten p))
                (= (apply-m blocksread-n p d) (apply-m blocksread p d))
                (= (apply blocksread-n p) (apply blocksread p))))
  :hints (("Goal" :in-theory (enable phase0read))))


(defthm i2c-p0r-0
  (implies (and (hinv1) (hinv2) (hinv3) (phase0read p2 d2)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (= p2 p)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable hinv3.l hinv3.r endphase0))))
```

```
(defthm i2c-p0r-1
  (implies (and (hinv1) (hinv2) (hinv3) (phase0read p2 d2)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (= p2 q)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable hinv3.l hinv3.r endphase0))))

(defthm i2c-p0r-2
  (implies (and (hinv1) (hinv2) (hinv3) (phase0read p2 d2)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hasread p d q blocksread-n)
                (hasread p d q blocksread)))
  :hints (("Goal" :in-theory (enable hasread))))

(defthm i2c-p0r-3
  (implies (and (hinv1) (hinv2) (hinv3) (phase0read p2 d2)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hinv3.l phase-n p d q blocksread-n)
                (hinv3.l phase p d q blocksread)))
  :hints (("Goal" :in-theory (enable hinv3.l))))

(defthm i2c-p0r-4
  (implies (and (hinv1) (hinv2) (hinv3) (phase0read p2 d2)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hinv3.r dblock-n blocksread-n p d q)
                (hinv3.r dblock blocksread p d q)))
  :hints (("Goal" :in-theory (enable hinv3.r))))


(defthm i2c-p0r-5
```

```
  (implies (and (hinv1) (hinv2) (hinv3) (phase0read p2 d2)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
                (not (= p2 q))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :use hinv3-implies-particular
           :in-theory (disable hinv3-implies-particular))))

)) ; end local progn

(defthm i2c-p0r
  (implies (and (hinv1) (hinv2) (hinv3) (phase0read p2 d2)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :cases ((and (not (= p2 p)) (not (= p2 q)))))))

; End of file ------------------------------------------------------------------
```

D.24 i2c/p12r-d2=d.lisp

```
; p12r-d2=d.lisp

(in-package "S")

; (ld "defpkg.lisp")
; (certify-book "p12r-d2=d" 1)

(include-book "common-i2c")
(include-book "common-p12r")
(include-book "startballot")
(include-book "p12r-p2=p")
(include-book "p12r-p2=q")

; case: (and (not (= p2 p) (= p2 q)))
; -----------------------------------
```

```
(defthm p12r-p2not=p-p2not=q-hinv3.l
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (= d2 d)
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hinv3.l phase-n p d q blocksread-n)
                (hinv3.l phase   p d q blocksread)))
  :hints (("Goal" :in-theory (enable hinv3.l))))

(defthm p12r-p2not=p-p2not=q-hinv3.r
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (= d2 d)
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hinv3.r dblock-n blocksread-n p d q)
                (hinv3.r dblock   blocksread   p d q)))
  :hints (("Goal" :in-theory (enable hinv3.r))))

(defthm p12r-p2not=p-p2not=q
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (= d2 d)
                (not (= p2 p))
                (not (= p2 q))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :use hinv3-implies-particular)))

; case: (= p2 p)
; --------------




; case: (= p2 q)
; --------------

(defthm p12r-d2=d
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
```

```
               (phase1or2read p2 d2 q2 b-witness)
               (= d2 d)
               (hinv3.l phase-n p d q blocksread-n))
          (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :cases ((and (not (= p2 p)) (not (= p2 q)))))))

; End of file ------------------------------------------------------------------
```

D.25 i2c/p12r-d2not=d.lisp

```
; p12r-d2not=d.lisp

; (ld "defpkg.lisp")
; (certify-book "p12r-d2not=d" 1)

(in-package "S")

(include-book "common-i2c")
(include-book "startballot")

(local
 (progn

(defthm p12r-trivial-on-else-branch
  (implies (and (hinv1) (hinv2) (hinv3)
                (phase1or2read p2 d2 q2 b-witness)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (not (< (apply-m disk d2 q2 "mbal")
                        (apply-m dblock p2 "mbal")))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable phase1or2read))))

(defthm disk-helper-maybe
 (implies (and (phase1or2read p2 d2 q2 b-witness)
               (< (APPLY-M DISK D2 Q2 "mbal")
                  (APPLY-M DBLOCK P2 "mbal"))
               (not (= d2 d)))
         (= (apply (apply blocksread-n p) d)
            (apply (apply blocksread   p) d)))
    :hints (("Goal" :in-theory (enable phase1or2read)
             :cases ((= p2 p))))))
```

```
(defthm disk-helper1
 (implies (and (phase1or2read p d2 q2 b-witness)
               (< (APPLY-M DISK D2 Q2 "mbal")
                  (APPLY-M DBLOCK P "mbal"))
               (not (= d2 d)))
          (= (apply (apply blocksread-n q) d)
             (apply (apply blocksread   q) d)))
     :hints (("Goal" :in-theory (enable phase1or2read))))

(defthm disk-helper2
  (implies (and (phase1or2read p d2 q2 b-witness)
                (< (APPLY-M DISK D2 Q2 "mbal")
                   (APPLY-M DBLOCK P "mbal"))
                (not (= d2 d)))
            (iff (hasread p d q blocksread-n)
                 (hasread p d q blocksread)))
  :hints (("Goal" :in-theory (enable hasread))))

(defthm disk-helper3
  (implies (and (phase1or2read p2 d2 q2 b-witness)
                 (< (APPLY-M DISK D2 Q2 "mbal")
                    (APPLY-M DBLOCK P2 "mbal"))
                 (not (= d2 d)))
           (iff (hinv3.l phase-n p d q blocksread-n)
                (hinv3.l phase p d q blocksread)))
  :hints (("Goal" :in-theory (enable hinv3.l hasread phase1or2read))))

(defthm disk-helper4
  (implies (and (phase1or2read p2 d2 q2 b-witness)
                 (< (APPLY-M DISK D2 Q2 "mbal")
                    (APPLY-M DBLOCK P2 "mbal"))
                 (not (= d2 d)))
           (iff (hinv3.r dblock-n blocksread-n p d q)
                (hinv3.r dblock blocksread p d q)))
  :hints (("Goal" :in-theory (enable hinv3.r hasread phase1or2read))))

)) ; end local progn

(defthm p12r-d2not=d

  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (not (= d2 d))
```

```
               (hinv3.l phase-n p d q blocksread-n))
          (hinv3.r dblock-n blocksread-n p d q))

  :hints (("Goal" :cases ((< (APPLY-M DISK D2 Q2 "mbal")
                             (APPLY-M DBLOCK P2 "mbal"))))
          ("Subgoal 2" :in-theory (enable phase1or2read)))
  :otf-flg t)

; End of file ------------------------------------------------------------------
```

D.26 i2c/p12r-p2=p.lisp

```
; p12r-d2=d.lisp

(in-package "S")

; (ld "defpkg.lisp")
; (certify-book "p12r-p2=p" 1)

(include-book "common-i2c")
(include-book "common-p12r")
(include-book "startballot")
(include-book "p12r-pdp")
(include-book "p12r-pdq")
(include-book "p12r-q2not=q")

; case: (= q2 q) -- leslie's case
; --------------

; case (= p q)

(defthm p12r-p=q
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (= d2 d)
                (= p2 p)
                (= q2 q)
                (= p q)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q)))
```

```
; end case (= p q)

; case (not (= p q))


(defthm p12r-pnot=q
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (= d2 d)
                (= p2 p)
                (= q2 q)
                (not (= p q))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q)))

; end case (not (= p q))

(defthm p12r-q2=q
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (= d2 d)
                (= p2 p)
                (= q2 q)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :cases ((= p q)))))

; case: (not (= q2 q))
; --------------------

(defthm p12r-p2=p
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (= d2 d)
                (= p2 p)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :cases ((= q2 q)))))

; End of file ------------------------------------------------------------------
```

D.27 i2c/p12r-p2=q.lisp

```
; p12r-d2=d.lisp

(in-package "S")

; (ld "defpkg.lisp")
; (certify-book "p12r-p2=q" 1)

(include-book "common-i2c")
(include-book "common-p12r")
(include-book "startballot")
(include-book "p12r-p2=p")

(local
 (progn

   (defthm hinv3.l-p-q
     (equal (hinv3.l phase-n p d q blocksread-n)
            (hinv3.l phase-n q d p blocksread-n))
     :hints (("Goal" :in-theory (enable hinv3.l)))
     :rule-classes nil)

   (defthm hinv3.r-p-q
     (equal (hinv3.r dblock-n blocksread-n p d q)
            (hinv3.r dblock-n blocksread-n q d p))
     :hints (("Goal" :in-theory (enable hinv3.r))))

   (defthm p12r-p2=q-0
     (implies (and (hinv1) (hinv2) (hinv3)
                   (mem p (proc)) (mem q (proc)) (mem d (disk))
                   (phase1or2read p2 d2 q2 b-witness)
                   (= d2 d)
                   (= p2 q)
                   (hinv3.l phase-n q d p blocksread-n))
              (hinv3.r dblock-n blocksread-n q d p))
     :hints (("Goal" :by p12r-p2=p)))

   )) ; end local progn

(defthm p12r-p2=q
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
```

```
               (= d2 d)
               (= p2 q)
               (hinv3.l phase-n p d q blocksread-n))
          (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :use (hinv3.r-p-q hinv3.l-p-q p12r-p2=q-0)
           :in-theory nil)))

; End of file ------------------------------------------------------------
```

D.28 i2c/p12r-pdp.lisp

```
; case: phase1or2read(p,d,q), p=q
; ------------------------------

(in-package "S")

; I used the following command to certify this book:
; (ld "defpkg.lisp")
; (certify-book "p12r-pdp" 1)


(include-book "common-i2c")
(include-book "common-p12r")
(include-book "startballot")
(include-book "../hinv2-exports")

(local
 (progn

(defthm xxx4
  (implies (and (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (phase1or2read p d q b-witness)
                (< (apply-m disk d q "mbal")
                   (apply-m dblock p "mbal"))
                (= (apply-m disk d q) (apply dblock q)))
           (hinv3.r dblock-n blocksread-n p d q))
                    :hints (("goal" :in-theory (enable phase1or2read hinv3.r))))

(defthm xxx5
  (implies (and (hinv2)
```

```
               (= p q)
               (mem p (proc))
               (mem q (proc))
               (mem d (disk))
               (phase1or2read p d q b-witness)
               (< (apply-m disk d q "mbal")
                  (apply-m dblock p "mbal")))
          (hinv3.r dblock-n blocksread-n p d q))
                    :hints (("goal" :in-theory (enable phase1or2read))))

)) ; end local progn

(defthm p12r-pdp.lisp
  (implies (and (hinv1) (hinv2) (hinv3)
               (= p q)
               (mem p (proc))
               (mem q (proc))
               (mem d (disk))
               (phase1or2read p d q b-witness)
               (hinv3.l phase-n p d q blocksread-n))
          (hinv3.r dblock-n blocksread-n p d q))
                    :hints (("Goal" :cases ((< (apply-m disk d q "mbal")
                                               (apply-m dblock p "mbal")))))))

; End of file ------------------------------------------------------------------
```

D.29 i2c/p12r-pdq.lisp

```
; case: phase1or2read(p,d,q), p#q
; -------------------------------

(in-package "S")

; (ld "defpkg.lisp")
; (certify-book "p12r-pdq" 1)

(include-book "common-i2c")
(include-book "common-p12r")
(include-book "../hinv2-exports")

(local
 (progn
```

```
; Now I show that the theorem follows if the if branch is taken.

(defthm hasread-stays-hasread
  (implies (and (phase1or2read p d q b-witness)
                (< (apply-m disk d q "mbal") (apply-m dblock p "mbal"))
                (not (= p q))
                (hasread q d p blocksread-n))
           (hasread q d p blocksread))
  :hints (("Goal" :in-theory (enable hasread phase1or2read))))

(defthm phase-stays-phase
  (implies (and (phase1or2read p d q b-witness)
                (< (apply-m disk d q "mbal") (apply-m dblock p "mbal"))
                (not (= p q))
                (mem (apply phase-n q) (brace 1 2)))
           (mem (apply phase q) (brace 1 2)))
  :hints (("Goal" :in-theory (enable phase1or2read))))

(defthm phase-stays-phase-with-hide
  (implies (and (phase1or2read p d q b-witness)
                (< (apply-m disk d q "mbal") (apply-m dblock p "mbal"))
                (not (= p q))
                (mem (apply phase-n q) (hide (brace 1 2))))
           (mem (apply phase q) (hide (brace 1 2))))
  :hints (("Goal" :use phase-stays-phase
           :expand ((hide (brace 1 2)))
           :in-theory nil)))

(defthm hasread-blocksread-q-d-not-nil
  (implies (hasread q d p blocksread)
           (not (= (apply-m blocksread q d) nil)))
  :hints (("Goal" :in-theory (enable hasread))))

(defthm xxx4
  (implies (and (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (phase1or2read p d q b-witness)
                (< (apply-m disk d q "mbal")
                   (apply-m dblock p "mbal"))
                (= (apply-m disk d q) (apply dblock q)))
           (hinv3.r dblock-n blocksread-n p d q))
                    :hints
                    (("goal" :in-theory
```

```
                            (enable phase1or2read hinv3.r))))

(in-theory (disable hinv2))

; at this point, i have everything i need to prove this theorem and i
; can do so with the proofchecker. i just need to figure out what goes
; wrong with acl2.

(defthm xxx5
        (implies (and (hinv2)
                      (mem p (proc))
                      (mem q (proc))
                      (mem d (disk))
                      (phase1or2read p d q b-witness)
                      (not (= p q))
                      (< (apply-m disk d q "mbal")
                         (apply-m dblock p "mbal"))
                      (hinv3.l phase-n p d q blocksread-n))
                 (hinv3.r dblock-n blocksread-n p d q))
        :instructions
        (:bash (:rewrite xxx4)
               :bash :bash (:dv 1)
               (:rewrite hinv2-lemma)
               :top :bash :bash (:rewrite hinv2-lemma2)
               :bash (:change-goal nil t)
               (:dv 1)
               (:rewrite hasread-blocksread-q-d-not-nil)
               :demote (:dv 1 8)
               :expand :top :bash
               (:rewrite phase-stays-phase-with-hide)
               :bash :bash :bash :demote (:dv 1 8)
               :expand :top :bash))

)) ;end local progn

(defthm p12r-pdq
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p d q b-witness)
                (not (= p q))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :cases ((< (apply-m disk d q "mbal") (apply-m dblock p "mbal"))))))

; End of file ------------------------------------------------------------------
```

D.30 i2c/p12r-pdx.lisp

```
; p12r-pdx.lisp

(in-package "S")

; I used the following command to certify this book:
; (ld "defpkg.lisp")
; (certify-book "p12r-pdx" 1)

(include-book "common-i2c")
(include-book "common-p12r")


(local
 (progn

   (newdefmap exists-hasread-map2 (blocksread-p-d q)
              :for br :in blocksread-p-d :such-that (= (apply br "proc") q))

   (defun exists-hasread2 (blocksread-p-d q)
     (not (ur-elementp (exists-hasread-map2 blocksread-p-d q))))

   (defun hasread2 (p d q blocksread)
     (exists-hasread2 (apply-m blocksread p d) q))

   (in-theory (enable exists-hasread-map2))

   (defthm z
     (implies (not (= q2 q))
              (ur-elementp (exists-hasread-map2
                            (scons (func ("block" (apply-m disk d q))
                                         ("proc" q))
                                   nil)
                            q2))))

   (defthm one
     (implies (and (= blocksread-n
                      (except-m blocksread p d
                                (union (apply-m blocksread p d)
                                       (brace (func ("block" (apply-m disk d q))
```

```
                                                ("proc" q))))))
                (not (= q2 q))
                (hasread2 p d q2 blocksread-n))
           (hasread2 p d q2 blocksread)))

(defthm two
  (implies (and (= blocksread-n
                   (except-m blocksread p d
                             (union (apply-m blocksread p d)
                                    (brace (func ("block" (apply-m disk d q))
                                                 ("proc" q))))))
                (not (= q2 p))
                (hasread2 q2 d p blocksread-n))
           (hasread2 q2 d p blocksread)))

(defun hinv3.l2 (phase p d q blocksread)
  (and (mem (apply phase p) (hide (brace 1 2)))
       (mem (apply phase q) (hide (brace 1 2)))
       (hasread2 p d q blocksread)
       (hasread2 q d p blocksread)))

(defthm three
  (implies (and (phase1or2read p2 d2 q2 b-witness)
                (< (apply-m disk d2 q2 "mbal")
                   (apply-m dblock p2 "mbal"))
                (not (= q2 q))
                (not (= q2 p))
                (hinv3.l2 phase-n p d q blocksread-n))
           (hinv3.l2 phase p d q blocksread))
  :hints (("Goal" :in-theory (enable phase1or2read))))

(defthm exists-hasread-map2=exists-hasread-map
  (equal (exists-hasread-map2 dom q)
         (exists-hasread-map  dom q))
  :hints (("Goal" :in-theory (enable exists-hasread-map))))

(defthm hasread2=hasread
  (equal (hasread2 p d q blocksread)
         (hasread  p d q blocksread))
  :hints (("Goal" :in-theory (enable hasread exists-hasread))))

(defthm hinv3.l2=hinv3.l
  (equal (hinv3.l2 phase p d q blocksread)
         (hinv3.l  phase p d q blocksread))
  :hints (("Goal" :in-theory (enable hinv3.l))))
```

```
(defthm four
  (implies (and (MEM (FUNC ("block" (APPLY DBLOCK Q))
                           ("proc" Q))
                     (APPLY-M BLOCKSREAD P D))
                (phase1or2read p d q2 b-witness)
                (< (apply-m disk d q2 "mbal")
                   (apply-m dblock p "mbal")))
           (MEM (FUNC ("block" (APPLY DBLOCK-n Q))
                      ("proc" Q))
                (APPLY-M BLOCKSREAD-n P D)))
  :hints (("Goal" :in-theory (enable phase1or2read))))

(defthm five
  (implies (and (MEM (FUNC ("block" (APPLY DBLOCK P))
                           ("proc" P))
                     (APPLY-M BLOCKSREAD Q D))
                (phase1or2read p d q2 b-witness)
                (< (apply-m disk d q2 "mbal")
                   (apply-m dblock p "mbal")))
           (MEM (FUNC ("block" (APPLY DBLOCK-n P))
                      ("proc" P))
                (APPLY-M BLOCKSREAD-n Q D)))
  :hints (("Goal" :in-theory (enable phase1or2read))))

(defthm six
  (implies (and (hinv3.r dblock blocksread p d q)
                (phase1or2read p d q2 b-witness)
                (< (apply-m disk d q2 "mbal")
                   (apply-m dblock p "mbal")))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable phase1or2read hinv3.r))))

(defthm seven
  (implies (and (phase1or2read p2 d2 q2 b-witness)
                (< (apply-m disk d2 q2 "mbal")
                   (apply-m dblock p2 "mbal"))
                (not (= q2 q))
                (not (= q2 p))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.l phase p d q blocksread))
  :hints (("Goal" :use three)))

(defthm eight
  (implies (and (hinv3)
```

```
                  (mem p (proc)) (mem q (proc)) (mem d (disk))
                  (phase1or2read p d q2 b-witness)
                  (< (apply-m disk d q2 "mbal")
                     (apply-m dblock p "mbal"))
                  (not (= q2 q))
                  (not (= q2 p))
                  (hinv3.l phase-n p d q blocksread-n))
             (hinv3.r dblock-n blocksread-n p d q))
    :hints (("Goal" :use hinv3-implies-particular
             :in-theory (disable hinv3-implies-particular))))

   )) ; end local progn

(defthm p12r-pdx
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p d q2 b-witness)
                (not (= q2 q))
                (not (= q2 p))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))

  :hints (("Goal" :cases ((< (apply-m disk d q2 "mbal")
                             (apply-m dblock p "mbal"))))))

; End of file ------------------------------------------------------------------
```

D.31 i2c/p12r-q2=p-2.lisp

```
; p12r-q2=p-2.lisp

(in-package "S")

; (ld "defpkg.lisp")
; (certify-book "p12r-q2=p-2" 1)

(include-book "common-i2c")
(include-book "common-p12r")
(include-book "startballot")
(include-book "p12r-pdx")
(include-book "p12r-pdp")
```

```
(local
 (progn

   (newdefmap exists-hasread-map2 (blocksread-p-d q)
              :for br :in blocksread-p-d :such-that (= (apply br "proc") q))

   (defun exists-hasread2 (blocksread-p-d q)
     (not (ur-elementp (exists-hasread-map2 blocksread-p-d q))))

   (defun hasread2 (p d q blocksread)
     (exists-hasread2 (apply-m blocksread p d) q))

   (in-theory (enable exists-hasread-map2))

   (defthm z
     (implies (not (= q2 q))
              (ur-elementp (exists-hasread-map2
                            (scons (func ("block" (apply-m disk d q))
                                         ("proc" q))
                                   nil)
                            q2))))

   (defthm one
     (implies (and (= blocksread-n
                      (except-m blocksread p d
                                (union (apply-m blocksread p d)
                                       (brace (func ("block" (apply-m disk d q))
                                                    ("proc" q))))))
                   (not (= q2 q))
                   (hasread2 p d q2 blocksread-n))
              (hasread2 p d q2 blocksread)))

   (defthm two
     (implies (and (= blocksread-n
                      (except-m blocksread p d
                                (union (apply-m blocksread p d)
                                       (brace (func ("block" (apply-m disk d q))
                                                    ("proc" q))))))
                   (not (= q2 p))
                   (hasread2 q2 d p blocksread-n))
              (hasread2 q2 d p blocksread)))

   (defun hinv3.l2 (phase p d q blocksread)
     (and (mem (apply phase p) (hide (brace 1 2)))
          (mem (apply phase q) (hide (brace 1 2)))
```

```
        (hasread2 p d q blocksread)
        (hasread2 q d p blocksread)))

(defthm three
  (implies (and (phase1or2read p2 d2 q2 b-witness)
                (< (apply-m disk d2 q2 "mbal")
                   (apply-m dblock p2 "mbal"))
                (= d2 d)
                (= p2 p)
                (= q2 p)
                (not (= q2 q))
                (not (= p q))
                (hinv3.l2 phase-n p d q blocksread-n))
           (hinv3.l2 phase p d q blocksread))
  :hints (("Goal" :in-theory (enable phase1or2read))))

(defthm exists-hasread-map2=exists-hasread-map
  (equal (exists-hasread-map2 dom q)
         (exists-hasread-map  dom q))
  :hints (("Goal" :in-theory (enable exists-hasread-map))))

(defthm hasread2=hasread
  (equal (hasread2 p d q blocksread)
         (hasread  p d q blocksread))
  :hints (("Goal" :in-theory (enable hasread exists-hasread))))

(defthm hinv3.l2=hinv3.l
  (equal (hinv3.l2 phase p d q blocksread)
         (hinv3.l  phase p d q blocksread))
  :hints (("Goal" :in-theory (enable hinv3.l))))

(defthm four
  (implies (and (MEM (FUNC ("block" (APPLY DBLOCK Q))
                           ("proc" Q))
                     (APPLY-M BLOCKSREAD P D))
                (phase1or2read p d q2 b-witness)
                (< (apply-m disk d q2 "mbal")
                   (apply-m dblock p "mbal")))
           (MEM (FUNC ("block" (APPLY DBLOCK-n Q))
                      ("proc" Q))
                (APPLY-M BLOCKSREAD-n P D)))
  :hints (("Goal" :in-theory (enable phase1or2read))))

(defthm five
  (implies (and (MEM (FUNC ("block" (APPLY DBLOCK P))
```

```
                              ("proc" P))
                         (APPLY-M BLOCKSREAD Q D))
                   (phase1or2read p d q2 b-witness)
                   (< (apply-m disk d q2 "mbal")
                      (apply-m dblock p "mbal")))
             (MEM (FUNC ("block" (APPLY DBLOCK-n P))
                        ("proc" P))
                  (APPLY-M BLOCKSREAD-n Q D)))
    :hints (("Goal" :in-theory (enable phase1or2read))))

  (defthm six
    (implies (and (hinv3.r dblock blocksread p d q)
                  (phase1or2read p d q2 b-witness)
                  (< (apply-m disk d q2 "mbal")
                     (apply-m dblock p "mbal")))
             (hinv3.r dblock-n blocksread-n p d q))
    :hints (("Goal" :in-theory (enable phase1or2read hinv3.r))))

  (defthm seven
    (implies (and (phase1or2read p2 d2 q2 b-witness)
                  (< (apply-m disk d2 q2 "mbal")
                     (apply-m dblock p2 "mbal"))
                  (= d2 d)
                  (= p2 p)
                  (= q2 p)
                  (not (= q2 q))
                  (not (= p q))
                  (hinv3.l phase-n p d q blocksread-n))
             (hinv3.l phase p d q blocksread))
    :hints (("Goal" :use three)))

  (defthm eight
    (implies (and (hinv3)
                  (mem p (proc)) (mem q (proc)) (mem d (disk))
                  (phase1or2read p d q2 b-witness)
                  (< (apply-m disk d q2 "mbal")
                     (apply-m dblock p "mbal"))
;                 (= d2 d)
;                 (= p2 p)
                  (= q2 p)
                  (not (= q2 q))
                  (not (= p q))
                  (hinv3.l phase-n p d q blocksread-n))
             (hinv3.r dblock-n blocksread-n p d q))
    :hints (("Goal" :use hinv3-implies-particular
```

```
             :in-theory (disable hinv3-implies-particular))))


)) ;end local progn

(defthm p12r-q2=p-2
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (= d2 d)
                (= p2 p)
                (= q2 p)
                (not (= q2 q))
                (not (= p q))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))

  :hints (("Goal" :cases ((< (apply-m disk d q2 "mbal")
                             (apply-m dblock p "mbal"))))))


; End of file ------------------------------------------------------------------
```

D.32 i2c/p12r-q2not=q.lisp

```
; p12r-q2not=q.lisp

(in-package "S")

; (ld "defpkg.lisp")
; (certify-book "p12r-q2not=q" 1)

(include-book "common-i2c")
(include-book "common-p12r")
(include-book "startballot")
(include-book "p12r-pdx")
(include-book "p12r-pdp")
(include-book "p12r-q2=p-2")

; case: (= q2 p)
; --------------
```

```
; case: (= p q) -- then we're back to p12r(p,d,p)

(local
 (progn

(defthm p12r-q2=p
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (= d2 d)
                (= p2 p)
                (= q2 p)
                (= p q)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q)))

; case: (not (= p q))

(defthm p12r-q2=p-2
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (= d2 d)
                (= p2 p)
                (= q2 p)
                (not (= q2 q))
                (not (= p q))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q)))

; case: (not (= q2 p))
; --------------------

(defthm p12r-q2not=q-3
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (= d2 d)
                (= p2 p)
                (not (= q2 q))
                (not (= q2 p))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q)))

)) ; end local progn
```

```
(defthm p12r-q2not=q
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (= d2 d)
                (= p2 p)
                (not (= q2 q))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :cases ((= q2 p)))))

; End of file ------------------------------------------------------------------
```

D.33 i2c/p12r.lisp

```
; p12r.lisp

(in-package "S")

; (ld "defpkg.lisp")
; (certify-book "p12r" 1)

; case: (= d2 d)
; --------------

(include-book "p12r-d2=d")

; case: (not (= d2 d))
; --------------------

(include-book "p12r-d2not=d")

; p12.lisp
; --------

(defthm p12r
  (implies (and (hinv1) (hinv2) (hinv3)
                (mem p (proc)) (mem q (proc)) (mem d (disk))
                (phase1or2read p2 d2 q2 b-witness)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
```

```
  :hints (("Goal" :cases ((= d2 d)))))

; End of file ------------------------------------------------------------------
```

D.34 i2c/p12w.lisp

```
; p12w.lisp

(in-package "S")

; (ld "defpkg.lisp")
; (certify-book "p12w" 1)

(include-book "common-i2c")

(local
 (progn

(defthm phase1or2write-hasread-3
  (implies (and (not (hasread p d q blocksread))
                (phase1or2write p2 d2))
           (not (hasread p d q blocksread-n)))
  :hints (("Goal" :in-theory (enable phase1or2write))))

(defthm phase1or2write-hasread-4
  (implies (and (not (mem (apply phase p) (hide (brace 1 2))))
                (phase1or2write p2 d2))
           (not (mem (apply phase-n p) (hide (brace 1 2)))))
  :hints (("Goal" :in-theory (enable phase1or2write))))

(defthm wb-phase1or2write
  (implies (and (phase1or2write p2 d2)
                (not (= p2 p)))
           (and (= (apply input-n p) (apply input p))
                (= (apply output-n p) (apply output p))
                (= (apply-m disk-n d p) (apply-m disk d p))
                (= (apply phase-n p) (apply phase p))
                (= (apply dblock-n p) (apply dblock p))
                (= (apply diskswritten-n p) (apply diskswritten p))
                (= (apply-m blocksread-n p d) (apply-m blocksread p d))
                (= (apply blocksread-n p) (apply blocksread p))))
  :hints (("Goal" :in-theory (enable phase1or2write))))
```

```
(defthm wb-phase1or2write-2
  (implies (phase1or2write p2 d2)
           (and (= (apply input-n p) (apply input p))
                (= (apply output-n p) (apply output p))
                (= (apply phase-n p) (apply phase p))
                (= (apply dblock-n p) (apply dblock p))
                (= (apply-m blocksread-n p d) (apply-m blocksread p d))
                (= (apply blocksread-n p) (apply blocksread p))))
  :hints (("Goal" :in-theory (enable phase1or2write))))

(defthm i2c-p12w-1
  (implies (and (hinv1) (hinv2) (hinv3) (phase1or2write p2 d2)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk)))
           (iff (hasread p d q blocksread-n)
                (hasread p d q blocksread)))
  :hints (("Goal" :in-theory (enable hasread))))

(defthm i2c-p12w-3
  (implies (and (hinv1) (hinv2) (hinv3) (phase1or2write p2 d2)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk)))
           (iff (hinv3.l phase-n p d q blocksread-n)
                (hinv3.l phase p d q blocksread)))
  :hints (("Goal" :in-theory (enable hinv3.l))))

(defthm i2c-p12w-4
  (implies (and (hinv1) (hinv2) (hinv3) (phase1or2write p2 d2)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk)))
           (iff (hinv3.r dblock-n blocksread-n p d q)
                (hinv3.r dblock blocksread p d q)))
  :hints (("Goal" :in-theory (enable hinv3.r))))

)) ; end local progn

(defthm i2c-p12w
  (implies (and (hinv1) (hinv2) (hinv3) (phase1or2write p2 d2)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
```

```
               (hinv3.l phase-n p d q blocksread-n))
          (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :cases ((and (not (= p2 p)) (not (= p2 q)))))))

; End of file ------------------------------------------------------------------
```

D.35 i2c/startballot.lisp

```
; startballot.lisp

; I used the following command to certify this book:
; (ld "defpkg.lisp")
; (certify-book "startballot" 1)


(in-package "S")


(include-book "common-i2c")

(local
 (progn

   (defthm startballot-hasread
     (implies (and (startballot p b-witness)
                   (= p2 p))
              (not (hasread p2 d q blocksread-n)))
     :hints (("Goal" :in-theory (enable startballot))))

   (defthm startballot-hasread-2
     (implies (startballot p2 b-witness)
              (not (hasread p2 d q blocksread-n)))
     :hints (("Goal" :in-theory (enable startballot))))

   (defthm wb-startballot
     (implies (and (startballot p2 b-witness)
                   (not (= p2 p)))
              (and (= (apply input-n p) (apply input p))
                   (= (apply output-n p) (apply output p))
                   (= (apply-m disk-n d p) (apply-m disk d p))
                   (= (apply phase-n p) (apply phase p))
                   (= (apply dblock-n p) (apply dblock p))
```

```
                (= (apply diskswritten-n p) (apply diskswritten p))
                (= (apply-m blocksread-n p d) (apply-m blocksread p d))
                (= (apply blocksread-n p) (apply blocksread p))))
  :hints (("Goal" :in-theory (enable startballot))))

(in-theory (disable startballot))

(defthm i2c-startballot-0
  (implies (and (hinv1) (hinv2) (hinv3) (startballot p2 b-witness)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (= p2 p)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable hinv3.l hinv3.r))))

(defthm i2c-startballot-1
  (implies (and (hinv1) (hinv2) (hinv3) (startballot p2 b-witness)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (= p2 q)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable hinv3.l hinv3.r))))

(defthm i2c-startballot-2
  (implies (and (hinv1) (hinv2) (hinv3) (startballot p2 b-witness)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hasread p d q blocksread-n)
                (hasread p d q blocksread)))
  :hints (("Goal" :in-theory (enable hasread))))

                                    ; this is not needed
(defthm i2c-startballot-3
  (implies (and (hinv1) (hinv2) (hinv3) (startballot p2 b-witness)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
```

```
                (not (= p2 q)))
           (iff (hasread q d p blocksread-n)
                (hasread q d p blocksread)))
  :hints (("Goal" :in-theory (enable hasread))))

(in-theory (disable hinv1 hinv3))

(defthm i2c-startballot-4
  (implies (and (hinv1) (hinv2) (hinv3) (startballot p2 b-witness)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hinv3.l phase-n p d q blocksread-n)
                (hinv3.l phase p d q blocksread)))
  :hints (("Goal" :in-theory (enable hinv3.l))))

(defthm i2c-startballot-5
  (implies (and (hinv1) (hinv2) (hinv3) (startballot p2 b-witness)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (not (= p2 p))
                (not (= p2 q)))
           (iff (hinv3.r dblock-n blocksread-n p d q)
                (hinv3.r dblock blocksread p d q)))
  :hints (("Goal" :in-theory (enable hinv3.r))))


(defthm i2c-startballot-6
  (implies (and (hinv1) (hinv2) (hinv3) (startballot p2 b-witness)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (= p2 p)
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :in-theory (enable hinv3.r hinv3.l))))

(defthm i2c-startballot-7
  (implies (and (hinv1) (hinv2) (hinv3) (startballot p2 b-witness)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
```

```
                  (= p2 q)
                  (hinv3.l phase-n p d q blocksread-n))
              (hinv3.r dblock-n blocksread-n p d q))
     :hints (("Goal" :in-theory (enable hinv3.r hinv3.l))))

   )) ; end local progn

(defthm i2c-startballot
  (implies (and (hinv1) (hinv2) (hinv3) (startballot p2 b-witness)
                (mem p (proc))
                (mem q (proc))
                (mem d (disk))
                (hinv3.l phase-n p d q blocksread-n))
           (hinv3.r dblock-n blocksread-n p d q))
  :hints (("Goal" :cases ((= p2 p) (= p2 q))))
  :otf-flg t)

; End of file ------------------------------------------------------------------
```