RL-TR-95-295, Vol III (of four)
Final Technical Report
April 1996

ROMULUS, A COMPUTER SECURITY PROPERTIES MODELING ENVIRONMENT: ROMULUS LIBRARY OF MODELS

Odyssey Research Associates, Inc.

S. Brackin, S. Foley, L. Gong, B. Hartman, A. Heff, G. Hird, D. Long, D. McCullough, I. Meisels, D. Rosenthal, I.Sutherland, and A. Weitzman

Rome Laboratory
Air Force Materiel Command
Rome, New York

RL-TR- 95-295, Vol III(of four), has been reviewed and is approved for publication.

APPROVED: John C. Faust

JOHN C. FAUST
Project Engineer

FOR THE COMMANDER: [illegible]

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE April 1996	3. REPORT TYPE AND DATES COVERED Final Aug 90 - Jun 94

4. TITLE AND SUBTITLE ROMULUS, A COMPUTER SECURITY PROPERTIES MODEL ENVIRONMENT: Romulus Library of Models	5. FUNDING NUMBERS C - F30602-90-C-0092 PE - 35167G PR - 1065 TA - 01 WU - 03
6. AUTHOR(S) S. Brackin, S. Foley, L. Gong, B. Hartman, A. Heff, G. Hird, D. Long, D. McCullough, I. Meisels, D. Rosenthal, I. Sutherland, and A. Weitzman	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Odyssey Research Associates, Inc. 301 Dates Drive Ithaca NY 14850-1326	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/C3AB 525 Brooks Rd Rome NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-95-295, Vol III (of four)
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: John C. Faust/C3AB/(315) 330-3241	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Romulus security properties modeling environment contains tools, theories, and models that support the high-level design and analysis of secure systems.

The Romulus nondisclosure tool supports development and analysis of distributed composite security models and their properties. The Romulus modeling approach establishes the models on a solid theoretical basis and uses formal mathematical tools to aid in the analysis. Romulus allows a user to express a model of a secure system using a formal specification notation that combines graphics and text. Verification of the model proves that it satisfies its critical properties. The user verifies the model by using a combination of automatic decision procedures and interactive theorem proving. The primary emphasis in the current system is the analysis of multilevel trusted system models to see if they satisfy nondisclosure properties. Romulus also includes a tool for formally specifying and verifying authentication protocols. This tool can be used to reason about the beliefs of the parties engaged in a protocol in order to analyze whether the protocol achieves the desired behavior. The Romulus theories include formal theories of nondisclosure, integrity, and (see reverse)

14. SUBJECT TERMS Computer security, Nondisclosure, Integrity, Availability, Security properties modeling, Information flow analysis, Design verification, Authentication protocol analysis, (see reverse)	15. NUMBER OF PAGES 280 16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

13. (Cont'd)

availability security. The Romulus library of models demonstrates the application of these theories.

14. (Cont'd)

Multilevel security, Security policy

Preface

This four volume report describes Romulus, a security modeling environment. Romulus includes a tool for constructing graphical hierarchical process representations; an information flow analyzer; a process specification language; and techniques to aid in doing proofs of security properties. Romulus also contains tools for the specification and analysis of authentication protocols. Using Romulus, a user can develop and analyze security models and properties. The foundations of Romulus are formal theories of security; applications of these theories are demonstrated in a library of models.

In this volume, we assume that the reader has some familiarity with the Romulus tools, the HOL system, and security issues in general.

Organization of the Romulus Documentation Set

Volume I of this documentation set is an overview of Romulus. Volume II describes the Romulus theories of nondisclosure, integrity, and availability. This is Volume III; it describes the Romulus library of models. Volume IV is the Romulus User's Manual; it contains descriptions of the Romulus tools, how to use them, and tutorial examples.

Organization of This Volume

This volume is the library of models. It provides a collection of examples that show how to build and analyze models for security properties. The examples described in this volume are: a Generic Guard Model, a Secure Minix Model, a Network Driver Model, Authentication Protocol Models, a Multilevel Distributed Database Integrity Model, a The Fault Tolerant Reference Monitor Model, and a Real-Time Scheduling Model.

Conventions

This document set uses the following conventions. Computer code, specifications, program names, file names, and similar material are typeset using a `typewriter` font. Interactive computer sessions are surrounded by a rounded box. Within this box, user input is typeset using an *`italic typewriter`* font; computer output is typeset using the `typewriter` font. Some computer output has been reformatted for presentation purposes; it may not appear in this document exactly as it appears on your screen.

Acknowledgements

Romulus has been in development for several years. We wish to acknowledge the contributions of everyone who has worked on this project. Portions of this volume were extracted from previous ORA reports and course materials, specifically, [19], [20], and [21].

Contents

List of Figures

Chapter 1

Introduction

The Romulus library of models contains a collection of examples that provide insight into how to build and analyze models for security properties. The library includes examples from the areas of nondisclosure, integrity, and availability. These properties are described in Volume II of the Romulus Documentation Set. The examples are

- an abstract guard (nondisclosure),
- Minix -- an operating system (nondisclosure),
- a network driver (nondisclosure and integrity),
- authentication protocols (integrity),
- a distributed database (integrity),
- the Fault Tolerant Reference Monitor (nondisclosure and availability), and
- a real-time scheduler (availability).

Because the areas of nondisclosure, integrity, and availability are so broad, our models address specific aspects of each area. For nondisclosure the focus is on mandatory access control, and in particular, the restrictiveness theory. For integrity, the models handle label integrity (network driver), correctness of distributed authentication, and a method of achieving serializability (distributed databases). For availability, the models address fault tolerance and real-time scheduling.

We give a brief description of each model below. We then provide a complete description in the following chapters.

Abstract Guard

The abstract guard is a generic description of a process that filters out messages that should not be delivered to processes operating in a given security range. Because it models the use of a shared resource (that is, shared directories), we use a variation on restrictiveness, called shared-state restrictiveness, to show that it is secure with respect to nondisclosure.

Minix

The Minix model is a specification of an MLS operating system based on the Minix operating system created by Tanenbaum. The model includes variations of Tanenbaum's kernel, memory management, and file system adapted to handle Mandatory Access Control. This model shows that it is feasible to formalize and prove restrictiveness of a complex system.

Network Driver

The network driver model describes a method for surfacing memory management attributes needed to ensure that the integrity of an MLS security level is being maintained. This integrity property is used in the security argument that the network labeler and network driver correctly set and maintain the security level. The model concentrates on an intuitive description of the methodology. We formalize relevant parts of the network labeler specification and its properties.

Authentication Protocols

The models here describe two separate authentication protocols. We analyze the Denning-Sacco protocol and the Needham-Schroeder protocol. The protocol messages are formally defined and specified. We formally state pre- and post-conditions for each protocol. We perform the analysis in the Romulus implementation of authentication logic. This logic is a belief logic, which enables us to express the belief-states of principals and prove that after protocol execution, the principals are operating with a correct set of beliefs about the identity of other principals and the adequacy of encryption keys. We discuss some interesting flaws exposed in the second example.

Distributed Database

The distributed database model specifies the trusted part of a protocol developed by Kogan and Jajodia for distributed, replicated databases; this protocol is both restrictive and one-copy serializable. The restrictiveness argument is straightforward, as the trusted protocol simply has to maintain and deliver messages to databases at particular levels. The serializability proof is not included here; it is contained in [11]. However, this model does contain a formal specification of how the trusted part of the protocol should be constructed.

Fault Tolerant Reference Monitor

The fault tolerant reference monitor (FTRM) is a model of a file access control mechanism, that is both secure with respect to storage channels (restrictive) and is fault tolerant. The main type of failure that is modeled is data corruption caused by disk failures. This failure is addressed by replicating the data. When a read or write request arrives, the request is propagated to the different servers and the retrieved information is compared. The result on which the majority of servers agree is used as the actual result. If there is a sufficient amount of replication and the number of faults at a given time is not too large, then this method provides the desired protection against these faults.

Real-Time Scheduler

This model explores some aspects of availability for real-time systems. In particular, it examines a part of the task scheduling loosely based on the operational flight program of the A-7E Navy Aircraft. This model shows how to formally augment the typical Romulus state machine models with timing to handle the analysis of these kinds of properties.

Chapter 2

The Shared-State Generic Guard Model

2.1 Introduction

This chapter gives a HOL90 formalization of the theory of *shared-state restrictiveness*, a mandatory access control property that simplifies the standard nondisclosure theory of restrictiveness [14, 15] and applies it in a natural way to processes involving shared state (e.g., processes communicating through shared directories). The chapter uses this theory to formalize security for a case naturally involving shared state, a generic guard program that chooses which messages can be released to a relatively untrustworthy entity and which are sent to an audit directory in order to be investigated. The chapter concludes with a formal proof that a guard is shared-state restrictive if its components satisfy a strong but simple security property.

For more information on HOL90, see Volume IV of the Romulus Documentation Set. For more information on shared-state restrictiveness, see [21] and Volume II of the Romulus Documentation Set.

The next few paragraphs list several properties of shared-state restrictiveness that indicate why we believe it to be of interest. These properties generally contrast it with ordinary restrictiveness.

Like restrictiveness, shared-state restrictiveness is a composable property, so that the result of properly connecting shared-state restrictive components is shared-state restrictive. Unlike restrictiveness, shared-state restrictiveness describes process behavior purely in terms of transformations on system variables rather than in terms of input and output events.

Shared-state restrictiveness distinguishes the system variables that can be changed by entities outside the system (*labeled* variables) from the system variables that can be changed only by the system itself (*unlabeled* variables). Input events in ordinary restrictiveness are analogous to changes in labeled variables made by entities outside the system; output events in restrictiveness are analogous to changes in labeled variables made by the system itself. Shared-state restrictiveness can hold, though, for systems in which "input" changes to system variables and "output" changes to system variables can occur simultaneously, and in which more than one system variable can change at a time.

Shared-state restrictiveness defines all sensitivity-level information for system variables in terms of a *security structure* projection function that captures only the information in system variables that is legitimately accessible to one having clearance at a particular level. This function subsumes all of ordinary restrictiveness' functions assigning security levels to input and output events and its projection functions capturing just the information in system variables needed to give the future system behavior visible at a particular level. See [21] for a proof that being shared-state restrictive guarantees that a system has appropriate nondisclosure properties.

Shared-state restrictiveness is a more natural tool for analyzing systems involving shared state than ordinary restrictiveness is. The simplest way of applying ordinary restrictiveness to systems involving shared state is to introduce new processes that manage the shared resources; this approach raises issues such as whether these new processes are always guaranteed to reply to requests and whether these new processes are themselves secure.

The generic guard process formalized in this chapter is very abstract, modeling virtually any program for analyzing messages and choosing which ones can be released. The results presented here could presumably be strengthened to show that specific guard programs are shared-state restrictive.

The "objects" for the generic guard are messages and bags of messages; a bag is an abstraction of a directory. The labeled variables for the guard model, which determine its external interfaces, are just three bags of messages: a bag for input messages, a bag for output messages, and a bag for messages that have been rejected for output and saved for auditing. The model's security structure assigns fixed security levels to each message. The guard design guarantees that only one message is accessed at a time and that if a new (or moved) message is created from an old one the new message has the same level as the old one.

We prove that a simple security property is sufficient to guarantee shared-state restrictiveness. An informal statement of this property follows: the guard's single-step behavior visible at a security level would be a possible system behavior if the system variables visible at that security level were the only ones that had ever existed. Although this property is rather strong, we believe it could be shown to follow from simpler and more specific properties of guard components — properties that would be easier to establish in themselves than shared-state or regular restrictiveness. These properties could then be used as guides by the designers of actual guards.

2.2 Shared-State Restrictiveness

This section contains and describes the specification of the general theory of shared-state restrictiveness.

2.2.1 Front Matter

The specification begins by removing any earlier versions of the theory of shared-state restrictiveness, creating a new theory of it, loading the HOL inductive definitions library, and loading a file of Romulus utilities that make it easier to make and use inductive definitions.

```
System.system "rm -f sharedstate.holsig sharedstate.thms";

new_theory "sharedstate";;

load_library{lib = Sys_lib.ind_def_lib, theory = "-"};
open Inductive_def;

use "romutils.sml";
```

2.2.2 Type Declarations

The specification next describes the type of the state of a system as it is viewed in the theory of shared-state restrictiveness. The state is a collection of variables with associated values, and the variables are divided into two categories: the *labeled* variables are those accessible to and possibly modifiable by agents outside the system itself, while the *unlabeled* variables are those internal to the system, accessible to it alone, and modifiable by it alone.

In the specification, the type variables `'labeled` and `'unlabeled` are intended to be instantiated with the types of records containing all the labeled and unlabeled variables, respectively, of an arbitrary system to be analyzed for being shared-state restrictive. These records are further intended to be records of the form defined by the Romulus type-defining utility `romrecord`, which not only defines the record types but also defines functions for accessing and updating the entries in these records; the specification given later for the generic guard model assumes that the functions for accessing and updating record entries follow the naming conventions used for these functions by `romrecord`.

Since type abbreviations are not yet implemented for HOL90 theories, the specification defines SML variables as abbreviated names for types, and then accesses these types with antiquotation.

```
val state = ty_antiq(==`:'labeled # 'unlabeled`==);
```

The specification also assumes that the type variable `'level` will be instantiated with the arbitrary type of security levels for whichever system is being analyzed.

2.2.3 Reachable States and Nondeterminism

The specification then defines those system states that can possibly be attained by inductively defining the predicate `reachable` as a function of the system's initial state and its possible state transitions.

A system's state transitions fall into one of two disjoint categories:

- An *outer* transition is one made by an agent external to the system being analyzed, an agent such as a person or another system. Typical external transitions are adding a file to an input directory or removing it from an output directory. An outer transition can affect only the labeled part of the system's state.

- An *inner* transition is one made by the system itself. An inner transition can change both the labeled and unlabeled parts of the system's state.

In the specification, a system's possible initial states and the possible outer transitions affecting it are given straightforwardly by predicates, but the description of a system's possible inner transitions is slightly more complicated. The inner transitions are given by a function whose first argument is an integer "nondeterminism argument" representing something like the time or the current value of the seed to a random number generator. The nondeterminism argument expresses the notion that although a system's actions need not be determined by its state variables, these "choices" cannot be made on the basis of information that is not available at all security levels, so they can be computed from knowledge of the state variables and the nondeterminism argument.

The `named_rules_new_inductive_definition`function used in the specification's definition of `reachable` is a Romulus utility that calls the standard HOL function `new_inductive_definition`, but allows explicit names to be associated with each rule for the inductive relation being defined. In the definition, the variables `initial`, `inner_trans`, and `outer_trans` denote an arbitrary predicate giving possible initial states, an arbitrary function of nondeterminism arguments and states giving inner transitions, and an arbitrary predicate giving possible outer transitions, respectively. The specification says simply that every initial state is reachable, that every state produced by an inner transition from a reachable state is reachable, and that every state produced by an outer transition from a reachable state is reachable.

```
val {desc = reachable_rules, induction_thm = reachable_ind} =
 let
  val reachable =
   --`reachable:
       (^state -> bool) ->             (* initial state *)
       (num -> ^state -> ^state) ->    (* inner transitions *)
       (^state -> ^state -> bool) ->   (* outer transitions *)
       ^state ->
       bool`--;
  val initial = --`initial:^state -> bool`--;
  val inner_trans = --`inner_trans:num -> ^state -> ^state`--;
  val outer_trans = --`outer_trans:^state -> ^state -> bool`--;
 in
```

```
   named_rules_new_inductive_definition
    "reachable"
    Prefix
    (--`^reachable initial inner_trans outer_trans state`--, [])
 [
 ("initial_reachable",
  {hypotheses = [],
   side_conditions = [--`^initial s`--],
   (*-------------------------------------------------------*)
   conclusion =
    (--`^reachable initial inner_trans outer_trans s`--)}),

 ("inner_reachable",
  {hypotheses =
    [--`^reachable initial inner_trans outer_trans s`--],
   side_conditions =
    [--`^inner_trans n s = s'`--],
   (*-------------------------------------------------------*)
   conclusion =
    (--`^reachable initial inner_trans outer_trans s'`--)}),

 ("outer_reachable",
  {hypotheses =
    [--`^reachable initial inner_trans outer_trans s`--],
   side_conditions =
    [--`^outer_trans s s'`--],
   (*-------------------------------------------------------*)
   conclusion =
    (--`^reachable initial inner_trans outer_trans s'`--)})
 ]
  end;
```

The specification calls the Romulus utility show_inductive_def_props to call HOL functions proving results that are often convenient when dealing with inductively defined predicates: a theorem giving a slightly strengthened form of the basic induction theorem, which restricts attention to cases in which the relation being defined actually holds; and a cases theorem that allows one to prove that an inductively defined relation holds if and only if it holds by virtue of one of the rules defining the relation.

```
show_inductive_def_props
 reachable_rules
 reachable_ind
 "reachable";
```

2.2.4 Extended Inner Transitions

The specification then defines an *extended* inner transition as the result of one or more inner transitions for some values of the nondeterminism argument. The specification uses `named_rules_new_inductive_definition` as before and again calls `show_inductive_def_props` to prove standard useful theorems about the relation defined.

```
val {desc = ext_inner_trans_rules,
     induction_thm = ext_inner_trans_ind} =
 let
  val ext_inner_trans =
   --`ext_inner_trans:
       (num -> ^state -> ^state) ->       (* inner transitions *)
       ^state ->
       ^state ->
       bool`--;
  val inner_trans = --`inner_trans:num -> ^state -> ^state`--;
 in
  named_rules_new_inductive_definition
   "ext_inner_trans"
   Prefix
   (--`^ext_inner_trans inner_trans s s'`--, [])
[
("inner_ext_inner",
 {hypotheses = [],
  side_conditions = [--`^inner_trans n s = s'`--],
  (*-----------------------------------------------------*)
  conclusion =
   (--`^ext_inner_trans inner_trans s s'`--)}),

("ext_ext_inner",
 {hypotheses =
   [--`^ext_inner_trans inner_trans s s'`--],
  side_conditions =
   [--`^inner_trans n s' = s''`--],
  (*-----------------------------------------------------*)
  conclusion =
   (--`^ext_inner_trans inner_trans s s''`--)})
]
 end;

show_inductive_def_props
 ext_inner_trans_rules
```

```
ext_inner_trans_ind
"ext_inner_trans";
```

2.2.5 Security Structures

A *security structure* is a projection function that takes a security level and a state (labeled, unlabeled, or total, where a total state is a labeled state together with an unlabeled state) and returns the state containing all, but only, the information in that state visible at that security level. The security structure on the labeled state corresponds to the level-assignment functions for input and output events in ordinary restrictiveness, while the security structure on the unlabeled state corresponds to the projection function in ordinary restrictiveness.

The specification defining shared-state restrictiveness does not assert anything about security structures on the labeled and unlabeled parts of a system's state, since these structures will be specific to the system being analyzed, but it does define a convenience function `totalss` that defines the projection to a level of a total state as the product of the projection's to that level of the total state's labeled and unlabeled parts.

```
new_definition(
 "totalss",
 let
  val totalss =
   --`totalss:
       'level ->
       ('level -> 'labeled -> 'labeled) ->
       ('level -> 'unlabeled -> 'unlabeled) ->
       ^state ->
       ^state`--;
 in
  --`^totalss lev lss uss (ls,us) = (lss lev ls, uss lev us)`--
 end);
```

2.2.6 Shared-State Restrictiveness

With these simple preliminaries out of the way, the specification then formally defines shared-state restrictiveness. A system given by a predicate `initial` that defines possible initial states, a function `inner_trans` mapping nondeterminism arguments and states to states that defines inner transitions, a predicate `outer_trans` that defines possible outer transitions, and a security structure `labeledss` on the system's labeled state variables is *shared-state restrictive* if there exists a security structure `unlabeledss` on the system's unlabeled state variables such that the following condition holds:

> For any two reachable states `s1` and `s2`, any nondeterminism argument `n`, and any security level `lev`, if the total security structures on `s1` and `s2` induced by `labeledss` and `unlabeledss` for level `lev` are equal, then any inner transition made on `s1` with nondeterminism argument `n` is to a state whose total security structure for level `lev` is the same as the total security structure for that level of some state obtained from `s2` by an arbitrary positive number of inner and outer transitions.

Very informally, if two states seem equivalent at a level and the system could do something in one of these states, whatever it does that is visible at that level could also have been the part visible at that level of some combination of actions the system or those acting on it could have taken in the other state.

In understanding the specification, note that the states `s1` and `s2` are given by the pairs `(ls1,us1)` and `(ls2,us2)`, respectively, showing the labeled and unlabeled parts of each state.

```
new_definition(
 "ssrestrictive",
 let
  val ssrestrictive =
   --`ssrestrictive:
        (^state -> bool) ->
        (num -> ^state -> ^state) ->
        (^state -> ^state -> bool) ->
        ('level -> 'labeled -> 'labeled) ->
        bool`--;
 in
  --`
   ^ssrestrictive initial inner_trans outer_trans labeledss =
     ?(unlabeledss:'level -> 'unlabeled -> 'unlabeled).
      !ls1 us1 ls2 us2 n lev.
       ((reachable initial inner_trans outer_trans (ls1,us1)) /\
        (reachable initial inner_trans outer_trans (ls2,us2)) /\
        (totalss lev labeledss unlabeledss (ls1,us1) =
          totalss lev labeledss unlabeledss (ls2,us2))) ==>
        (?ls2' us2'.
          (ext_inner_trans inner_trans (ls2,us2) (ls2',us2')) /\
          (totalss
             lev labeledss unlabeledss (inner_trans n (ls1,us1)) =
           totalss
             lev labeledss unlabeledss (ls2',us2')))
  '--
end);
```

2.2.7 Final Lines

The final lines of the specification normally write the theory ss of shared-state restrictiveness just created to the disk and cause HOL90 to exit. In this version of the specification, since the generic guard specification and proof follow, the specification does not cause HOL90 to exit.

```
export_theory();
```

2.3 Generic Guard Specification

This section gives the specification of a generic guard program that examines messages in an input directory and either accepts them as safe and passes them on to an output directory or rejects them as unsafe and sends them to an audit directory.

This specification uses the following naming conventions: if something is specific to a particular guard, its name begins with g_ for a function or G for a type; if it is specific only to any guard described in the generic way developed in this file, its name begins with gg_ for a function or GG for a type; if it is not specific to guards, its name begins with a letter other than g or G.

2.3.1 Initial Lines

The initial lines of the generic guard specification remove any older versions of the theory of generic guards, create a new theory gg, make the previous theory ss of shared-state restrictiveness one of its parent theories, and load the Romulus utility romrecord for creating record types with associated access and update functions. These lines normally load in the theory `ss` previously saved to disk and make it a new parent theory, but in this version that is not necessary, since HOL90 never exited.

```
System.system "rm -f gg.holsig gg.thms";

new_theory "gg";

load_librarylib = find_library "romulus", theory = "-";
```

2.3.2 Type Definitions

The specification then defines all the types used, particularly the labeled and unlabeled parts of the state, both of which are given as records defined with `romrecord`. In these definitions, the type variables `'Message` and `'Level` denote completely arbitrary types of messages and security levels, and the type variable `'GInState` denotes the type of a completely arbitrary data structure characterizing an executing guard program's internal state. (These data structures might be lists of the files in particular directories that a guard component had not yet finished processing, for example.)

The specification first defines the labeled part of a guard state. The labeled part of the state consists of three bags containing the messages waiting to be processed, those cleared for output, and those rejected for output and sent to be audited. A bag is treated as a function from messages to natural numbers, where the value of the bag on a message is the number of times the message appears in the bag. A labeled variable is treated as a pair of functions, one for accessing and the other for modifying, a particular element of the labeled state. The predicate `validlvar` is true only of the pairs of functions intended to be treated as labeled variables, pairs of functions defined by `romrecord`.

```
val GGBag = ty_antiq(==`:'Message -> num`==);

val (GGLState_Def, GGLState) =
 romrecord
  "GGLState"
  [
   ("Inputbag",    ==`:^GGBag`==),
   ("Outputbag",   ==`:^GGBag`==),
   ("Auditbag",    ==`:^GGBag`==)
  ];

val GGLVar =
 ty_antiq
  (==`:(^GGLState -> ^GGBag) #
       (^GGBag -> ^GGLState -> ^GGLState)`==);

new_definition
 ("validlvar",
  let
   val validlvar = --`validlvar:^GGLVar -> bool`--;
  in
   --`^validlvar (accessbag,updatebag) =
        (((accessbag,updatebag) = (Inputbag,update_Inputbag)) \/
         ((accessbag,updatebag) = (Outputbag,update_Outputbag)) \/
         ((accessbag,updatebag) = (Auditbag,update_Auditbag)))`--
  end);
```

The specification then defines the unlabeled part of a guard state as a single object of the unconstrained type `'GInState`, and then defines SML variables as abbreviations for the labeled, unlabeled, and total guard states.

```
val (GGUState_Def, GGUState) =
 romrecord
  "GGUState"
  [
   ("Internal",    ==`:'GInState`==)
  ];

val GGLState = ty_antiq(==`:('Message)GGLState`==);
val GGUState = ty_antiq(==`:('GInState)GGUState`==);
val GGState = ty_antiq(==`:^GGLState # ^GGUState`==);
```

2.3.3 Primitive Constants and Functions

The specification then declares functions and constants taken as primitive in the generic guard model. These functions and constants involve details that are not specified or that can vary in different guards. Several of the functions take an integer nondeterminism argument. For ease in future reference to polymorphic constants and functions, the specification defines an SML variable for each function or constant whose value is this constant or function with an appropriate type binding.

Several of the definitions use the type `GGObject`, which is the type of a pair consisting of a labeled variable, necessarily a bag, and a message in this bag.

```
val GGObject = ty_antiq(==`:^GGLVar # 'Message`==);
```

Function `dom` gives the dominance relation on security levels:

```
new_constant{Name="dom", Ty= ==`:'Level -> 'Level -> bool`==};

val dom = --`dom:'Level -> 'Level -> bool`--;
```

Function `msg_level` assigns security levels to messages:

```
new_constant{Name="msg_level", Ty= ==`:'Message -> 'Level`==};

val msg_level = --`msg_level:'Message -> 'Level`--;
```

Constant `systemhigh` is the highest security level:

```
new_constant{Name="systemhigh", Ty= ==`:'Level`==};

val systemhigh = --`systemhigh:'Level`--;
```

Constant `systemlow` is the lowest security level:

```
new_constant{Name="systemlow", Ty= ==`:'Level`==};

val systemlow = --`systemlow:'Level`--;
```

Function `g_proj_in_state` maps a level and a guard internal state to a guard internal state intended to be a state the guard would have been in if no messages not dominated by that level had ever been in any of the guard's input, output, or audit bags. This function is not used in the definition of the guard itself, but is used in the definition of a security structure on the guard's unlabeled state, and this security structure is later used in the proof that the guard is shared-state restrictive.

```
new_constant
 {Name="g_proj_in_state",
  Ty= ==`:'Level -> 'GInState -> 'GInState`==};

val g_proj_in_state =
 --`g_proj_in_state:'Level -> 'GInState -> 'GInState`--;
```

Function g_init_in_state is the initial value of the guard internal state:

```
new_constant{Name="g_init_in_state", Ty= ==`:'GInState`==};

val g_init_in_state = --`g_init_in_state:'GInState`--;
```

Predicate g_makes_no_access is true for a nondeterminism argument and a guard state if the guard's next action does not access any message:

```
new_constant
 {Name="g_makes_no_access",
  Ty= ==`:num -> ^GGState -> bool`==};

val g_makes_no_access =
 --`g_makes_no_access:num -> ^GGState -> bool`--;
```

Function g_no_access_internal maps a nondeterminism argument and a guard internal state to the internal state the guard will assume for this nondeterminism argument if it makes a state transition without accessing any message:

```
new_constant
 {Name="g_no_access_internal",
  Ty= ==`:num -> 'GInState -> 'GInState`==};

val g_no_access_internal =
 --`g_no_access_internal:num -> 'GInState -> 'GInState`--;
```

Function g_accessed_object for a nondeterminism argument and a guard state is the message object — basically, the bag and the message in this bag — if the guard's next action accesses such a message:

```
new_constant
 {Name="g_accessed_object",
  Ty= ==`:num -> ^GGState -> ^GGObject`==};

val g_accessed_object =
 --`g_accessed_object:num -> ^GGState -> ^GGObject`--;
```

Function g_msg_access_internal maps a nondeterminism argument, a guard internal state, and a message object to the internal state the guard will assume for this nondeterminism argument after accessing the message in this message object:

```
new_constant
 {Name="g_msg_access_internal",
  Ty= ==`:num -> 'GInState -> ^GGObject -> 'GInState`==};

val g_msg_access_internal =
 --`g_msg_access_internal:num -> 'GInState -> ^GGObject -> 'GInState`--;
```

Predicate `g_makes_no_deletion` is true for a nondeterminism argument, a guard state, and a message object if the guard's next action does not delete or move the message in this message object:

```
new_constant
 {Name="g_makes_no_deletion",
  Ty= ==`:num -> ^GGState -> ^GGObject -> bool`==};

val g_makes_no_deletion =
 --`g_makes_no_deletion:num -> ^GGState -> ^GGObject -> bool`--;
```

Predicate `g_makes_no_new_msg` is true for a nondeterminism argument, a guard state, and a message object if the guard's next action does not change or relocate this message:

```
new_constant
 {Name="g_makes_no_new_msg",
  Ty= ==`:num -> ^GGState -> ^GGObject -> bool`==};

val g_makes_no_new_msg =
 --`g_makes_no_new_msg:num -> ^GGState -> ^GGObject -> bool`--;
```

Function `g_new_msg_object` for a nondeterminism argument, a guard state, and a message object is the changed or moved message object produced by the guard in its next action. If the message is only moved, only the labeled-variable components of the two message objects will differ.

```
new_constant
 {Name="g_new_msg_object",
  Ty= ==`:num -> ^GGState -> ^GGObject -> ^GGObject`==};

val g_new_msg_object =
 --`g_new_msg_object:num -> ^GGState -> ^GGObject -> ^GGObject`--;
```

2.3.4 Security Structures

The specification then defines the labeled and unlabeled security structures for a generic guard. The labeled security structure's value at a security level `lev` simply removes those messages whose levels are not dominated by `lev`. Since the structure of the guard's internal state is not assumed to be known, the unlabeled security structure is taken to be given by the primitive function `g_proj_in_state`.

```
new_definition(
 "projectbag",
 let
  val projectbag = --`projectbag: 'Level -> ^GGBag -> ^GGBag`--;
 in
  --`^projectbag level bag msg =
       ((dom level (msg_level msg)) =>
         (bag msg)
       |
        0)`--
 end);

new_definition(
 "gg_l_ss",
 let
  val gg_l_ss = --`gg_l_ss: 'Level -> ^GGLState -> ^GGLState`--;
 in
  --`^gg_l_ss lev ls =
       (update_Inputbag (projectbag lev (Inputbag ls))
       (update_Outputbag (projectbag lev (Outputbag ls))
       (update_Auditbag (projectbag lev (Auditbag ls))
         ls)))`--
 end);

new_definition(
 "gg_u_ss",
 let
  val gg_u_ss = --`gg_u_ss: 'Level -> ^GGUState -> ^GGUState`--;
 in
  --`^gg_u_ss lev us =
       (update_Internal (g_proj_in_state lev (Internal us)) us)`--
 end);
```

2.3.5 Guard Initial State

The specification then defines the guard's initial state as having all input, output, and audit bags empty, and having the guard's internal state given by the primitive constant g_init_in_state.

```
new_definition("emptybag", --`(emptybag:^GGBag) m = 0`--);

new_definition(
 "gg_initial",
 let
  val emptybag = --`emptybag:^GGBag`--;
 in
  --`
   gg_initial (ls,us) =
     ((ls = (Make_GGLState ^emptybag ^emptybag ^emptybag)) /\
      (us = (Make_GGUState ^g_init_in_state)))
  `--
 end);
```

2.3.6 Internal and External Transitions

This section gives the core of the guard specification, the internal transitions that define the guard's actions and the external transitions that define the environment's actions on the guard.

The internal and external transitions are given in terms of the convenience functions addtobag, remfrombag, and change. As their names indicate, the first two of these functions add messages to bags or remove them from bags. (In the definition of remfrombag, note that HOL's subtraction operation on non-negative integers is already defined so that $0 - 1 = 0$.) Function change replaces one of the labeled-variable bags in a guard labeled state with the result of applying an arbitrary bag-valued function to this bag; it saves having to describe a bag twice, once to access its old value and again to replace its old value with a new one.

```
new_definition
 ("addtobag",
  let
   val addtobag = --`addtobag:'Message -> ^GGBag -> ^GGBag`--;
  in
   --`^addtobag addm bag m =
          ((m = addm) => ((bag m) + 1) | (bag m))'--
    end);

  new_definition
   ("remfrombag",
    let
     val remfrombag = --'remfrombag:'Message -> ^GGBag -> ^GGBag'--;
    in
     --'^remfrombag remm bag m =
          ((m = remm) => ((bag m) - 1) | (bag m))'--
    end);

  new_definition
   ("change",
    let
     val change =
      --'change:
       ^GGLVar -> (^GGBag -> ^GGBag) -> ^GGLState -> ^GGLState'--;
    in
     --'^change (accessbag,updatebag) f ls =
          updatebag (f (accessbag ls)) ls'--
    end);
```

The specification of the guard's inner transitions covers all of the possibilities that the guard makes a transition without accessing a message, makes a transition by accessing a message but not changing any message, accesses a message and deletes it, or accesses a message and changes and/or moves it. The specification expresses the assumptions that a guard will access only one message at a time, will change state only on the basis of the contents of the message accessed, and will delete and/or modify only the message it is accessing. The unspecified details are given by the following primitive functions:

- `g_makes_no_access` tells whether the guard accesses a message;
- `g_no_access_internal` tells which internal state change the guard makes if it does not access a message;
- `g_msg_access_internal` tells which internal state change the guard makes, as a function of the message's contents, if the guard does access a message;

- `g_accessed_object` tells which message, and its location, if the guard accesses a message;
- `g_makes_no_deletion` tells whether the guard deletes the message it accesses;
- `g_makes_no_new_msg` tells whether the guard moves or modifies the message it accesses; and
- `g_new_msg_object` tells what new message the guard creates from the message it accesses if it moves or modifies this message and where it puts the new message.

```
new_definition(
 "gg_inner_trans",
 let
  val gg_inner_trans =
   --`gg_inner_trans:num -> ^GGState -> ^GGState`--;
 in
  --`
   ^gg_inner_trans n (ls, us) =

     let us' =
      (update_Internal
        ((^g_makes_no_access n (ls,us)) =>
          (^g_no_access_internal n (Internal us))
        |
         (^g_msg_access_internal
           n
           (Internal us)
           (^g_accessed_object n (ls,us))))
        us) in

     let ls' =
      ((^g_makes_no_access n (ls,us)) =>
         ls
      |
       (let obj = (^g_accessed_object n (ls,us)) in
        ((^g_makes_no_deletion n (ls,us) obj) =>
           ls
        |
         (let ((accessbag,updatebag),msg) = obj in
          (~g_makes_no_new_msg n (ls,us) obj) =>
           (change (accessbag,updatebag) (remfrombag msg) ls)
          |
           (let obj' = (g_new_msg_object n (ls,us) obj) in
            let ((accessbag',updatebag'),msg') = obj' in
             (change (accessbag',updatebag') (addtobag msg')
             (change (accessbag,updatebag) (remfrombag msg)
               ls)))))) in

      (ls',us')
    '--
  end);
```

The definition of outer transitions simply says that messages can be added to the input bag or removed from the audit or output bags at any time. Outer transitions cannot change the unlabeled state variables.

```
new_definition(
 "gg_outer_trans",
 let
  val gg_outer_trans =
   --'gg_outer_trans:^GGState -> ^GGState -> bool'--;
 in
  --'
   ^gg_outer_trans (ls, us) (ls', us') =

    ((?msg.
       (ls' =
         (change (Inputbag,update_Inputbag) (addtobag msg) ls)) \/
       (ls' =
         (change (Outputbag,update_Outputbag) (remfrombag msg) ls)) \/
       (ls' =
         (change (Auditbag,update_Auditbag) (remfrombag msg) ls))) /\

      (us' = us))
  '--
 end);
```

2.3.7 Assumptions About Primitive Functions

This section lists the assumptions made about the constants and functions taken as primitive. The first four of these assumptions are straightforward, asserting that the dominance relation on security levels is reflexive and transitive and that `systemlow` and `systemhigh` are minimal and maximal security levels, respectively.

```
new_open_axiom(
 "dom_reflexive",
 --`!l. ^dom l l`--);

new_open_axiom(
 "dom_transitive",
 --`!l1 l2 l3. ((^dom l1 l2) /\ (dom l2 l3)) ==> (dom l1 l3)`--);

new_open_axiom(
 "systemlow_low",
 --`!l. ^dom l systemlow`--);

new_open_axiom(
 "systemhigh_high",
 --`!l. ^dom systemhigh l`--);
```

The final assumption is not straightforward and should eventually be replaced with weaker assumptions about the specific primitive functions used in the definition of inner transitions. It says that for given values of the nondeterminism argument and a security level, and an arbitrary reachable state, the security-structure projection to this level of the inner transition made with that nondeterminism argument from that state is the same as the inner transition that would have been made with that nondeterminism argument if the guard's state had been the security-structure projection to that level of the guard's true state.

```
new_open_axiom(
 "appearances_would_be_realities",
 let
  val reachable =
   --`reachable:
       (^GGState -> bool) ->
       (num -> ^GGState -> ^GGState) ->
       (^GGState -> ^GGState -> bool) ->
       ^GGState ->
       bool`--;
  val totalss =
   --`totalss:
        'Level ->
        ('Level -> ^GGLState -> ^GGLState) ->
        ('Level -> ^GGUState -> ^GGUState) ->
        ^GGState ->
        ^GGState`--;
  in
   --`
    !n ls us lev.
     ^reachable gg_initial gg_inner_trans gg_outer_trans (ls,us) ==>
     (^totalss
        lev
        gg_l_ss
        gg_u_ss
        (gg_inner_trans n (ls,us)) =
       gg_inner_trans
        n
        (^totalss lev gg_l_ss gg_u_ss (ls,us)))
   `--
  end);
```

2.4 Proof

This section gives the proof that, under the assumptions made, the generic guard is shared-state restrictive. A complete proof transcript, including all of HOL90's responses, is given in Appendix 2.A.

The proof begins by stating the polymorphic goal in appropriate generality, so that it will hold for all values of the type variables 'Message, 'Level, and 'GInState.

```
g(`ssrestrictive
    (gg_initial:^GGState -> bool)
    (gg_inner_trans:num -> ^GGState -> ^GGState)
    (gg_outer_trans:^GGState -> ^GGState -> bool)
    (gg_l_ss:'Level -> ^GGLState -> ^GGLState)`);
```

The proof then expands out the definition of ssrestrictive and starts showing that gg_u_ss is a security structure on the labeled state having the required properties for showing that the guard is shared-state restrictive.

```
e(REWRITE_TAC [definition "sharedstate" "ssrestrictive"]);
(EXISTS_TAC (--`gg_u_ss:'Level -> ^GGUState -> ^GGUState`--));
```

The first step is to show that the property holds for all pairs of states and all nondeterminism arguments and levels if it holds for arbitrarily selected states, nondeterminism arguments, and levels; the next step is to move the hypotheses into the goal's list of assumptions.

```
e(REPEAT STRIP_TAC);
```

It turns out to be sufficient, thanks to the strong assumptions made, to take the extended inner transition with required properties that must exist from state (ls2,us2) to be a one-step inner transition.

```
val gg_inner_trans =
 --`gg_inner_trans:num -> ^GGState -> ^GGState`--;
e(EXISTS_TAC (--`FST (^gg_inner_trans n (ls2,us2))`--));
e(EXISTS_TAC (--`SND (^gg_inner_trans n (ls2,us2))`--));
e(REWRITE_TAC [theorem "pair" "PAIR"]);
```

The proof then shows that the one-step inner transition is an extended inner transition.

```
e(CONJ_TAC);
e(MATCH_MP_TAC (theorem "sharedstate" "inner_ext_inner"));
e(EXISTS_TAC (--`n:num`--));
e(REWRITE_TAC []);
```

Finally, the proof shows that the `totalss` projections of the two one-step inner transitions from states (ls1,us1) and (ls2,us2) are equal.

```
e(IMP_RES_TAC (axiom "-" "appearances_would_be_realities"));
e(ASM_REWRITE_TAC []);
```

The following lines save the result for future use and write the gg theory to the disk:

```
save_top_thm "gg_ssrestrictive";
export_theory();
```

2.A Appendix: Transcripts of Proofs

This appendix contains transcripts of the HOL90 sessions proving the result described in section 2.4. The user inputs are in the lines beginning with - and are in italic type.

```
- g(`ssrestrictive
      (gg_initial:^GGState -> bool)
      (gg_inner_trans:num -> ^GGState -> ^GGState)
      (gg_outer_trans:^GGState -> ^GGState -> bool)
      (gg_l_ss:'Level -> ^GGLState -> ^GGLState)`);
(--`ssrestrictive gg_initial gg_inner_trans gg_outer_trans gg_l_ss`--)
=============================


val it = () : unit
- e(REWRITE_TAC [definition "sharedstate" "ssrestrictive"]);
OK..
1 subgoal:
(--`?unlabeledss.
      !ls1 us1 ls2 us2 n lev.
        reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1) /\
        reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2) /\
        (totalss lev gg_l_ss unlabeledss (ls1,us1) =
         totalss lev gg_l_ss unlabeledss (ls2,us2)) ==>
        (?ls2' us2'.
          ext_inner_trans gg_inner_trans (ls2,us2) (ls2',us2') /\
          (totalss lev gg_l_ss unlabeledss (gg_inner_trans n (ls1,us1)) =
           totalss lev gg_l_ss unlabeledss (ls2',us2')))`--)
=============================


val it = () : unit
- e(EXISTS_TAC (--`gg_u_ss:'Level -> ^GGUState -> ^GGUState`--));
OK..
1 subgoal:
(--`!ls1 us1 ls2 us2 n lev.
      reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1) /\
      reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2) /\
      (totalss lev gg_l_ss gg_u_ss (ls1,us1) =
       totalss lev gg_l_ss gg_u_ss (ls2,us2)) ==>
      (?ls2' us2'.
        ext_inner_trans gg_inner_trans (ls2,us2) (ls2',us2') /\
        (totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
         totalss lev gg_l_ss gg_u_ss (ls2',us2')))`--)
=============================


val it = () : unit
- e(REPEAT STRIP_TAC);
OK..
1 subgoal:
(--`?ls2' us2'.
      ext_inner_trans gg_inner_trans (ls2,us2) (ls2',us2') /\
      (totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
       totalss lev gg_l_ss gg_u_ss (ls2',us2'))`--)
=============================
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1)`--)
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2)`--)
  (--`totalss lev gg_l_ss gg_u_ss (ls1,us1) =
    totalss lev gg_l_ss gg_u_ss (ls2,us2)`--)


val it = () : unit
- val gg_inner_trans =
 --`gg_inner_trans:num -> ^GGState -> ^GGState`--;

val gg_inner_trans = (--`gg_inner_trans`--) : term

- e(EXISTS_TAC (--`FST (^gg_inner_trans n (ls2,us2))`--));
OK..
1 subgoal:
(--`?us2'.
      ext_inner_trans gg_inner_trans (ls2,us2)
        (FST (gg_inner_trans n (ls2,us2)),us2') /\
      (totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
       totalss lev gg_l_ss gg_u_ss
                    (FST (gg_inner_trans n (ls2,us2)),us2'))`--)
=============================
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1)`--)
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2)`--)
  (--`totalss lev gg_l_ss gg_u_ss (ls1,us1) =
    totalss lev gg_l_ss gg_u_ss (ls2,us2)`--)


val it = () : unit
- e(EXISTS_TAC (--`SND (^gg_inner_trans n (ls2,us2))`--));
OK..
```

```
1 subgoal:
(--`ext_inner_trans gg_inner_trans (ls2,us2)
      (FST (gg_inner_trans n (ls2,us2)),
                            SND (gg_inner_trans n (ls2,us2))) /\
    (totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
     totalss lev gg_l_ss gg_u_ss
       (FST (gg_inner_trans n (ls2,us2)),
                            SND (gg_inner_trans n (ls2,us2))))`--)
============================
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1)`--)
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2)`--)
  (--`totalss lev gg_l_ss gg_u_ss (ls1,us1) =
    totalss lev gg_l_ss gg_u_ss (ls2,us2)`--)


val it = () : unit
- e(REWRITE_TAC [theorem "pair" "PAIR"]);
OK..
1 subgoal:
(--`ext_inner_trans gg_inner_trans (ls2,us2)
                                 (gg_inner_trans n (ls2,us2)) /\
    (totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
     totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls2,us2)))`--)
============================
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1)`--)
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2)`--)
  (--`totalss lev gg_l_ss gg_u_ss (ls1,us1) =
    totalss lev gg_l_ss gg_u_ss (ls2,us2)`--)


val it = () : unit
-  e(CONJ_TAC);
OK..
2 subgoals:
(--`totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
    totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls2,us2))`--)
============================
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1)`--)
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2)`--)
  (--`totalss lev gg_l_ss gg_u_ss (ls1,us1) =
    totalss lev gg_l_ss gg_u_ss (ls2,us2)`--)


(--`ext_inner_trans gg_inner_trans (ls2,us2)
                                        (gg_inner_trans n (ls2,us2))`--)
=============================
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1)`--)
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2)`--)
  (--`totalss lev gg_l_ss gg_u_ss (ls1,us1) =
    totalss lev gg_l_ss gg_u_ss (ls2,us2)`--)


val it = () : unit
- e(MATCH_MP_TAC (theorem "sharedstate" "inner_ext_inner"));
OK..
1 subgoal:
(--`?n'. gg_inner_trans n' (ls2,us2) = gg_inner_trans n (ls2,us2)`--)
=============================
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1)`--)
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2)`--)
  (--`totalss lev gg_l_ss gg_u_ss (ls1,us1) =
    totalss lev gg_l_ss gg_u_ss (ls2,us2)`--)


val it = () : unit
- e(EXISTS_TAC (--`n:num`--));
OK..
1 subgoal:
(--`gg_inner_trans n (ls2,us2) = gg_inner_trans n (ls2,us2)`--)
=============================
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1)`--)
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2)`--)
  (--`totalss lev gg_l_ss gg_u_ss (ls1,us1) =
    totalss lev gg_l_ss gg_u_ss (ls2,us2)`--)


val it = () : unit
-  e(REWRITE_TAC []);
OK..

Goal proved.
|- gg_inner_trans n (ls2,us2) = gg_inner_trans n (ls2,us2)

Goal proved.
|- ?n'. gg_inner_trans n' (ls2,us2) = gg_inner_trans n (ls2,us2)

Goal proved.
|- ext_inner_trans gg_inner_trans (ls2,us2) (gg_inner_trans n (ls2,us2))
```

```
Remaining subgoals:
(--`totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
    totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls2,us2))`--)
==============================
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1)`--)
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2)`--)
  (--`totalss lev gg_l_ss gg_u_ss (ls1,us1) =
    totalss lev gg_l_ss gg_u_ss (ls2,us2)`--)


val it = () : unit
- e(IMP_RES_TAC (axiom "-" "appearances_would_be_realities"));
OK..
1 subgoal:
(--`totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
    totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls2,us2))`--)
==============================
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1)`--)
  (--`reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2)`--)
  (--`totalss lev gg_l_ss gg_u_ss (ls1,us1) =
    totalss lev gg_l_ss gg_u_ss (ls2,us2)`--)
  (--`!n lev.
      totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls2,us2)) =
      gg_inner_trans n (totalss lev gg_l_ss gg_u_ss (ls2,us2))`--)
  (--`!n lev.
      totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
      gg_inner_trans n (totalss lev gg_l_ss gg_u_ss (ls1,us1))`--)


val it = () : unit
- e(ASM_REWRITE_TAC []);
OK..

Goal proved.
...
|- totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
   totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls2,us2))

Goal proved.
...
|- totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
   totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls2,us2))
```

```
Goal proved.
...
|- ext_inner_trans gg_inner_trans (ls2,us2)
                                  (gg_inner_trans n (ls2,us2)) /\
   (totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
    totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls2,us2)))

Goal proved.
...
|- ext_inner_trans gg_inner_trans (ls2,us2)
     (FST (gg_inner_trans n (ls2,us2)),
                            SND (gg_inner_trans n (ls2,us2))) /\
   (totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
    totalss lev gg_l_ss gg_u_ss
      (FST (gg_inner_trans n (ls2,us2)),
         SND (gg_inner_trans n (ls2,us2))))

Goal proved.
...
|- ?us2'.
     ext_inner_trans gg_inner_trans (ls2,us2)
       (FST (gg_inner_trans n (ls2,us2)),us2') /\
     (totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
      totalss lev gg_l_ss gg_u_ss
         (FST (gg_inner_trans n (ls2,us2)),us2'))

Goal proved.
...
|- ?ls2' us2'.
     ext_inner_trans gg_inner_trans (ls2,us2) (ls2',us2') /\
     (totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
      totalss lev gg_l_ss gg_u_ss (ls2',us2'))

Goal proved.
|- !ls1 us1 ls2 us2 n lev.
     reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1) /\
     reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2) /\
     (totalss lev gg_l_ss gg_u_ss (ls1,us1) =
      totalss lev gg_l_ss gg_u_ss (ls2,us2)) ==>
     (?ls2' us2'.
       ext_inner_trans gg_inner_trans (ls2,us2) (ls2',us2') /\
       (totalss lev gg_l_ss gg_u_ss (gg_inner_trans n (ls1,us1)) =
        totalss lev gg_l_ss gg_u_ss (ls2',us2')))
```

```
Goal proved.
|- ?unlabeledss.
     !ls1 us1 ls2 us2 n lev.
       reachable gg_initial gg_inner_trans gg_outer_trans (ls1,us1) /\
       reachable gg_initial gg_inner_trans gg_outer_trans (ls2,us2) /\
       (totalss lev gg_l_ss unlabeledss (ls1,us1) =
        totalss lev gg_l_ss unlabeledss (ls2,us2)) ==>
       (?ls2' us2'.
         ext_inner_trans gg_inner_trans (ls2,us2) (ls2',us2') /\
         (totalss lev gg_l_ss unlabeledss (gg_inner_trans n (ls1,us1)) =
          totalss lev gg_l_ss unlabeledss (ls2',us2')))

Goal proved.
|- ssrestrictive gg_initial gg_inner_trans gg_outer_trans gg_l_ss

Top goal proved.
val it = () : unit
- save_top_thm "gg_ssrestrictive";
val it = |- ssrestrictive gg_initial gg_inner_trans gg_outer_trans gg_l_ss
  : thm
- export_theory();

Theory "gg" exported.
val it = () : unit
- exit();
```

Chapter 3

The Secure Minix Model

3.1 Introduction

This chapter gives a specification for a secure (in the sense of restrictive [14, 15]) operating system based on the Minix operating system created by Tanenbaum and described in the book "Operating System: Design and Implementation"[22]. The main additions to Minix in the system modeled here are data structures for maintaining security-level information and procedures for imposing mandatory access controls. The system model given here is believed to be restrictive, and the chapter proves some of the conditions sufficient to guarantee that it is so. The chapter proves that the model is a server process and proves that its outputs in response to any input are at security levels greater than or equal to the security level of the input. Section 3.3.4 lists further conditions whose proofs would suffice to show that the model is restrictive.[1]

For further discussion of restrictiveness, see Volume II of the Romulus Documentation Set.

The security model used in the operating system modeled here is for the most part standard. The active "subjects" in the model are user processes, and the "objects" are data files or user processes. (The `kill` command provides an example of treating a user process as an object.) Subjects and

[1]This model uses a version of Romulus that allows the levels of input and output events to depend on the process's state, something that is not supported in the current Romulus. Further work is needed to support this feature in a robust manner.

objects are assigned security levels, and a subject is not permitted to obtain information about an object unless the subject's level dominates the object's level or to affect an object unless the object's level dominates the subject's level. Since the model assumes that security levels are given by information stored in the operating system itself, the model might be generalized to describe an operating system that allows security levels to be changed dynamically. Also, since the model's security is analyzed in terms of restrictiveness in the special case of buffered server processes, the model also implicitly addresses possible "covert channels" involving shared resources such as system data tables.

The operating system model given here is primarily of interest because when and if the proof that it is restrictive is completed, it will provide insight into the design of restrictive operating systems. If proven restrictive, it will show that even something as complicated as this model of an operating system can be made restrictive. The model and the proofs of its properties given here also serve as nice examples of how the Romulus techniques and utilities for specifying processes and proving facts about them can be used to manage complexity and simplify proofs. Although the two proofs given here are short, they together include proving over 60 distinct subgoals.

The restrictive process in this specification roughly corresponds to the kernel, file system, and memory management of Tanenbaum's system. Tanenbaum's user processes, plus the memory allocated to these processes, correspond to the user processes in this specification. In the remainder of this chapter, "kernel" will refer to the restrictive process and "user processes" or just "processes" will refer to the user processes. The model specified here primarily differs from Minix in that it maintains and uses information on security levels to make its actions secure.

The model also differs from Minix in being a partial description. It includes the process management, signal, file management, protection, and time management calls `alarm`, `brk`, `chmod`, `chown`, `close`, `create`, `exec`, `exit`, `fork`, `fstat`, `getuid`, `ioctl`, `kill`, `lseek`, `mknod`, `open`, `pause`, `read`, `setgid`, `setuid`, `signal`, `stat`, `stime`, `time`, `times`, `umask`, `utime`, `wait`, and `write`, but it takes several functions involved in performing these operations as primitive and characterizes them only by their names, the types of their arguments and return values, and assumed properties about them. The model does not include the following process and directory management calls: `access`, `chdir`, `chroot`, `dup`, `pipe`, `link`, `unlink`, `mount`, `unmount`, and sync.

3.2 Formal Model

This section contains and describes the SML code formally specifying the kernel model for HOL90. The kernel is given as a server process that waits for input from a user process, processes this input by possibly producing output and changing its state parameters, and then returns to wait for the next input from a user process.

3.2.1 Front Matter

The specification creates a HOL90 theory named `minix`, defining types, constants, and functions and assigning properties to the constants and functions with definitions or axioms. The specification begins with lines removing any older versions of the theory `minix`, setting the theory path to include the library of Romulus process and security theories, and creating the `minix` theory as a child theory of the Romulus security theory. These lines also load the code for Romulus convenience functions for creating concrete-recursive and record types.

```
System.system "rm -f minix.holsig minix.thms";

new_theory "minix";

load_librarylib = get_library_from_disk "romulus",
             theory = "-";
```

3.2.2 Type Declarations

The specification next declares all the types used in the specification of the kernel process, culminating in the definition of the type of PSL object appropriate as a model of the kernel process. The types defined include the kernel state parameters, the input events, and the output events. For maximum generality, the specification uses type variables instead of explicit types wherever possible, so the types it declares are actually polymorphic type constructors. The type variables `'UserProcess`, `'File`, `'FileContents`, `'Arg`, and 'Level denote the arbitrary types of user process identifiers, file identifiers, file contents, arguments to calls executing files, and security levels.

SML Abbreviations

Since type definitions are not yet implemented in HOL90, the specification defines SML variables to use as convenient abbreviations for complicated types; this type information can then be conveniently obtained by antiquotation. The SML variables `Segments`, `PermissionsMask`, and `CreateMask`, give the types of the following: the text, data, and stack segments allocated to user processes; the read, write, and execute permissions requested by processes for files; and the bits characterizing a newly created file.

```
val Segments =
 ty_antiq(==`:(num # num) # (num # num) # (num # num)`==);

val PermissionsMask =
 ty_antiq( ==`:bool # bool # bool`==);  (* read, write, execute *)

val CreateMask =
 ty_antiq(
  ==`:
   bool #       (* read access allowed *)
   bool #       (* write access allowed *)
   bool #       (* execute access allowed *)
   bool #       (* directory *)
   bool #       (* special file *)
   bool #       (* setuid allowed *)
   bool         (* setgid allowed *)
  `==);
```

State Parameters

With these preliminary definitions out of the way, the specification gives its first major type declarations, those of process records, inode records, and access records. It then defines the kernel's state parameter as a record containing five fields:

- A function assigning a process record, giving process information, to every user process.

- A function assigning an inode record, giving file information, to every file.
- A function assigning an access record, giving information on a process' current access to a file, to every user process and file.
- A function assigning (possibly empty) contents to every file.
- The system time.

In recognition of how these things are typically implemented, the four functions in the kernel state parameter are called *tables.* The precise definitions of process, inode, and access records follow.

In these definitions, the Romulus utility **romrecord** returns a pair consisting of 1) a theorem giving an abstract characterization of the concrete-recursive type being defined and 2) an SML variable giving an abbreviation for this type with all polymorphism replaced by instantiation with appropriate type variables. This utility also defines functions for creating each record and for accessing and updating each entry in each record.

```
val (ProcessRecord_Def, ProcessRecord) =
 romrecord
  "ProcessRecord"
  [
   ("p_segs",       ==`:^Segments`==), (* text, data, stack *)
   ("p_exitstatus", ==`:num`==),  (* exit status *)
   ("p_sigstatus",  ==`:num`==),  (* signal for killed *)
   ("p_pid",        ==`:num`==),  (* process ID *)
   ("p_parentid",   ==`:num`==),  (* process ID of parent *)
   ("p_procgrp",    ==`:num`==),  (* process group *)
   ("p_realuid",    ==`:num`==),  (* real process user ID *)
   ("p_effuid",     ==`:num`==),  (* effective process user ID *)
   ("p_realgid",    ==`:num`==),  (* real process group ID *)
   ("p_effgid",     ==`:num`==),  (* effective process group ID *)
   ("p_func",       ==`:num`==),  (* pointer to single user? *)
   ("p_ignore",     ==`:bool`==), (* ignore signal *)
   ("p_catch",      ==`:bool`==), (* catch signal *)
   ("p_scheduled",  ==`:bool`==), (* process scheduled *)
   ("p_in_use",     ==`:bool`==), (* process slot in use *)
   ("p_waiting",    ==`:bool`==), (* process waiting *)
   ("p_hanging",    ==`:bool`==), (* process hanging *)
   ("p_paused",     ==`:bool`==), (* process paused *)
  ("p_alarm_on",   ==`:bool`==), (* process alarm on *)
  ("p_separate",   ==`:bool`==), (* separate I and D space *)
  ("p_mask",       ==`:^PermissionsMask`==), (* default mask *)
  ("p_level",      ==`:'Level`==) (* security level *)
 ];

val (InodeRecord_Def, InodeRecord) =
 romrecord
  "InodeRecord"
  [
   ("i_uid",      ==`:num`==),   (* owner's process ID *)
   ("i_filesize", ==`:num`==),   (* current file size in bytes *)
   ("i_modtime",  ==`:num`==),   (* time of last modification *)
   ("i_gid",      ==`:num`==),   (* group number of file *)
   ("i_nlinks",   ==`:num`==),   (* number of links to file *)
   ("i_zone",     ==`:num`==),   (* zone #s: direct, ind, dbl ind? *)
   ("i_dev",      ==`:num`==),   (* device inode is on *)
   ("i_inodenum", ==`:num`==),   (* inode number on minor device *)
   ("i_count",    ==`:num`==),   (* number of times inode used *)
   ("i_ioctl",    ==`:num`==),   (* I/O mode of special file *)
   ("i_in_use",   ==`:bool`==),  (* file slot in use *)
   ("i_setuid",   ==`:bool`==),  (* setuid allowed? *)
   ("i_setgid",   ==`:bool`==),  (* setgid allowed? *)
   ("i_perm_x",   ==`:bool`==),  (* rwx bit for exec? *)
   ("i_perm_w",   ==`:bool`==),  (* rwx bit for write? *)
   ("i_perm_r",   ==`:bool`==),  (* rwx bit for read? *)
   ("i_mode_f",   ==`:bool`==),  (* special file *)
   ("i_mode_d",   ==`:bool`==),  (* directory *)
   ("i_dirt",     ==`:bool`==),  (* CLEAN (false) or DIRTY (true) *)
   ("i_pipe",     ==`:bool`==),  (* pipe *)
   ("i_mount",    ==`:bool`==),  (* mounted *)
   ("i_seek",     ==`:bool`==),  (* lseek in progress *)
   ("i_level",    ==`:'Level`==) (* security level *)
  ];

val (AccessRecord_Def, AccessRecord) =
 romrecord
  "AccessRecord"
  [
   ("a_lcount",   ==`:num`==),  (* length count *)
   ("a_pos",      ==`:num`==),  (* current index position *)
   ("a_in_use",   ==`:bool`==), (* file in use *)
   ("a_read",     ==`:bool`==), (* read access *)
   ("a_write",    ==`:bool`==)  (* write access *)
```

```
romrecord
 "MinixState"
 [
  ("s_ptb",
    ==`:'UserProcess->^ProcessRecord`==),      (* Process Table *)
  ("s_itb",
    ==`:'File -> ^InodeRecord`==),             (* Inode Table *)
  ("s_atb",
    ==`:'UserProcess->'File->^AccessRecord`==), (* Access Table *)
  ("s_ftb",
     ==`:'File -> 'FileContents`==),           (* File Table *)
  ("s_systemtime",
    ==`:num`==)                                (* system time *)
 ];
```

Input Events

The specification next defines the input events to the kernel. These input events are themselves defined in terms of the concrete recursive type `MinixRequest`, which names the various requests to the kernel and gives the types of the additional pieces of information provided with each request — for example, the name of the file to close. An input event is defined as the type constructor `Inport`, corresponding to a single abstract input port conveying messages to the kernel, applied to a user process and a `MinixRequest`; the input event `Inport user request` is interpreted as the receipt of `request` from `user`.

In these definitions, the Romulus utility `romcontype` returns a pair consisting of 1) a theorem giving an abstract characterization of the concrete-recursive type being defined and 2) an SML variable giving an abbreviation for this type with all polymorphism replaced by instantiation with appropriate type variables.

```
val (MinixRequest_Def, MinixRequest) =
 romcontype
  "MinixRequest"
  [
   ("alarm",  []),
   ("brk",    []),
   ("chmod",  [==`:'File`==, ==`:^PermissionsMask`==,
               ==`:bool`==, ==`:bool`==]),
   ("chown",  [==`:'File`==, ==`:num`==, ==`:num`==]),
    ("chmod",  [==`:'File`==, ==`:^PermissionsMask`==,
                ==`:bool`==, ==`:bool`==]),
    ("chown",  [==`:'File`==, ==`:num`==, ==`:num`==]),
    ("close",  [==`:'File`==]),
    ("create", [==`:'File`==, ==`:^CreateMask`==]),
    ("exec",   [==`:'File`==, ==`:('Arg)list`==]),
    ("exit",   []),
    ("fork",   []),
    ("fstat",  [==`:'File`==]),
    ("getgid", []),
    ("getuid", []),
    ("ioctl",  [==`:'File`==, ==`:num`==]),
    ("kill",   [==`:'UserProcess`==]),
    ("lseek",  [==`:'File`==, ==`:num`==]),
    ("mknod",  [==`:'File`==, ==`:^CreateMask`==]),
    ("open",   [==`:'File`==, ==`:^PermissionsMask`==]),
    ("pause",  []),
    ("read",   [==`:'File`==]),
    ("setgid", [==`:num`==]),
    ("setuid", [==`:num`==]),
    ("signal", [==`:num`==]),
    ("stat",   [==`:'File`==]),
    ("stime",  [==`:num`==]),
    ("time",   []),
    ("times",  []),
    ("umask",  [==`:^PermissionsMask`==]),
    ("utime",  [==`:'File`==, ==`:num`==]),
    ("wait",   []),
    ("write", [==`:'File`==, ==`:'FileContents`==])
   ];

 val (MinixInEv_Def, MinixInEv) =
  romcontype
   "MinixInEv"
   [
    ("Inport", [==`:'UserProcess`==, ==`:^MinixRequest`==])
   ];
```

Output Events

The specification next defines the output events produced by the kernel. These output events are themselves defined in terms of the concrete recursive type MinixReply, which gives the types of the information appropriate to those requests receiving information in addition to a status value. An output event is defined as the type constructor `Outport`, corresponding to a single abstract output port conveying information other than a status reply, applied to a user process and a `MinixRequest`, or as the type constructor `Statport`, corresponding to a single abstract port conveying status information, applied to a user process and a non-negative integer. The output event `Outport user reply` is interpreted as sending non-status information `reply` to `user`. The output event `Statport user n` is interpreted as sending status value `n` to `user`.

In these definitions, the Romulus utility **romcontype** returns a pair consisting of 1) a theorem giving an abstract characterization of the concrete-recursive type being defined and 2) an SML variable giving an abbreviation for this type with all polymorphism replaced by instantiation with appropriate type variables.

```
val (MinixReply_Def, MinixReply) =
 romcontype
  "MinixReply"
  [
   ("execreply",   [==`:'UserProcess`==]),
   ("forkreply",   [==`:'UserProcess`==]),
   ("fstatreply",  [==`:num`==]),
   ("getgidreply", [==`:num`==]),
   ("getuidreply", [==`:num`==]),
   ("killreply",   [==`:num`==]),
   ("readreply",   [==`:'FileContents`==]),
   ("statreply",   [==`:num`==]),
   ("timereply",   [==`:num`==]),
   ("timesreply",  [==`:num # num`==]),
   ("umaskreply",  [==`:^PermissionsMask`==])
  ];

val (MinixOutEv_Def, MinixOutEv) =
 romcontype
  "MinixOutEv"
  [
   ("Outport",  [==`:'UserProcess`==, ==`:^MinixReply`==]),
   ("Statport", [==`:'UserProcess`==, ==`:num`==])
  ];
```

Invocations

The specification next defines the invocations to be used later in defining the kernel process. Invocations are essentially names for calls to PSL-valued functions; they are mapped to the PSL processes resulting from these calls. Invocations provide a means for overcoming the limitation in HOL's `define_type` function that concrete recursive types cannot be defined in terms of functions whose values are of the type being defined.

The invocations name calls to the kernel process itself and to the function computing the kernel process' response to input events. All these calls include the kernel's state parameter as an argument.

```
val (MinixInvoc_Def, MinixInvoc) =
 romcontype
  "MinixInvoc"
  [
   ("Minix",         [==`:^MinixState`==]),
   ("MinixResponse", [==`:^MinixState`==, ==`:^MinixInEv`==])
  ];
```

PSL Processes

The invocations complete the definition of the type of PSL processes appropriate as models of the Minix kernel. The specification defines the SML variable `MinixProc` as an abbreviation for this type.

```
val MinixProc =
 ty_antiq(==`:(^MinixOutEv,^MinixInEv,^MinixInvoc) process`==);
```

3.2.3 Functions Taken as Primitive

The specification then declares constants for the functions taken as primitive in the model. These functions involve details of the implementation that the model excludes.

Function `allocate_process_segments` returns the text, data, and stack segments of a new process.

```
new_constant {
 Name = "allocate_process_segments",
 Ty = ==`:'UserProcess ->
         ^MinixState ->
         (num # num) # (num # num) # (num # num)`==};
```

Function args_okay tells whether the arguments given in a call to an executable file are of the types appropriate to that file.

```
new_constant {
 Name = "args_okay", Ty = ==`:('Arg)list -> 'File -> bool`==};
```

Constant function catch_sig names the value sent to a process in order to cause it to terminate.

```
new_constant{Name = "catch_sig", Ty = ==`:num`==};
```

Function date is assumed to be the current time.

```
new_constant{Name = "date", Ty = ==`:num`==};
```

Function dom is the dominance relation on security levels.

```
new_constant{Name = "dom", Ty = ==`:'Level -> 'Level -> bool`==};
```

Function dummy_process returns dummy processes with given security levels.

```
new_constant{Name="dummy_process",Ty = ==`:'Level->'UserProcess`==};
```

Constant function empty_file_contents is the content of an empty file.

```
new_constant{Name="empty_file_contents",Ty = ==`:'FileContents`==};
```

Function execute_okay tells whether a particular user process has permission to execute a particular file for a particular kernel state parameter.

```
new_constant{
 Name = "execute_okay",
 Ty = ==`:'UserProcess -> 'File -> ^MinixState -> bool`==};
```

Function file_status computes an integer summarizing selected information from a file's inode entry.

```
new_constant{Name="file_status",Ty = ==`:^InodeRecord -> num`==};
```

Function id_process returns the process having a particular ID.

```
new_constant{Name = "id_process",Ty = ==`:num -> 'UserProcess`==};
```

Function image maps an executable file and a list of arguments to the corresponding process when that file is executed on those arguments.

```
new_constant{
 Name = "image", Ty = ==`:'File -> ('Arg)list -> 'UserProcess`==};
```

Function new_pos returns the actual file position number corresponding to an index from the start of the file.

```
new_constant{
 Name = "new_pos", Ty = ==`:'File -> ^MinixState -> num -> num`==};
```

Function new_process returns a new process for a given kernel process state parameter.

```
new_constant{
 Name = "new_process", Ty = ==`:^MinixState -> 'UserProcess`==};
```

Function new_zone returns the zone for a newly created file.

```
new_constant{Name="new_zone",Ty = ==`:'File->^MinixState->num`==};
```

Function process_effgid creates an effective group ID for a new process.

```
new_constant{Name="process_effgid",Ty = ==`:'UserProcess -> num`==};
```

Function process_effuid creates an effective user ID for a new process.

```
new_constant{Name="process_effuid",Ty = ==`:'UserProcess -> num`==};
```

Function process_id creates an ID for a new process.

```
new_constant{Name = "process_id",Ty = ==`:'UserProcess -> num`==};
```

Function process_time returns the time a process has been executing.

```
new_constant{Name = "process_time", Ty = ==`:'UserProcess -> num`==};
```

Function room_in_access_table tells whether there is space for one more entry in the access table.

```
new_constant{
 Name = "room_in_access_table", Ty = ==`:^MinixState -> bool`==};
```

Function room_in_inode_table tells whether there is space for one more entry in the inode table.

```
new_constant{
 Name = "room_in_inode_table", Ty = ==`:^MinixState -> bool`==};
```

Function room_in_process_table tells whether there is space for one more entry in the process table.

```
new_constant {
 Name = "room_in_process_table", Ty = ==`:^MinixState -> bool`==};
```

Constant functions sig_catch and sig_ignore name arguments to the signal command.

```
new_constant{Name = "sig_catch",Ty = ==`:num`==};
new_constant{Name = "sig_ignore", Ty = ==`:num`==};
```

Function space_for_new_process tells whether there is enough space in memory to add another user process.

```
new_constant{
 Name = "space_for_new_process", Ty = ==`:^MinixState -> bool`==};
```

Function space_to_load tells whether there is sufficient space in memory to load an executable file.

```
new_constant {
 Name = "space_to_load", Ty = ==`:'File -> ^MinixState -> bool`==};
```

Function stack_size_okay tells whether the memory segments assigned to a process overlap.

```
new_constant {
 Name = "stack_size_okay", Ty = ==`:^Segments -> bool`==};
```

Function stat_rep computes the appropriate successful execution status reply for a process and a MinixRequest.

```
new_constant {
 Name="stat_rep",Ty = ==`:'UserProcess -> ^MinixRequest -> num`==};
```

Function super tells whether a process is a super-user process.

```
new_constant{Name = "super", Ty = ==`:'UserProcess -> bool`==};
```

Constant function `systemlow` is the lowest security level.

```
new_constant{Name = "systemlow", Ty = ==`:'Level`==};
```

Constant function `systemhigh` is the highest security level.

```
new_constant{Name = "systemhigh", Ty = ==`:'Level`==};
```

Function `used_times` returns the user and system times used by a process.

```
new_constant {
 Name = "used_times",
 Ty = ==`:'UserProcess -> ^MinixState -> (num # num)`==};
```

3.2.4 Assumptions for Primitive Functions

The specification next states assumed properties of the functions taken as primitive in the model. These assumptions are stated in terms of essentially arbitrary constants, two of which are polymorphic.

`Make_ProcessRecord`, `Make_InodeRecord`, and `Make_AccessRecord` were defined by the earlier calls to the utility `romrecord`.

```
new_definition
 ("empty_p",
  --`
   empty_p:^ProcessRecord =
    Make_ProcessRecord
     ((0,0),(0,0),(0,0)) 0 0 0 0 0 0 0 0 0 0
     F F F F F F F F F (F,F,F) systemhigh
  `--);

new_definition
 ("empty_i",
  --`
```

```
    empty_i:^InodeRecord =
     Make_InodeRecord
      0 0 0 0 0 0 0 0 0 0 F F F F F F F F F F F F systemhigh
   `--);

new_definition
 ("empty_a",
  --`
   empty_a:^AccessRecord =
    Make_AccessRecord 0 0 F F F
  `--);
```

After making these preliminary definitions, the specification asserts assumed properties of the functions taken as primitive in the model:

```
new_open_axiom(
 "systemlow_low",
 (--`!l:'Level. dom l systemlow`--));

new_open_axiom(
 "systemhigh_high",
 (--`!l:'Level. dom systemhigh l`--));

new_open_axiom(
 "dom_reflexive",
 (--`!l:'Level. dom l l`--));

new_open_axiom(
 "dom_transitive",
 (--`!(l1:'Level) l2 l3.
     ((dom l1 l2) /\ (dom l2 l3)) ==> (dom l1 l3)`--));

new_open_axiom(
 "dummy_process_injective",
 let
  val dummy_process = --`dummy_process:'Level -> 'UserProcess`--;
 in
  (--`!l1 l2.
      ((^dummy_process l1) = (dummy_process l2)) = (l1 = l2)`--)
 end);

new_open_axiom(
 "new_process_new",
 let
 val new_process = --`new_process:^MinixState -> 'UserProcess`--;
in
 (--`!state:^MinixState.
     ((s_ptb state) (^new_process state)) = empty_p`--)
end);
```

3.2.5 Initial State Parameter

The specification next defines the initial value of the kernel's state parameter. Intuitively, the only user processes with entries in the process table are dummy processes for each of the security levels, no file has an entry in the inode table, no user process has any access to any file, no file has nonempty contents, and the system time is 0.

The `update_...` functions were defined by earlier calls to `romrecord`. The call `update_p_pid id procrec`, as a typical instance, returns the process record whose value in the `p_pid` entry is `id` and whose values in all other entries are the same as the values of the corresponding entries in `procrec`.

```
new_definition(
 "MinixInitParam",
 let
  val MinixInitParam = --`MinixInitParam:^MinixState`--;
  val empty_p = --`empty_p:^ProcessRecord`--;
  val empty_i = --`empty_i:^InodeRecord`--;
 in
  --`
   ^MinixInitParam =

     let initptb (p:'UserProcess) =
      ((?level:'Level. p = (dummy_process level)) =>
        (update_p_pid (process_id p)
        (update_p_level (@level:'Level. p = (dummy_process level))
          ^empty_p))
       |
        empty_p) in

     let inititb (f:'File) =
      ^empty_i in

     let initatb (p:'UserProcess) (f:'File) =
      empty_a in
```

```
    let initftb (f:'File) =
     (empty_file_contents:'FileContents) in

    let initsystime = 0 in

    Make_MinixState initptb inititb initatb initftb initsystime
 '--
end);
```

3.2.6 Invariant

The specification next defines the invariant, a predicate to be shown by induction to be true of every state parameter attained by the kernel process. The invariant serves as an optional, useful induction hypothesis about the kernel process' state parameter. Intuitively, it says that the level of the dummy process for each level is always that level, that the level of every empty or undefined file is `systemlow`, that a user process has read access to a file only if the level of that process dominates the level of that file, and that a user process has write access to a file only if the level of that file dominates the level of that process.

```
new_definition(
 "MinixInvariant",
 let
  val MinixInvariant = --'MinixInvariant:^MinixState -> bool'--;
  val dummy_process = --'dummy_process:'Level -> 'UserProcess'--;
 in
  --'
   ^MinixInvariant mstate =

    (!level.
      (p_level ((s_ptb mstate) (^dummy_process level))) = level) /\

   (!f.
     (((s_ftb mstate) f) = empty_file_contents) ==>
     ((i_level ((s_itb mstate) f)) =
      systemlow)) /\

   !p f.
    ((a_read ((s_atb mstate) p f))
     ==> (dom
           (p_level ((s_ptb mstate) p))
```

```
          (i_level ((s_itb mstate) f)))) /\
   ((a_write ((s_atb mstate) p f))
    ==> (dom
          (i_level ((s_itb mstate) f))
          (p_level ((s_ptb mstate) p))))
 '--
end);
```

3.2.7 Projection Function

The specification then defines the projection function, a function of a security level and a kernel state parameter. The projection's value at a level and a state parameter is the state parameter containing all, but only, the information necessary to produce that part of the kernel's behavior produced with the original state parameter that is visible at that level. Intuitively, every process whose level is not dominated by the projection level is removed from the process table, every file whose level is not dominated by the projection level is removed from the inode table, every entry involving one of these processes or one of these files is removed from the access table, and every file whose level is not dominated by the projection level is removed from the file table. The system time is unchanged.

The specification uses several of the record-entry access functions defined by earlier calls to `romrecord`.

```
new_definition(
 "MinixProjection",
 let
  val MinixProjection =
   --'MinixProjection:
       'Level -> ^MinixState -> ^MinixState'--;
 in
  --'
   ^MinixProjection
     level
     mstate =

   let startptb = s_ptb mstate in
   let startitb = s_itb mstate in
   let startatb = s_atb mstate in
   let startftb = s_ftb mstate in
```

```
    let projectedptb p =
     (dom level (p_level (startptb p))) =>
       (startptb p)
     |
       empty_p in

    let projecteditb f =
     (dom level (i_level (startitb f))) =>
       (startitb f)
     |
       empty_i in

    let projectedatb p f =
     ((dom level (p_level (startptb p))) /\
      (dom level (i_level (startitb f)))) =>
       (startatb p f)
     |
       empty_a in

    let projectedftb f =
     (dom level (i_level (startitb f))) =>
       (startftb f)
     |
       empty_file_contents in

      (update_s_ptb projectedptb
      (update_s_itb projecteditb
      (update_s_atb projectedatb
      (update_s_ftb projectedftb
        mstate))))
  `--
  end);
```

3.2.8 Security-Level Assignments

The specification then defines the functions assigning security levels to input and output events. Intuitively, the level of any input or output event is the level of the user process sending or receiving the event, where the level of each user process is part of the information in the process table for the current kernel state parameter. Separate functions must be defined for inputs and outputs, since input events and output events are of distinct types.

The specification uses `new_recursive_definition`, which defines functions on concrete recursive types. One of its arguments, rec_axiom, is the theorem giving an abstract characterization of the concrete recursive type over which the function is being defined.

```
 new_recursive_definition {
  name = "MinixInLevel",
  fixity = Prefix,
  rec_axiom = MinixInEv_Def,
  def =
   let
    val MinixInLevel =
     --`MinixInLevel:^MinixState -> ^MinixInEv -> 'Level`--;
   in
    --`
    (^MinixInLevel mstate (Inport rp request) =
       (p_level
         ((s_ptb mstate) rp)))
    `--
   end};

new_recursive_definition {
 name = "MinixOutLevel",
 fixity = Prefix,
 rec_axiom = MinixOutEv_Def,
 def =
  let
   val MinixOutLevel =
    --`MinixOutLevel:^MinixState -> ^MinixOutEv -> 'Level`--;
  in
   --`
    (^MinixOutLevel mstate (Outport receiver response) =
      (p_level
        ((s_ptb mstate) receiver))) /\

    (^MinixOutLevel mstate (Statport receiver n) =
      (p_level
        ((s_ptb mstate) receiver)))
   `--
  end};
```

3.2.9 Invocation Interpretations

The specification then gives the core of the kernel model, the interpretations of the invocations of the kernel server process and the function giving the kernel's response to input events. The function `MinixInvocVal`, defined last, asserts that functions `minix` and `minixResponse` are called from PSL processes via the invocations `Minix` and `MinixResponse`, respectively. The specification leads up to the definition of `MinixInvocVal`, first defining the `minixResponse` subroutine `reqresponse`, then defining `minixResponse` and `minix`, and finally defining `MinixInvocVal`. The function `reqresponse`, which gives the response for a kernel state parameter and requesting user process to an arbitrary Minix request, contains most of the details of the model of the kernel process.

Function `reqresponse`

The function `reqresponse` gives the kernel's response to each possible request from a user process. It is defined using `new_recursive_definition`, over the concrete-recursive type `MinixRequest`.

Initial Lines The initial lines of the `reqresponse` definition name the function, provide the appropriate theorem giving an abstract characterization of the function's domain, and define local SML variables that are used to provide type information for polymorphic constants or which serve as convenient abbreviations.

```
new_recursive_definition {
 name = "reqresponse",
 fixity = Prefix,
 rec_axiom = MinixRequest_Def,
 def =
  let

   (* typed polymorphic constants *)

   val reqresponse =
    --`reqresponse:
        ^MinixState->'UserProcess->^MinixRequest->^MinixProc`--;
   val id_process =
    --`id_process:num -> 'UserProcess`--;
    val image = --`image:'File -> ('Arg)list -> 'UserProcess`--;
    val new_zone =
     --`new_zone:'File -> ^MinixState -> num`--;
    val stat_rep =
     --`stat_rep:'UserProcess -> ^MinixRequest -> num`--;

    (* useful abbreviations *)

    val oldptb = --`s_ptb (mstate:^MinixState)`--;
    val olditb = --`s_itb (mstate:^MinixState)`--;
    val oldatb = --`s_atb (mstate:^MinixState)`--;
    val oldftb = --`s_ftb (mstate:^MinixState)`--;
    val rf_dominates_rp =
     --`dom (i_level (^olditb rf)) (p_level (^oldptb rp))`--;
    val rp_dominates_rf =
     --`dom (p_level (^oldptb rp)) (i_level (^olditb rf))`--;
    val rp_using_rf = --`a_in_use (^oldatb rp rf)`--;
   in
    --`
```

The remaining paragraphs in this section give the rest of the definition of reqresponse, with one paragraph for each form of MinixRequest.

Specification for alarm For an alarm request, the kernel produces an appropriate status reply and sets the alarm_on bit in the requesting process' process-table record to "true".

```
  (^reqresponse mstate rp (alarm) =
    (Send (Statport rp (^stat_rep rp alarm));;
    (Call
      (Minix
       (let newptb p =
         ((p = rp) =>
            (update_p_alarm_on T (^oldptb rp))
         | (^oldptb p)) in
        (update_s_ptb newptb mstate))))) /\
```

Specification for brk For a brk request, which changes a user process' allocated data segment, the kernel first produces an appropriate status reply. If the data and stack segments of the memory of the requesting process overlap, it ignores the request. Otherwise, it allocates new text, data, and stack segments for the requesting process, making corresponding changes in the process table.

```
(^reqresponse mstate rp (brk) =
  (Send (Statport rp (^stat_rep rp brk)));;
  (If (~(stack_size_okay (p_segs (^oldptb rp))))
   (Call (Minix mstate))  (* then cannot carry out request *)
   (Call                  (* else can carry out request *)
     (Minix
       (let newptb p =
         ((p = rp) =>
            (update_p_segs (allocate_process_segments rp mstate)
              (^oldptb rp))
          |
           (^oldptb p)) in
        (update_s_ptb newptb mstate)))))) /\
```

Specification for chmod For a chmod request, which changes a file's permissions, the kernel first produces an appropriate status reply. If the requesting process is neither the owner of the file whose permissions it is trying to modify nor a superuser, or the level of the file does not dominate the level of the requesting process, the kernel ignores the request. Otherwise, it resets the permissions mask and the setuid and setgid bits in the file's inode table entry as requested.

```
(^reqresponse mstate rp (chmod rf pmask uidbit gidbit) =
  (Send (Statport rp (^stat_rep rp (chmod rf pmask uidbit gidbit)));;
  (If ((~((i_uid (^olditb rf)) = (p_realuid (^oldptb rp))) /\
        ~(super rp)) \/
       ~(^rf_dominates_rp))
   (Call (Minix mstate)) (* then cannot carry out request *)
   (Call                 (* else can carry out request *)
     (Minix
       (let (m_perm_r, m_perm_w, m_perm_x) = pmask in
        let newitb f =
         ((f = rf) =>
           (update_i_perm_r m_perm_r
           (update_i_perm_w m_perm_w
           (update_i_perm_x m_perm_x
           (update_i_setuid uidbit
           (update_i_setgid gidbit
             (^olditb rf))))))
```

```
          |
           (^olditb f)) in
          (update_s_itb newitb mstate)))))) /\
```

Specification for `chown` For a `chown` request, which changes a file's permissions, the kernel first produces an appropriate status reply. If the requesting process is not a superuser, or the level of the file does not dominate the level of the requesting process, the kernel ignores the request. Otherwise, it resets the user ID and group ID values in the file's inode table entry as requested.

```
(^reqresponse mstate rp (chown rf newuid newgid) =
  (Send (Statport rp (^stat_rep rp (chown rf newuid newgid))));;
  (If (~(super rp) \/
       ~(^rf_dominates_rp))
   (Call (Minix mstate))  (* then cannot carry out request *)
   (Call                  (* else can carry out request *)
     (Minix
       (let newitb f =
         ((f = rf) =>
           (update_i_uid newuid
           (update_i_gid newgid
             (^olditb rf)))
         |
          (^olditb f)) in
        (update_s_itb newitb mstate)))))) /\
```

Specification for `close` For a `close` request, the kernel first produces an appropriate status reply. If the access table entry for the requesting process and requested file does not indicate that this process has the file open, the kernel ignores the request. Otherwise, it decrements the access count in the access table entry, and if this makes the count 0, resets the `in_use` flag to "false".

```
(^reqresponse mstate rp (close rf) =
  (Send (Statport rp (^stat_rep rp (close rf))));;
  (If (~(^rp_using_rf))
   (Call (Minix mstate))  (* then cannot carry out request *)
   (Call                  (* else can carry out request *)
     (Minix
       (let oldaccesscount = a_lcount (^oldatb rp rf) in
let newatb p f =
 (((p = rp) /\ (f = rf)) =>
   ((oldaccesscount = 1) =>
     (update_a_in_use F
     (update_a_lcount 0
       (^oldatb rp rf)))
    |
     (update_a_lcount (oldaccesscount - 1)
       (^oldatb rp rf)))
 |
  (^oldatb p f)) in
(update_s_atb newatb mstate)))))) /\
```

Specification for create For a create request, which creates a new file or truncates an existing one, the kernel first produces an appropriate status reply. If the file already exists, and the level of the file does not dominate the level of the requesting process, the file is not writable, or the file is a directory, the kernel ignores the request. If the file does not exist, and there is no room for a new entry in the inode or access tables, the kernel also ignores the request.

Otherwise, the kernel creates or modifies the inode table entry for the created or truncated file, resetting the modification time, the number of links, the zone, and the count of times the inode is used. If it creates a new file, the kernel sets the file's inode permission, mode, uid, and gid bits as requested, except that it only sets the read, write, and execute permissions "true" if the requesting process' default permissions mask also has these values set "true", and it gives the newly created file the security level of the process creating it.

Whether it creates or truncates the file, the kernel resets the file's access table entry for the requesting process to indicate that the file is in use, readable, writable, empty, and indexed from the beginning, and resets the file's file table entry to indicate that it is empty.

```
(^reqresponse mstate rp (create rf cmask) =
  (Send (Statport rp (^stat_rep rp (create rf cmask))));;
  (If ((i_in_use (^olditb rf)) =>
         (~(^rf_dominates_rp) \/
          ~(i_perm_w (^olditb rf)) \/
          (i_mode_d (^olditb rf)))
      |
      (~(room_in_inode_table mstate) \/
       ~(room_in_access_table mstate)))
(Call (Minix mstate))  (* then cannot carry out request *)
(Call                  (* else can carry out request *)
  (Minix
    (let (m_perm_r, m_perm_w, m_perm_x,
          m_mode_d, m_mode_f, m_setuid, m_setgid) = cmask in
     let (p_perm_r, p_perm_w, p_perm_x) = (p_mask (^oldptb rp)) in
     let newitb f =
      ((f = rf) =>
        (update_i_modtime date
        (update_i_nlinks 1
        (update_i_zone (^new_zone rf mstate)
        (update_i_count 1
         ((i_in_use (^olditb rf)) =>
           (^olditb rf)
         |
          (update_i_in_use T
          (update_i_perm_r (m_perm_r /\ p_perm_r)
          (update_i_perm_w (m_perm_w /\ p_perm_w)
          (update_i_perm_x (m_perm_x /\ p_perm_x)
          (update_i_mode_d m_mode_d
          (update_i_mode_f m_mode_f
          (update_i_setuid m_setuid
          (update_i_setgid m_setgid
          (update_i_uid (p_effuid (^oldptb rp))
          (update_i_gid (p_effgid (^oldptb rp))
          (update_i_level (p_level (^oldptb rp))
            empty_i)))))))))))))))
     |
      (^olditb f)) in
    let newatb p f =
     (((p = rp) /\ (f = rf)) =>
       (update_a_in_use T
       (update_a_write T
       (update_a_read T
       (update_a_lcount 0
       (update_a_pos 0
         empty_a)))))
     |
      (^oldatb p f)) in
    let newftb f =
     ((f = rf) =>
        empty_file_contents
        |
         (~oldftb f)) in
        (update_s_itb newitb
        (update_s_atb newatb
        (update_s_ftb newftb
         mstate))))))) /\
```

Specification for `exec` For the `exec` request to execute a requested file on a supplied argument list, the kernel generates only a status reply to the requesting process and otherwise ignores the request if the requesting process does not have execute permission for the requested file, if there is not sufficient space to load the requested file into memory, or if the number or types of the supplied arguments are invalid. The status reply should indicate "file not found", regardless of the actual problem, if the level of the requesting process does not dominate the level of the requested file.

Otherwise, if the level of the requesting process dominates the level of the requested file, the kernel replaces the requesting process' memory image with the image of the file to be executed. The kernel replaces the image by sending the requesting process the image it is to replace itself with as a non-status reply to the `exec` request, and by replacing one or both of the requesting process' effective user and group IDs, doing so as determined by the `setuid` and `setgid` flags in the executed file's inode table entry.

Again otherwise, if the level of the requesting process does not dominate the level of the requested file, the kernel sends the requesting process a status reply indicating "file not found" and checks whether there is sufficient space in the process table for a new entry and sufficient space in memory for a new process. If not, the `exec` request has no further effect. If so, the kernel executes the requested file as if its execution had been requested by a dummy process having the same level as the file, but gives the fields in the new process' process-table entry, which normally are given the corresponding values for the parent process, the corresponding values for the requesting process. The kernel sends a status reply to the newly created process.

```
 (~reqresponse mstate rp (exec rf arglist) =
   (If (~(execute_okay rp rf mstate) \/
        ~(space_to_load rf mstate) \/
        ~(args_okay arglist rf))

    (* then request impossible to carry out *)
```

```
((Send (Statport rp (^stat_rep rp (exec rf arglist))));;
 (Call (Minix mstate))

(* else request possible to carry out *)

(If (^rp_dominates_rf)

 (* file low case *)

 ((Send (Outport rp (execreply (image rf arglist))));;
  (Call
    (Minix
     (let newptb p =
       ((p = rp) =>
          (((i_setuid (^olditb rf))/\(i_setgid (^olditb rf))) =>
            (update_p_effuid (process_effuid (^image rf arglist))
            (update_p_effgid (process_effgid (^image rf arglist))
              (^oldptb rp)))
          |
           ((i_setuid (^olditb rf))/\(~(i_setgid (^olditb rf)))) =>
            (update_p_effuid (process_effuid (^image rf arglist))
              (^oldptb rp))
          |
           ((~(i_setuid (^olditb rf)))/\(i_setgid (^olditb rf))) =>
            (update_p_effgid (process_effgid (^image rf arglist))
              (^oldptb rp))
          |
           (^oldptb rp))
       |
        (^oldptb p)) in
      (update_s_ptb newptb mstate)))))

 (* file not low case *)

 ((Send (Statport rp (^stat_rep rp (exec rf arglist))));;
  (If (~(room_in_process_table mstate) \/
       ~(space_for_new_process mstate))

   (* impossible to fork child case *)

   (Call (Minix mstate))

   (* possible to fork child case *)
((Send
   (Outport
     (new_process mstate)
     (execreply (image rf arglist))));;
 (Send
   (Statport
     (new_process mstate)
     (^stat_rep (new_process mstate) fork)));;
 (Call
   (Minix
    (let child = new_process mstate in
     let level = (i_level (^olditb rf)) in
     let parent = dummy_process level in
     let newptb p =
      ((p = child) =>
        (update_p_segs (allocate_process_segments child mstate)
        (update_p_scheduled T
        (update_p_in_use T
        (update_p_pid (process_id child)
        (update_p_parentid (p_pid (^oldptb parent))
        (update_p_level level
          (^oldptb rp)))))))
      |
       (^oldptb p)) in
     let newatb p f =
      ((p = child) =>
        ((a_in_use (^oldatb parent f)) =>
           (update_a_lcount ((a_lcount (^oldatb parent f)) + 1)
             (^oldatb parent f))
           |
            (^oldatb parent f))
       |
        (^oldatb p f)) in
      (update_s_ptb newptb
      (update_s_atb newatb
        mstate)))))))))) /\
```

Specification for `exit` For the `exit` request, if the requesting process' parent is waiting, the kernel terminates the requesting process by setting its process-table `in_use` value to "false". If the requesting process' parent is not waiting, the kernel terminates the requesting process by setting its process-table entry's hanging bit on and alarm and scheduling bits off. In either case, the kernel turns off the waiting bit for the requesting process' parent.

```
 (^reqresponse mstate rp (exit) =
   (Call
     (Minix
      (let parent = ^id_process (p_parentid (^oldptb rp)) in
       let newptb p =
        ((p = rp) =>
          ((p_waiting (^oldptb parent)) =>
            (update_p_in_use F (^oldptb rp))
           |
           (update_p_hanging T
           (update_p_alarm_on F
           (update_p_scheduled F
            (^oldptb rp)))))
         |
         (p = parent) =>
           (update_p_waiting F (^oldptb parent))
        |
         (^oldptb p)) in
       (update_s_ptb newptb mstate))))) /\
```

Specification for fork For the **fork** request, the kernel first sends a status reply to the requesting process. If there is not room in the process table for another entry, or if there is not sufficient space in memory for a new process, the kernel ignores the request. If there is sufficient space, the kernel creates a new process, making appropriate additions to the process and access tables.

```
 (^reqresponse mstate rp (fork) =
   (Send (Statport rp (^stat_rep rp fork)));;
   (If (~(room_in_process_table mstate) \/
        ~(space_for_new_process mstate))

    (* then cannot carry out request *)

    (Call (Minix mstate))

    (* else can carry out request *)

    ((Send (Outport rp (forkreply (new_process mstate))));;
     (Send
      (Statport
        (new_process mstate)
      (^stat_rep (new_process mstate) fork)));;
   (Call
     (Minix
      (let child = new_process mstate in
       let newptb p =
        ((p = child) =>
          (update_p_segs (allocate_process_segments child mstate)
          (update_p_scheduled T
          (update_p_in_use T
          (update_p_pid (process_id child)
          (update_p_parentid (p_pid (^oldptb rp))
            (^oldptb rp))))))
        |
         (^oldptb p)) in
       let newatb p f =
        ((p = child) =>
         ((a_in_use (^oldatb rp f)) =>
            (update_a_lcount ((a_lcount (^oldatb rp f)) + 1)
               (^oldatb rp f))
              |
               (^oldatb rp f))
         |
          (^oldatb p f)) in
       (update_s_ptb newptb
       (update_s_atb newatb
         mstate))))))) /\
```

Specification for `fstat` For the `fstat` request, the kernel first sends a status reply to the requesting process. If the requesting process does not have the requested file open, or if the level of the requesting process does not dominate the level of the requested file, the kernel ignores the request. (Note that, as in `exec`, the appropriate status reply if the level of the requesting process does not dominate the level of the requested file is "file not found".) If neither of these conditions hold, the kernel sends the requested status information for the requested file to the requesting process.

```
 (^reqresponse mstate rp (fstat rf) =
   (Send (Statport rp (^stat_rep rp (fstat rf)));;
   (If (~(^rp_using_rf) \/
        ~(^rp_dominates_rf))
    Skip       (* then request impossible to carry out *)
               (* else request possible to carry out *)
    (Send (Outport rp (statreply (file_status (^olditb rf))))));;
   (Call (Minix mstate))) /\
```

Specification for getgid For the getgid request, the kernel sends a status reply to the requesting process and then sends it the requested group ID information about itself.

```
 (^reqresponse mstate rp (getgid) =
   (Send (Statport rp (^stat_rep rp (getgid))));;
   (Send (Outport rp (getgidreply (p_effgid (^oldptb rp)))));;
   (Call (Minix mstate))) /\
```

Specification for getuid For the getuid request, the kernel sends a status reply to the requesting process and then sends it the requested user ID information about itself.

```
 (^reqresponse mstate rp (getuid) =
   (Send (Statport rp (^stat_rep rp (getuid))));;
   (Send (Outport rp (getuidreply (p_effuid (^oldptb rp)))));;
   (Call (Minix mstate))) /\
```

Specification for ioctl For the ioctl request, the kernel first sends a status reply to the requesting process. If the requesting process does not have the requested file open, if the requested file is not a special file, or if the level of the requested file does not dominate the level of the requesting process, the kernel makes no further response. Otherwise, it makes the requested changes to the file's IOCTL values.

```
 (^reqresponse mstate rp (ioctl rf n) =
   (Send (Statport rp (^stat_rep rp (ioctl rf n))));;
   (If (~(^rp_using_rf) \/
        ~(i_mode_f (^olditb rf)) \/
        ~((^rf_dominates_rp)))
    (Call (Minix mstate))  (* then cannot carry out request *)
    (Call                  (* else can carry out request *)
      (Minix
        (let newitb f =
          ((f = rf) =>
            (update_i_ioctl n (^olditb rf))
          |
           (^olditb f)) in
         (update_s_itb newitb mstate)))))) /\
```

Specification for `kill` For the `kill` request by a requesting process to kill a "victim" process, the kernel first sends a status reply to the requesting process. If the requesting process does not have the same real user ID as the victim, if the victim is hanging, if the victim's "ignore kill requests" flag is set, or if the victim's level does not dominate the level of the requesting process, the kernel makes no further response. Otherwise, the kernel kills the victim process. If the victim is set to catch the kill signal, the kernel kills it by sending it the `catch_sig` signal, then resets it so that it is no longer set to catch this signal. If the victim is not set to catch the kill signal, the kernel kills it in one of two ways, depending on whether its parent process is hanging, and then resets its parent so that this parent is not waiting. If the victim process' parent is hanging, the kernel terminates the victim by setting the victim's `in_use` value to "false"; otherwise, the kernel terminates the victim by setting its hanging bit on and its alarm and scheduling bits off.

```
(^reqresponse mstate rp (kill victim) =
  (Send (Statport rp (^stat_rep rp (kill victim))));;
  (If (~(p_realuid (^oldptb victim) = p_realuid (^oldptb rp)) \/
       (p_hanging (^oldptb victim)) \/
       (p_ignore (^oldptb victim)) \/
       ~(dom (p_level (^oldptb victim)) (p_level (^oldptb rp))))

   (* then request impossible to carry out *)

   (Call (Minix mstate))

   (* else request possible to carry out *)

   ((If (p_catch (^oldptb victim))
      (Send (Outport victim (killreply catch_sig)))
      Skip);;
    (Call
      (Minix
        (let parent =
          (id_process (p_parentid (^oldptb victim))) in
         let newptb p =
          ((p = victim) =>
             ((~(p_catch (^oldptb victim))) =>
               ((p_hanging (^oldptb parent)) =>
                (update_p_in_use F (^oldptb victim))
               |
                (update_p_hanging T
            (update_p_alarm_on F
            (update_p_scheduled F
              (^oldptb victim)))))
        |
         (update_p_catch F (^oldptb victim))
    |
     (p = parent) =>
       ((~(p_catch (^oldptb victim))) =>
         (update_p_waiting F (^oldptb parent))
        |
         (^oldptb parent))
    |
     (^oldptb p)) in
   (update_s_ptb newptb mstate))))))) /\
```

Specification for `lseek` For the `lseek` request to reposition the index into a file, the kernel first sends a status reply to the requesting process. If the requesting process does not have the requested file open, the kernel makes no further response. Otherwise, it makes the requested change in the file's access table entry.

```
(^reqresponse mstate rp (lseek rf position) =
  (Send (Statport rp (^stat_rep rp (lseek rf position))));;
  (If (~(^rp_using_rf))
   (Call (Minix mstate))  (* then cannot carry out request *)
   (Call                  (* else can carry out request *)
     (Minix
       (let newatb p f =
         (((p = rp) /\ (f = rf)) =>
           (update_a_pos (new_pos rf mstate position)
             (^oldatb rp rf))
          |
          (^oldatb p f)) in
        (update_s_atb newatb mstate)))))) /\
```

Specification for `mknod` For the `mknod` request to create a special file, the kernel first sends a status reply to the requesting process. If the file already exists, or there is no room for a new entry in the inode or access tables, or the requesting process is not a superuser process, the kernel makes no further response.

Otherwise, the kernel creates the inode table entry for the new file, setting the modification time, the number of links, the zone, and the count of times the inode is used. The kernel sets the file's inode permission, mode, uid, and gid bits as requested, except that it only sets the read, write, and execute permissions "true" if the requesting process' default permissions mask also has these values set "true", and it gives the newly created file the security level of the process creating it.

The kernel resets the file's access table entry for the requesting process to indicate that the file is in use, readable, writable, empty, and indexed from the beginning, and resets the file's file table entry to indicate that it is empty.

```
(^reqresponse mstate rp (mknod rf cmask) =
  (Send (Statport rp (^stat_rep rp (mknod rf cmask))));;
  (If ((i_in_use (^olditb rf)) \/
       ~(room_in_inode_table mstate) \/
       ~(room_in_access_table mstate)\/
       ~(super rp))
   (Call (Minix mstate)) (* then cannot carry out request *)
   (Call                 (* else can carry out request *)
      (Minix
        (let (m_perm_r, m_perm_w, m_perm_x,
              m_mode_d, m_mode_f, m_setuid, m_setgid) = cmask in
         let (p_perm_r,p_perm_w,p_perm_x) = (p_mask (^oldptb rp)) in
         let newitb f =
          ((f = rf) =>
             (update_i_in_use T
             (update_i_perm_r (m_perm_r /\ p_perm_r)
             (update_i_perm_w (m_perm_w /\ p_perm_w)
             (update_i_perm_x (m_perm_x /\ p_perm_x)
             (update_i_mode_d m_mode_d
             (update_i_mode_f m_mode_f
             (update_i_setuid m_setuid
             (update_i_setgid m_setgid
             (update_i_modtime date
             (update_i_uid (p_effuid (^oldptb rp))
             (update_i_gid (p_effgid (^oldptb rp))
             (update_i_nlinks 1
             (update_i_zone (^new_zone rf mstate)
             (update_i_count 1
             (update_i_level (p_level (^oldptb rp))
               empty_i)))))))))))))))
          |
           (^olditb f)) in
let newatb p f =
 (((p = rp) /\ (f = rf)) =>
   (update_a_in_use T
   (update_a_write T
   (update_a_read T
   (update_a_lcount 0
   (update_a_pos 0
     empty_a)))))
 |
  (^oldatb p f)) in
let newftb f =
 ((f = rf) =>
    empty_file_contents
 |
  (^oldftb f)) in
(update_s_itb newitb
(update_s_atb newatb
(update_s_ftb newftb
  mstate))))))) /\
```

Specification for open For the open request, the kernel first sends a status reply to the requesting process. If the file does not exist, if there is no room in the access table for another entry, if one of the permissions requested in the open is not one of the permissions possessed by the requesting process, or if the level of the requesting process does not dominate the level of the requested file, the kernel makes no further response. Otherwise, the kernel carries out the request by making appropriate modifications in the access table.

```
(^reqresponse mstate rp (open rf pmask) =
  (Send (Statport rp (^stat_rep rp (open rf pmask))));;
  (If (let (m_perm_r, m_perm_w, m_perm_x) = pmask in
       ~(i_in_use (^olditb rf)) \/
       ~(room_in_access_table mstate) \/
       (m_perm_x /\ ~(i_perm_x (^olditb rf))) \/
       (m_perm_w /\
         (~(i_perm_w (^olditb rf)) \/
          ~(^rf_dominates_rp) \/
         (i_mode_d (^olditb rf)))) \/
       (m_perm_r /\
         (~(i_perm_r (^olditb rf)) \/
       ~(^rp_dominates_rf))))
(Call (Minix mstate)) (* then cannot carry out request *)
(Call                 (* else can carry out request *)
   (Minix
     (let (m_perm_r, m_perm_w, m_perm_x) = pmask in
      let newatb p f =
       (((p = rp) /\ (f = rf)) =>
         (update_a_in_use T
         (update_a_lcount 1
         (update_a_pos 0
         (m_perm_w =>
           (update_a_write T
              (m_perm_r =>
                (update_a_read T empty_a)
              |
               empty_a))
          |
           (m_perm_r =>
             (update_a_read T empty_a)
           |
            empty_a)))))
       |
        (^oldatb p f)) in
      (update_s_atb newatb mstate)))))) /\
```

Specification for `pause` For the `pause` request, the kernel first sends a status reply to the requesting process, then sets the `paused` flag in that process' process-table entry to "true".

```
(^reqresponse mstate rp (pause) =
  (Send (Statport rp (^stat_rep rp pause));;
  (Call
    (Minix
     (let newptb p =
       ((p = rp) =>
          (update_p_paused T (^oldptb rp))
        |
         (^oldptb p)) in
      (update_s_ptb newptb mstate))))) /\
```

Specification for `read` For the `read` request, the kernel first sends a status reply to the requesting process. If the access table shows that the requesting process does not have read access to the requested file, the kernel makes no further response, but otherwise it returns the requested file contents. (Note that the model does not cover partial reads; these reads raise no additional nondisclosure issues.)

```
(^reqresponse mstate rp (read rf) =
  (Send (Statport rp (^stat_rep rp (read rf))));;
  (If (~(a_read (^oldatb rp rf)))
   Skip
   (Send (Outport rp (readreply (^oldftb rf))));;
  (Call (Minix mstate))) /\
```

Specification for setgid For the setgid request, the kernel first sends a status reply to the requesting process. If the requesting process is not a superuser process or if the group ID in the request is not the requesting process' real or effective group ID, the kernel makes no further response. Otherwise, the kernel resets the requesting process' group ID accordingly.

```
(^reqresponse mstate rp (setgid n) =
  (Send (Statport rp (^stat_rep rp (setgid n))));;
  (If (~(super rp) /\
       ~(n = (p_realgid (^oldptb rp))) /\
       ~(n = (p_effgid (^oldptb rp))))
   (Call (Minix mstate))  (* then cannot carry out request *)
   (Call                  (* else can carry out request *)
     (Minix
      (let newptb p =
        ((p = rp) =>
           (update_p_realgid n
           (update_p_effgid n
             (^oldptb rp)))
        |
         (^oldptb p)) in
       (update_s_ptb newptb mstate))))) /\
```

Specification for setuid For the setuid request, the kernel first sends a status reply to the requesting process. If the requesting process is not a superuser process or if the user ID in the request is not the requesting process' real or effective user ID, the kernel makes no further response. Otherwise, the kernel resets the requesting process' group ID accordingly.

```
(^reqresponse mstate rp (setuid n) =
(Send (Statport rp (^stat_rep rp (setuid n))));;
(If (~(super rp) /\
     ~(n = (p_realuid (^oldptb rp))) /\
     ~(n = (p_effuid (^oldptb rp))))
 (Call (Minix mstate))  (* then cannot carry out request *)
 (Call                  (* else can carry out request *)
   (Minix
    (let newptb p =
      ((p = rp) =>
         (update_p_realuid n
         (update_p_effuid n
           (^oldptb rp)))
      |
       (^oldptb p)) in
     (update_s_ptb newptb mstate)))))) /\
```

Specification for `signal` For the `signal` request, the kernel first sends a status reply to the requesting process, then resets the `catch` or `ignore` flags for the requesting process in response to the corresponding signals.

```
(^reqresponse mstate rp (signal n) =
  (Send (Statport rp (^stat_rep rp (signal n))));;
  (Call
    (Minix
     (let newptb p =
       ((p = rp) =>
        ((n = sig_catch) =>
          (update_p_catch T (^oldptb rp))
         |
         (n = sig_ignore) =>
          (update_p_ignore T (^oldptb rp))
         |
          (^oldptb rp))
        |
         (^oldptb p)) in
      (update_s_ptb newptb mstate))))) /\
```

Specification for `stat` For the `stat` request, which requests the status of a file, the kernel first sends a status reply to the requesting process. If the requesting process does not have the requested file open, or if the level of the requesting process does not dominate the level of the requested file, the kernel makes no further response. Otherwise, it replies with the requested status information.

```
(^reqresponse mstate rp (stat rf) =
  (Send (Statport rp (^stat_rep rp (stat rf))));;
  (If (~(^rp_using_rf) \/
       ~(^rp_dominates_rf))
   Skip        (* then request impossible to carry out *)
               (* else request possible to carry out *)
   (Send (Outport rp (statreply (file_status (^olditb rf)))));;
  (Call (Minix mstate))) /\
```

Specification for `stime` For the `stime` request, which sets the system time, the kernel first sends a status reply to the requesting process. If the requesting process is not a superuser process, the kernel makes no further response. Otherwise, it resets the system time as requested.

```
(^reqresponse mstate rp (stime newtime) =
  (Send (Statport rp (^stat_rep rp (stime newtime))));;
  (If (~(super rp) \/ ~((p_level (^oldptb rp)) = systemlow))
    (Call (Minix mstate))  (* then cannot carry out request *)
    (Call                  (* else can carry out request *)
       (Minix
         (update_s_systemtime newtime mstate))))) /\
```

Specification for `time` For the `time` request, the kernel sends a status reply to the requesting process and then sends the requested system time.

```
(^reqresponse mstate rp (time) =
  (Send (Statport rp (^stat_rep rp (time))));;
  (Send (Outport rp (timereply (s_systemtime mstate))));;
  (Call (Minix mstate))) /\
```

Specification for `times` For the `times` request, the kernel sends a status reply to the requesting process, then sends the requesting process the requested information about the times involved in its own execution.

```
(^reqresponse mstate rp (times) =
  (Send (Statport rp (^stat_rep rp (times))));;
  (Send (Outport rp (timesreply (used_times rp mstate))));;
  (Call (Minix mstate))) /\
```

Specification for umask For the umask request, the kernel sends a status reply to the requesting process, then sends the requesting process the requested information about its own file-creation mask.

```
(^reqresponse mstate rp (umask pmask) =
  (Send (Statport rp (^stat_rep rp (umask pmask))));;
  (Send (Outport rp (umaskreply (p_mask (^oldptb rp)))));;
  (Call
    (Minix
     (let newptb p =
       ((p = rp) =>
          (update_p_mask pmask (^oldptb rp))
       |
        (^oldptb p)) in
      (update_s_ptb newptb mstate))))) /\
```

Specification for utime For the utime request, which modifies the last-modification time for a file, the kernel first sends a status reply to the requesting process. If the requesting process' level does not dominate the level of the requested file, or if the requesting process is not a superuser process and the requesting process is not the owner of the requested file, the kernel makes no further response. Otherwise, it sets the "dirty" flag for the requested file to "true" and changes the file's last-modification time.

```
(^reqresponse mstate rp (utime rf newtime) =
  (Send (Statport rp (^stat_rep rp (utime rf newtime))));;
  (If (~(^rp_dominates_rf) \/
       (~(super rp) /\
        ~((i_uid (^olditb rf)) = (p_realuid (^oldptb rp)))))
   (Call (Minix mstate))  (* then cannot carry out request *)
   (Call                  (* else can carry out request *)
     (Minix
       (let newitb f =
         ((f = rf) =>
           (update_i_dirt T
           (update_i_modtime newtime
            (^olditb rf)))
         |
          (^olditb f)) in
        (update_s_itb newitb mstate)))))) /\
```

Specification for `wait` For the `wait` request, the kernel first sends a status reply to the requesting process. It then terminates all the hanging child processes of the requesting process by resetting their `in_use` values to "false", then sets the `waiting` flag for the requesting process to "true".

```
(^reqresponse mstate rp (wait) =
  (Send (Statport rp (^stat_rep rp wait));;
  (Call
    (Minix
     (let newptb p =
       (((rp = (id_process (p_parentid (^oldptb p)))) /\
         (p_hanging (^oldptb p))) =>
         (update_p_in_use F (^oldptb p))
       |
        (p = rp) =>
         (update_p_waiting T (^oldptb rp))
       |
         (^oldptb p)) in
      (update_s_ptb newptb mstate))))) /\
```

Specification for `write` For the `write` request, the kernel first sends a status reply to the requesting process. If the requesting process does not have write access to the requested file, the kernel makes no further response. Otherwise, it replaces the contents of the requested file with the requested value. (Note that the model does not cover partial writes; these writes raise no additional nondisclosure issues.)

```
(^reqresponse mstate rp (write rf contents) =
  (Send (Statport rp (^stat_rep rp (write rf contents))));;
  (If (~(a_write (^oldatb rp rf)))
   (Call (Minix mstate))  (* then cannot carry out request *)
   (Call                  (* else can carry out request *)
     (Minix
       (let newftb f =
         ((f = rf) =>
            contents
         |
          (^oldftb f)) in
        (update_s_ftb newftb mstate))))))
```

Final Lines The final lines of the `reqresponse` definition simply terminate the definition.

```
    '--
    end};
```

Function minixResponse

The function `minixResponse` gives the response of the kernel process to an arbitrary input event. The function "takes apart" the input event, obtaining the requesting process and its `MinixRequest`, then calls `reqresponse` with this information.

```
  new_recursive_definition {
   name = "minixResponse",
   fixity = Prefix,
   rec_axiom = MinixInEv_Def,
   def =
    let
     val minixResponse =
      --'minixResponse:^MinixState -> ^MinixInEv -> ^MinixProc'--;
     val reqresponse =
      --'reqresponse:
          ^MinixState->'UserProcess->^MinixRequest->^MinixProc'--;
    in
     --'
      (^minixResponse mstate (Inport rp request) =
         (^reqresponse mstate rp request))
     '--
    end};
```

Function minix

The function `minix` gives the top-level description of the kernel process. The function is simple: the kernel waits for an arbitrary input event, then invokes `MinixResponse` to determine its response as a function of its state parameter and this input event. (The `Receive` PSL command supplies the input event received, implicitly taken off a buffer, as an parameter to the function invoked.)

```
  new_definition(
   "minix",
   let
    val minix = --'minix:^MinixState -> ^MinixProc'--;
   in
```

```
 --`
  ^minix mstate =
    (Receive (\ev:^MinixInEv. T) (MinixResponse mstate))
  `--
end);
```

Function MinixInvocVal

The function MinixInvocVal maps every invocation to the corresponding value of a PSL-valued function, mapping the invocation constructor Minix to the function minix and the invocation constructor MinixResponse to the function minixResponse.

```
new_recursive_definition {
 name = "MinixInvocVal",
 fixity = Prefix,
 rec_axiom = MinixInvoc_Def,
 def =
  let
   val MinixInvocVal =
    --`MinixInvocVal:^MinixInvoc -> ^MinixProc`--
  in
   --`
    (^MinixInvocVal (Minix mstate) =
       (minix mstate)) /\

    (^MinixInvocVal (MinixResponse mstate inev) =
       (minixResponse mstate inev))
   `--
  end};
```

3.3 Proofs

This section contains and describes the HOL90 SML code formally proving security properties of the kernel model specified in section 3.2. It proves that the kernel model is a server process and that every output produced by the kernel model (before it loops back to wait for the next input) in response to an input event is at a security level greater than or equal to the security level of this input event.

The remainder of this section gives the proofs themselves, but does not give HOL90's actual replies. That information is given in Appendix 3.A.

3.3.1 Tactics

The proof begins by defining a special-purpose tactic `request_cases_TAC`. This tactic expands applications of the `minixResponse` subroutine `reqresponse` to an arbitrary `MinixRequest`, generating each of the 30 possible cases. The variable `m12_MinixRequest` is a name that will be introduced later by Romulus tactics for the second data entry in the first (and only) type of input event. This name is chosen to show the type of this entry, which is an arbitrary `MinixRequest`.

```
val request_cases_TAC =
 X_CASES_THEN
  [
   [],
   [],
   [--`rf:'File`--,--`pmask:^PermissionsMask`--,
    --`uidbit:bool`--,--`gidbit:bool`--],
   [--`rf:'File`--,--`newuid:num`--,--`newgid:num`--],
   [--`rf:'File`--],
   [--`rf:'File`--,--`cmask:^CreateMask`--],
   [--`rf:'File`--,--`arglist:('Arg)list`--],
   [],
   [],
   [--`rf:'File`--],
   [],
   [],
   [--`rf:'File`--,--`n:num`--],
   [--`victim:'UserProcess`--],
   [--`rf:'File`--,--`position:num`--],
   [--`rf:'File`--,--`cmask:^CreateMask`--],
   [--`rf:'File`--,--`pmask:^PermissionsMask`--],
   [],
   [--`rf:'File`--],
   [--`n:num`--],
   [--`n:num`--],
   [--`n:num`--],
   [--`rf:'File`--],
   [--`newtime:num`--],
   [],
   [],
   [--`pmask:^PermissionsMask`--],
   [--`rf:'File`--,--`newtime:num`--],
   [],
  [--`rf:'File`--,--`contents:'FileContents`--]
 ]
 (fn th =>
  REWRITE_TAC
   [th,
   (definition "-" "reqresponse")])
 (SPEC
   (--`request:^MinixRequest`--)
   (prove_cases_thm (prove_induction_thm MinixRequest_Def)));
```

3.3.2 Proof of BPSP_rightform

This section proves that the kernel model is a server process by showing that the predicate BPSP_rightform holds for the state parameter invariant MinixInvariant, the function assigning meanings to invocations MinixInvocVal, and the parameterized process (more accurately, invocation constructor) Minix. The result proved is polymorphic, so it holds for any types substituted for the type variables 'UserProcess, 'File, 'FileContents, 'Arg, and 'Level.

The proof begins by setting the goal in appropriate generality:

```
g(`BPSP_rightform
    (MinixInvariant:^MinixState -> bool)
    (MinixInvocVal:^MinixInvoc -> ^MinixProc)
    (Minix:^MinixState -> ^MinixInvoc)`);
```

It then uses BPSP_rightform_TAC to expand the definition of BPSP_-rightform and the invariant, and to confirm automatically that Minix names a parameterized process that receives arbitrary input events.

```
e(BPSP_rightform_TAC);
```

The remaining goal asserts that if MinixInvariant holds of param, then the following Loopsback condition holds, which says that reqresponse for the state parameter param (the user process that is the first data object in an arbitrary input event) and the MinixRequest (the second data object in an arbitrary input event) evaluates to a PSL object that ends with a call back to Minix (the invocation naming the kernel process as it waits for another input event).

```
Loopsback
 MinixInvocVal
 (INR Minix)
 (reqresponse param m11_UserProcess m12_MinixRequest)
```

Since the invariant hypothesis is not needed to prove the Loopsback result, the next line in the proof discards it.

```
e(DISCH_THEN (fn th => ALL_TAC));
```

The tactic request_cases_TAC then expands the goal into 30 subgoals, each asserting that the Loopsback condition holds for the response to one form of MinixRequest. Each of these goals can be proved by rewriting with the standard Romulus theorems that effectively define Loopsback and Terminates by structural induction on PSL objects, then rewriting with a trivial lemma saying that if both branches of a conditional expression have value "true" then the conditional expression has value "true". The HOL tactical THEN automatically applies tactics to all remaining subgoals, so

```
e(request_cases_TAC THEN
  REWRITE_TAC
   [theorem "romsecure" "Loopsback_Rewrites",
    theorem "romsecure" "Terminates_Rewrites"] THEN
  REWRITE_TAC [theorem "romlemmas" "rom_condlemma1"]);
```

completes the proof.

The lines

```
save_top_thm "Minix_rightform";
export_theory();
```

save the result for future use and write the minix theory to the disk.

3.3.3 Proof of BPSP_nowritesdown

This section proves that the kernel model's responses to any input event are at levels greater than or equal to the level of the input event by proving that the predicate BPSP_nowritesdown holds for the function dom defining the dominance relation on security levels, the functions MinixInLevel and MinixOutLevel assigning security levels to input and output events, the state parameter invariant MinixInvariant, the function assigning meanings to invocations MinixInvocVal, and the parameterized process (more accurately, invocation constructor) Minix. The result proved is polymorphic, so it holds for any types substituted for the type variables 'UserProcess, 'File, 'FileContents, 'Arg, and 'Level.

The proof begins by setting the goal in appropriate generality:

```
g('BPSP_nowritesdown
    (dom:'Level -> 'Level -> bool)
    (MinixInLevel:^MinixState -> ^MinixInEv -> 'Level)
    (MinixOutLevel:^MinixState -> ^MinixOutEv -> 'Level)
    (MinixInvariant:^MinixState -> bool)
    (MinixInvocVal:^MinixInvoc -> ^MinixProc)
    (Minix:^MinixState -> ^MinixInvoc)');
```

It then uses BPSP_nowritesdown_TAC to expand the definition of BPSP_-nowritesdown and the invariant, and to move the properties given by the invariant into the goal's hypothesis list. The tactic makes no further automatic simplifications.

```
e(BPSP_nowritesdown_TAC);
```

As before, request_cases_TAC breaks the goal up into 30 subgoals. Rewriting with the theorems defining NoWritesDown by structural induction on PSL objects and the definitions of the level-assignment functions shows that most of these subgoals are trivially true because the level of all output events is the same as the level of the input event that caused them. Every security level dominates itself by the axiom dom_reflexive, so rewriting with this axiom and a trivial lemma saying that if both clauses of a conditional expression have value "true" the whole expression has value "true" proves these subgoals. The lines

```
e(request_cases_TAC THEN
  REWRITE_TAC [theorem "romsecure" "NoWritesDown_Rewrites"] THEN
  REWRITE_TAC
   [definition "-" "MinixOutLevel",
    definition "-" "MinixInLevel",
    axiom "-" "dom_reflexive",
    theorem "romlemmas" "rom_condlemma1"]);
```

leave only three subgoals.

Two of those three subgoals are true because there is no disclosure in creating, and thus modifying, a previously non-existent user process. Rewriting with the axiom saying that a new process-table entry always replaces the entry for a previously non-existent process, the definition that a non-existent process has level `systemhigh`, and the axiom that `systemhigh` dominates all levels proves these subgoals:

```
e(REWRITE_TAC
   [axiom "-" "new_process_new",
    definition "-" "empty_p",
    definition "-" "p_level",
    axiom "-" "systemhigh_high",
    theorem "romlemmas" "rom_condlemma1"]);

e(REWRITE_TAC
   [axiom "-" "new_process_new",
    definition "-" "empty_p",
    definition "-" "p_level",
    axiom "-" "systemhigh_high",
    theorem "romlemmas" "rom_condlemma1"]);
```

(The repetition could have been avoided by using these lines with `THEN` after executing `request_cases_TAC`, but we use it for expository purposes, causing the two subgoals proved to be displayed in the proof-transcript given in Appendix 3.A.

The remaining subgoal is true because the response to a `kill` request is defined so that a user process can only kill another user process if the level of that other process dominates the level of the user process killing it. The subgoal follows by considering the two cases as to whether the required dominance relation holds, then rewriting:

```
e(ASM_CASES_TAC
   (--`((dom:'Level -> 'Level -> bool)
         (p_level (s_ptb (param:^MinixState) (victim:'UserProcess)))
         (p_level (s_ptb param rp)))`--) THEN
  ASM_REWRITE_TAC[theorem "romlemmas" "rom_condlemma1"]);
```

The line

```
save_top_thm "Minix_nowritesdown";
```

saves the result for future use.

3.3.4 Remaining Work

Proofs of the following conditions about the kernel model's state parameters would complete the proof that the kernel model is restrictive:

1. The kernel's initial state parameter `MinixInitParam` satisfies the invariant `MinixInvariant`, and if the parameter `mstate` satisfies this invariant, then for every input event `inev` the kernel's response `minixResponse mstate inev` ends with a call back to the kernel with a new state parameter that also satisfies this invariant. (Informally, the invariant is preserved.)

2. For any security level `lev`, any input event `inev`, and any kernel state parameter `mstate` satisfying the invariant `MinixInvariant`, if `mstate'` is any new kernel state parameter after the kernel started with state parameter `mstate` and finished responded to event `inev`, then `MinixProjection lev mstate = MinixProjection lev mstate'`. (Informally, hidden inputs cause only hidden changes in state parameters.)

3. For any security level `lev`, any input event `inev`, and any kernel state parameters `mstate` and `mstate'` satisfying the invariant `MinixInvariant` such that `MinixProjection lev mstate = MinixProjection lev mstate'`, the following conditions hold of the kernel's responses to event `inev` when starting with state parameters `mstate` and `mstate'`:

   - The decisions made in simplifying PSL's conditional `If` expressions are the same. (Informally, the execution paths followed do not distinguish two previously indistinguishable state parameters.)
   - Every output event at a level dominated by `lev` produced by one response could also be produced by the other response. (Informally, the output sequences do not distinguish two previously indistinguishable state parameters.)
   - If `newmstate` is a possible new state parameter after the kernel started with state parameter `mstate`, and `newmstate'` is a new state parameter after the kernel started with state parameter `mstate'`, then `MinixProjection lev newmstate = MinixProjection lev newmstate'`. (Informally, the two next state parameters do not distinguish two previously indistinguishable state parameters.)

We did not complete these proofs because of lack of time, but expect the first of them to be relatively easy and the others to be tedious but straightforward.

3.A Appendix: Transcripts of Proofs

This appendix contains transcripts of the HOL90 sessions proving the results described in section 3.3. The user inputs are in the lines beginning with - and are in italic type.

Proof: Kernel Model a Server Process

The following is the transcript of the proof of the `BPSP_rightform` property for the kernel process:

```
- g(`BPSP_rightform
    (MinixInvariant:^MinixState -> bool)
    (MinixInvocVal:^MinixInvoc -> ^MinixProc)
    (Minix:^MinixState -> ^MinixInvoc)`);
(--`BPSP_rightform MinixInvariant MinixInvocVal Minix`--)
==============================


val it = () : unit
- e(BPSP_rightform_TAC);
OK..
1 subgoal:
(--`(!level. p_level (s_ptb param (dummy_process level)) = level) /\
    (!f.
      (s_ftb param f = empty_file_contents) ==>
      (i_level (s_itb param f) = systemlow)) /\
    (!p f.
      (a_read (s_atb param p f) ==>
       dom (p_level (s_ptb param p)) (i_level (s_itb param f))) /\
      (a_write (s_atb param p f) ==>
       dom (i_level (s_itb param f)) (p_level (s_ptb param p)))) ==>
    Loopsback MinixInvocVal (INR Minix) (reqresponse param rp request)`--)
==============================


val it = () : unit
- e(DISCH_THEN (fn th => ALL_TAC));
```

```
OK..
1 subgoal:
(--`Loopsback MinixInvocVal (INR Minix) (reqresponse param rp request)`--)
=============================



val it = () : unit
- e(request_cases_TAC THEN
  REWRITE_TAC
   [theorem "romsecure" "Loopsback_Rewrites",
    theorem "romsecure" "Terminates_Rewrites"] THEN
  REWRITE_TAC [theorem "romlemmas" "rom_condlemma1"]);
OK..

Goal proved.
|- Loopsback MinixInvocVal (INR Minix) (reqresponse param rp request)

Goal proved.
|- (!level. p_level (s_ptb param (dummy_process level)) = level) /\
   (!f.
     (s_ftb param f = empty_file_contents) ==>
     (i_level (s_itb param f) = systemlow)) /\
   (!p f.
     (a_read (s_atb param p f) ==>
      dom (p_level (s_ptb param p)) (i_level (s_itb param f))) /\
     (a_write (s_atb param p f) ==>
      dom (i_level (s_itb param f)) (p_level (s_ptb param p)))) ==>
   Loopsback MinixInvocVal (INR Minix) (reqresponse param rp request)

Goal proved.
|- BPSP_rightform MinixInvariant MinixInvocVal Minix

Top goal proved.
val it = () : unit
- save_top_thm "Minix_rightform";
val it = |- BPSP_rightform MinixInvariant MinixInvocVal Minix : thm
```

Proof: Output Levels Dominate Input Levels

The following is the transcript of the proof that the BPSP_nowritesdown property holds for the kernel process:

```
- g(`BPSP_nowritesdown
    (dom:'Level -> 'Level -> bool)
    (MinixInLevel:^MinixState -> ^MinixInEv -> 'Level)
    (MinixOutLevel:^MinixState -> ^MinixOutEv -> 'Level)
    (MinixInvariant:^MinixState -> bool)
    (MinixInvocVal:^MinixInvoc -> ^MinixProc)
    (Minix:^MinixState -> ^MinixInvoc)`);
(--`BPSP_nowritesdown dom MinixInLevel MinixOutLevel MinixInvariant
      MinixInvocVal
      Minix`--)
=============================


val it = () : unit
- e(BPSP_nowritesdown_TAC);
OK..
1 subgoal:
(--`NoWritesDown dom (MinixOutLevel param) (p_level (s_ptb param rp))
      MinixInvocVal
      (INR Minix)
      (reqresponse param rp request)`--)
=============================
  (--`!level. p_level (s_ptb param (dummy_process level)) = level`--)
  (--`!f.
      (s_ftb param f = empty_file_contents) ==>
      (i_level (s_itb param f) = systemlow)`--)
  (--`!p f.
      (a_read (s_atb param p f) ==>
       dom (p_level (s_ptb param p)) (i_level (s_itb param f))) /\
      (a_write (s_atb param p f) ==>
       dom (i_level (s_itb param f)) (p_level (s_ptb param p)))`--)


val it = () : unit
- e(request_cases_TAC THEN
  REWRITE_TAC [theorem "romsecure" "NoWritesDown_Rewrites"] THEN
  REWRITE_TAC
   [definition "-" "MinixOutLevel",
    definition "-" "MinixInLevel",
    axiom "-" "dom_reflexive",
    theorem "romlemmas" "rom_condlemma1"]);
OK..

[Major collection... 52% used (7425260/14278572), 6520 msec]
3 subgoals:
```

```
(--`(~(p_realuid (s_ptb param victim) = p_realuid (s_ptb param rp)) \/
     p_hanging (s_ptb param victim) \/
     p_ignore (s_ptb param victim) \/
     ~(dom (p_level (s_ptb param victim)) (p_level (s_ptb param rp))))
    => T
    | ((p_catch (s_ptb param victim))
       => (dom (p_level (s_ptb param victim)) (p_level (s_ptb param rp)))
       | T)`--)
==============================
  (--`!level. p_level (s_ptb param (dummy_process level)) = level`--)
  (--`!f.
      (s_ftb param f = empty_file_contents) ==>
      (i_level (s_itb param f) = systemlow)`--)
  (--`!p f.
      (a_read (s_atb param p f) ==>
       dom (p_level (s_ptb param p)) (i_level (s_itb param f))) /\
      (a_write (s_atb param p f) ==>
       dom (i_level (s_itb param f)) (p_level (s_ptb param p)))`--)


(--`(~(room_in_process_table param) \/ ~(space_for_new_process param))
    => T
    | (dom (p_level (s_ptb param (new_process param)))
        (p_level (s_ptb param rp)))`--)
==============================
  (--`!level. p_level (s_ptb param (dummy_process level)) = level`--)
  (--`!f.
      (s_ftb param f = empty_file_contents) ==>
      (i_level (s_itb param f) = systemlow)`--)
  (--`!p f.
      (a_read (s_atb param p f) ==>
       dom (p_level (s_ptb param p)) (i_level (s_itb param f))) /\
      (a_write (s_atb param p f) ==>
       dom (i_level (s_itb param f)) (p_level (s_ptb param p)))`--)


(--`(~(execute_okay rp rf param) \/
     ~(space_to_load rf param) \/
     ~(args_okay arglist rf))
    => T
    | ((dom (p_level (s_ptb param rp)) (i_level (s_itb param rf)))
       => T
       | ((~(room_in_process_table param) \/ ~(space_for_new_process param))
          => T
          | (dom (p_level (s_ptb param (new_process param)))
              (p_level (s_ptb param rp)))))'--)
============================
  (--'!level. p_level (s_ptb param (dummy_process level)) = level'--)
  (--'!f.
      (s_ftb param f = empty_file_contents) ==>
      (i_level (s_itb param f) = systemlow)'--)
  (--'!p f.
      (a_read (s_atb param p f) ==>
       dom (p_level (s_ptb param p)) (i_level (s_itb param f))) /\
      (a_write (s_atb param p f) ==>
       dom (i_level (s_itb param f)) (p_level (s_ptb param p)))'--)


val it = () : unit
- e(REWRITE_TAC
   [axiom "-" "new_process_new",
    definition "-" "empty_p",
    definition "-" "p_level",
    axiom "-" "systemhigh_high",
    theorem "romlemmas" "rom_condlemma1"]);
OK..

Goal proved.
|- (~(execute_okay rp rf param) \/
    ~(space_to_load rf param) \/
    ~(args_okay arglist rf))
   => T
   | ((dom (p_level (s_ptb param rp)) (i_level (s_itb param rf)))
      => T
      | ((~(room_in_process_table param) \/ ~(space_for_new_process param))
         => T
         | (dom (p_level (s_ptb param (new_process param)))
             (p_level (s_ptb param rp)))))

Remaining subgoals:
(--'(~(p_realuid (s_ptb param victim) = p_realuid (s_ptb param rp)) \/
     p_hanging (s_ptb param victim) \/
     p_ignore (s_ptb param victim) \/
     ~(dom (p_level (s_ptb param victim)) (p_level (s_ptb param rp))))
    => T
    | ((p_catch (s_ptb param victim))
       => (dom (p_level (s_ptb param victim)) (p_level (s_ptb param rp)))
       | T)'--)
```

```
===========================
  (--`!level. p_level (s_ptb param (dummy_process level)) = level`--)
  (--`!f.
      (s_ftb param f = empty_file_contents) ==>
      (i_level (s_itb param f) = systemlow)`--)
  (--`!p f.
      (a_read (s_atb param p f) ==>
       dom (p_level (s_ptb param p)) (i_level (s_itb param f))) /\
      (a_write (s_atb param p f) ==>
       dom (i_level (s_itb param f)) (p_level (s_ptb param p)))`--)


(--`(~(room_in_process_table param) \/ ~(space_for_new_process param))
    => T
    | (dom (p_level (s_ptb param (new_process param)))
        (p_level (s_ptb param rp)))`--)
===========================
  (--`!level. p_level (s_ptb param (dummy_process level)) = level`--)
  (--`!f.
      (s_ftb param f = empty_file_contents) ==>
      (i_level (s_itb param f) = systemlow)`--)
  (--`!p f.
      (a_read (s_atb param p f) ==>
       dom (p_level (s_ptb param p)) (i_level (s_itb param f))) /\
      (a_write (s_atb param p f) ==>
       dom (i_level (s_itb param f)) (p_level (s_ptb param p)))`--)


val it = () : unit
- e(REWRITE_TAC
   [axiom "-" "new_process_new",
    definition "-" "empty_p",
    definition "-" "p_level",
    axiom "-" "systemhigh_high",
    theorem "romlemmas" "rom_condlemma1"]);
OK..

Goal proved.
|- (~(room_in_process_table param) \/ ~(space_for_new_process param))
   => T
   | (dom (p_level (s_ptb param (new_process param)))
       (p_level (s_ptb param rp)))

Remaining subgoals:
```

```
(--`(~(p_realuid (s_ptb param victim) = p_realuid (s_ptb param rp)) \/
     p_hanging (s_ptb param victim) \/
     p_ignore (s_ptb param victim) \/
     ~(dom (p_level (s_ptb param victim)) (p_level (s_ptb param rp))))
    => T
    | ((p_catch (s_ptb param victim))
       => (dom (p_level (s_ptb param victim)) (p_level (s_ptb param rp)))
       | T)`--)
==============================
  (--`!level. p_level (s_ptb param (dummy_process level)) = level`--)
  (--`!f.
      (s_ftb param f = empty_file_contents) ==>
      (i_level (s_itb param f) = systemlow)`--)
  (--`!p f.
      (a_read (s_atb param p f) ==>
       dom (p_level (s_ptb param p)) (i_level (s_itb param f))) /\
      (a_write (s_atb param p f) ==>
       dom (i_level (s_itb param f)) (p_level (s_ptb param p)))`--)


val it = () : unit
- e(ASM_CASES_TAC
   (--`((dom:'Level -> 'Level -> bool)
          (p_level (s_ptb (param:^MinixState) (victim:'UserProcess)))
          (p_level (s_ptb param rp)))`--) THEN
  ASM_REWRITE_TAC[theorem "romlemmas" "rom_condlemma1"]);
OK..

Goal proved.
|- (~(p_realuid (s_ptb param victim) = p_realuid (s_ptb param rp)) \/
    p_hanging (s_ptb param victim) \/
    p_ignore (s_ptb param victim) \/
    ~(dom (p_level (s_ptb param victim)) (p_level (s_ptb param rp))))
   => T
   | ((p_catch (s_ptb param victim))
      => (dom (p_level (s_ptb param victim)) (p_level (s_ptb param rp)))
      | T)

Goal proved.
|- NoWritesDown dom (MinixOutLevel param) (p_level (s_ptb param rp))
     MinixInvocVal
     (INR Minix)
     (reqresponse param rp request)
```

```
Goal proved.
|- BPSP_nowritesdown dom MinixInLevel MinixOutLevel MinixInvariant
     MinixInvocVal
     Minix

Top goal proved.
val it = () : unit
- save_top_thm "Minix_nowritesdown";
val it =
  |- BPSP_nowritesdown dom MinixInLevel MinixOutLevel MinixInvariant
       MinixInvocVal
       Minix : thm
```

Chapter 4

Augmenting the Network Driver Model

4.1 Introduction

The report "A Secure Network Device Driver" [19] presents a method and code to handle the secure delivery of MLS information to and from a network at the device driver level. The basic idea is to add label information to packets before they go out to the network and to route them to a higher level protocol at the appropriate security level when they arrive.

Here we utilize the design of [19] and do not critique it or change it. Instead, we deal with how to improve modeling methods to more clearly surface potential security problems. In particular, we describe how to provide more detail about proper handling of memory. This model is aimed at achieving a more complete characterization of the MAC nondisclosure problem. However, this model could also be considered an integrity model, as the main mechanism discussed is how to achieve integrity of the security label field.

For further discussion of nondisclosure and integrity theories, see Volume II of the Romulus Documentation Set.

4.1.1 The Problem

In a network device driver, one needs to show that a message has been correctly labeled before being sent out to the network or "looped-back" to the same computer. The problem with using traditional Romulus-style modeling is that the models are usually too abstract to describe whether a message's label could be tampered with before a message is sent out. The traditional Romulus-style models are typically of the form:

```
Process P(x) =
  -- get an input message
  receive(e) then
    begin
      -- add the label
      send(label_message(e,x));
      P(x);
    end
```

One can prove that such a model is secure by showing that the output message is correctly labeled. However, in this style specification, the memory is local to the process, so the possibility that the label field could be altered before the message is sent is not exhibited. The model presented here describes how to characterize this limitation and related problems due to having shared memory.

4.1.2 Trust and Modeling Memory Protection

The network demonstration example was built on top of a standard insecure UNIX, so the assurance level of the supervisor software is too low for the overall system to be certified in the B1 class. However, it was assumed that the trusted part of the protocol could be given the appropriate protection. In this model we want to characterize the obligations that the underlying software should meet to provide this protection. With a sufficiently secure platform, one could then show the underlying system met these conditions.

In UNIX, the memory manager does not provide access control for trusted processes running in supervisor mode. Such processes have the power to alter any part of memory. However, we should check that the memory manager properly allocates and deallocates trusted memory. This check simplifies the problem of analyzing whether one trusted process may be interfering with another trusted process.

We note that trusted systems may provide finer access controls on memory than just supervisor and non-supervisor memory (e.g., LOCK [6]). This finer control would reduce the burden of what trusted software needs to be analyzed.

4.1.3 The Network Driver

In the network driver model, the trusted labeling procedure operates in privileged mode and is invoked by a trusted procedure (i.e., part of the network IP protocol). One of the parameters to the labeling procedure is a pointer to memory containing information that the trusted process wants to send over the network; this block of memory is called an *mbuf*. If there is sufficient room in the mbuf, then the labeler just adds a label to this mbuf. Otherwise, the labeler gets a new mbuf, adds the label to it, and then links it to the old mbuf. The labeler then calls the device driver.

We would like to exhibit any security problems caused by other processes altering the memory containing the label. We want to be sure that the label put on the data by the trusted procedure cannot be altered by an untrusted process. We also want to constrain how trusted processes can alter the label.

In the network driver example, the memory allocated for the information and label field is kernel memory, and so a process that is not in supervisor mode should be unable to alter the label. However, the higher levels of the protocol, as well as other parts of the system that operate in supervisor mode, could potentially change the label field. We would like to surface these points and more carefully describe the obligations of the trusted software.

4.1.4 General Description of the Solution

In order to describe how the solution is handled, we need a way of talking about how the memory is protected. However, we want to avoid putting in lower level details that might overly constrain a design.

We will construct models to handle potential memory problems by decomposing the problem into the principal procedure (in this case the labeler), the memory handler, other trusted processes, and other untrusted processes.

We need to describe the method by which memory protection is shared between the trusted processes. We will call the kind of method a *protocol* even though it is considerably simpler than a network communications protocol.

Methodology

We would like to know that all of the trusted components of the system follow their part of the protocol. However, for the purposes of modeling just the specific, principal procedure, we suggest the following steps:

- Define the memory-sharing protocol.
- Show that if each of the parties of the memory-sharing protocol abide by their part of the protocol, then the memory is properly protected.
- Add the memory sharing constraints to the requirements of the memory manager if it has not yet been built, or check that the memory-sharing protocol can be supported with the given memory manager.
- Specify the principal procedure (or process).
- Check the trusted procedure:
  - Check that the trusted procedure satisfies the desired security theory (e.g., restrictiveness) provided that memory protocol is being followed by other processes.
  - Check that the trusted process abides by its part of the memory protocol.
  - Check that the trusted process does not interfere with other trusted processes' memory (including unallocated memory).
- Make sure the appropriate design/code level constraints are added to other parts of the trusted software.

4.2 Informal Description

4.2.1 Informal Statement of the Protocol

The memory manager will not provide access controls for trusted processes. We would like to be able to protect memory from most trusted processes, as well as from untrusted processes. However, protection of one trusted process from another trusted process will not be centrally enforced on the given platform. Thus the protocol will extend to all trusted parts of the system.

The memory handler will not allow protected memory to be altered by an untrusted process until that memory has been explicitly released. Protected memory will be allocated and deallocated, so that if trusted processes do not write to memory that was allocated to another process, then that memory will be protected. This requirement amounts to making sure that the allocation and deallocation routines do not allow some memory location to be allocated at the same time. Provisions for more complex memory sharing among trusted processes are not needed in this model and are not discussed.

Trusted processes that use protected memory are responsible for properly labeling any data. In this model, the network labeler will properly label the data before invoking the device driver. In particular, the label must be set after the memory is protected and before the pointer to this memory is passed to the device driver. The network labeler will not change that label after it has been set, and it will not pass the pointer to another procedure or process. The time-dependent nature of these constraints is shown in the function decomposition diagrams of Figure 4.1 and Figure 4.2. The first picture describes the case where a new mbuf needs to be allocated, and the second picture describes the case where the label is added to an existing mbuf.

The device driver will not change the label field and the memory manager will not change the label except when the memory is released. The device driver will release the memory before returning to the labeler.

When the memory is released it will be zeroed. Exception and resource exhaustion responses will be sent on any failure to any "allocate protected memory" request. However, failures to release protected memory will not be analyzed in this model. (One could add extra requirements for availability.)

4.2.2 Informal Correctness Argument

The desired property is that the message goes to the network with the correct label.

By the definition of the labeler, all packets get labeled with the level of the process that is sending the packet. As shown in Figure 4.1, when the label is added, the memory is protected. The memory is not unprotected or released until after the message is sent to the network. Also, neither the

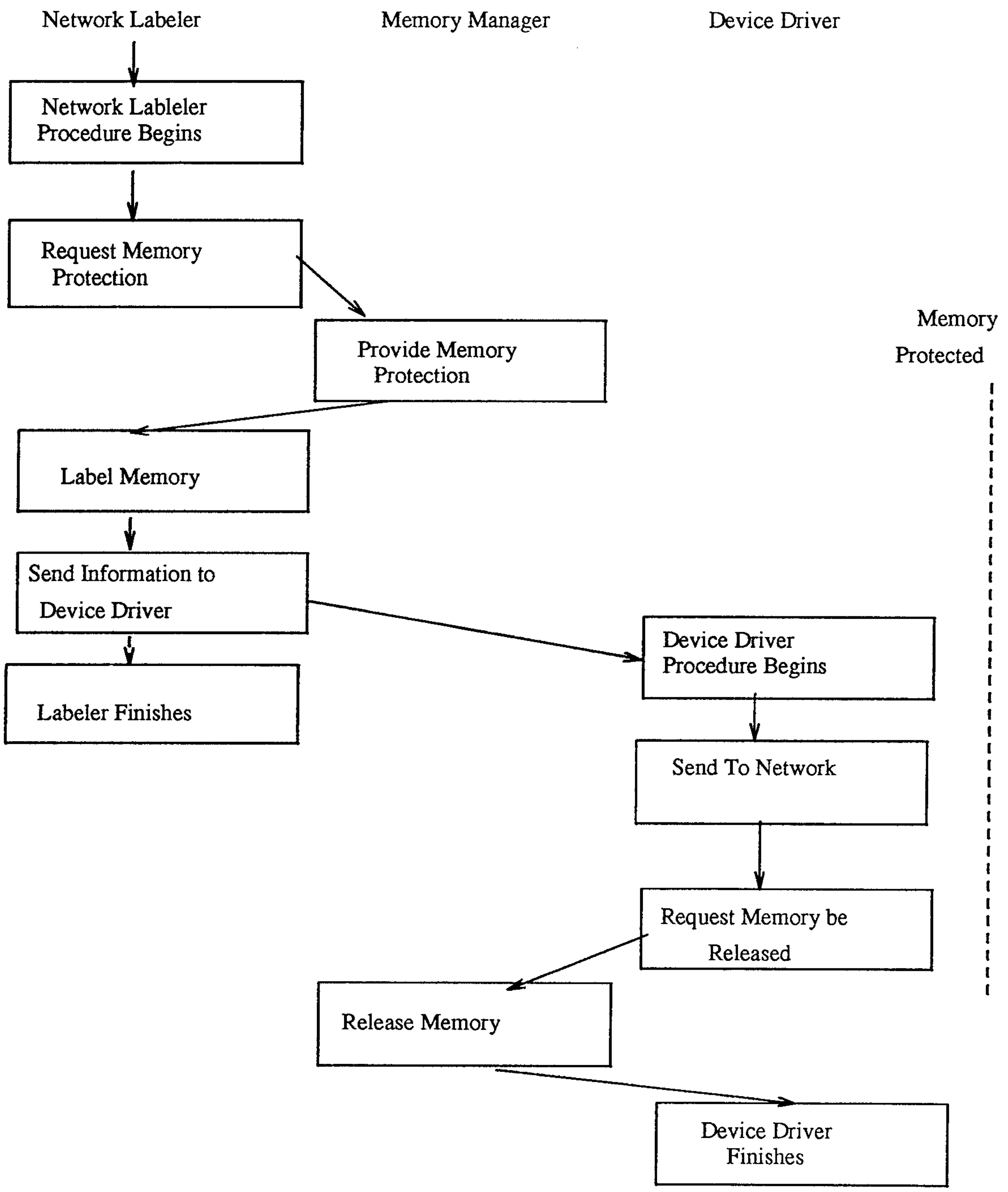
Network Labeler
Memory Manager
Device Driver
Network Lableler Procedure Begins
Request Memory Protection
Provide Memory Protection
Memory Protected
Label Memory
Send Information to Device Driver
Labeler Finishes
Device Driver Procedure Begins
Send To Network
Request Memory be Released
Release Memory
Device Driver Finishes

Figure 4.1: Memory Protection for New mbuf

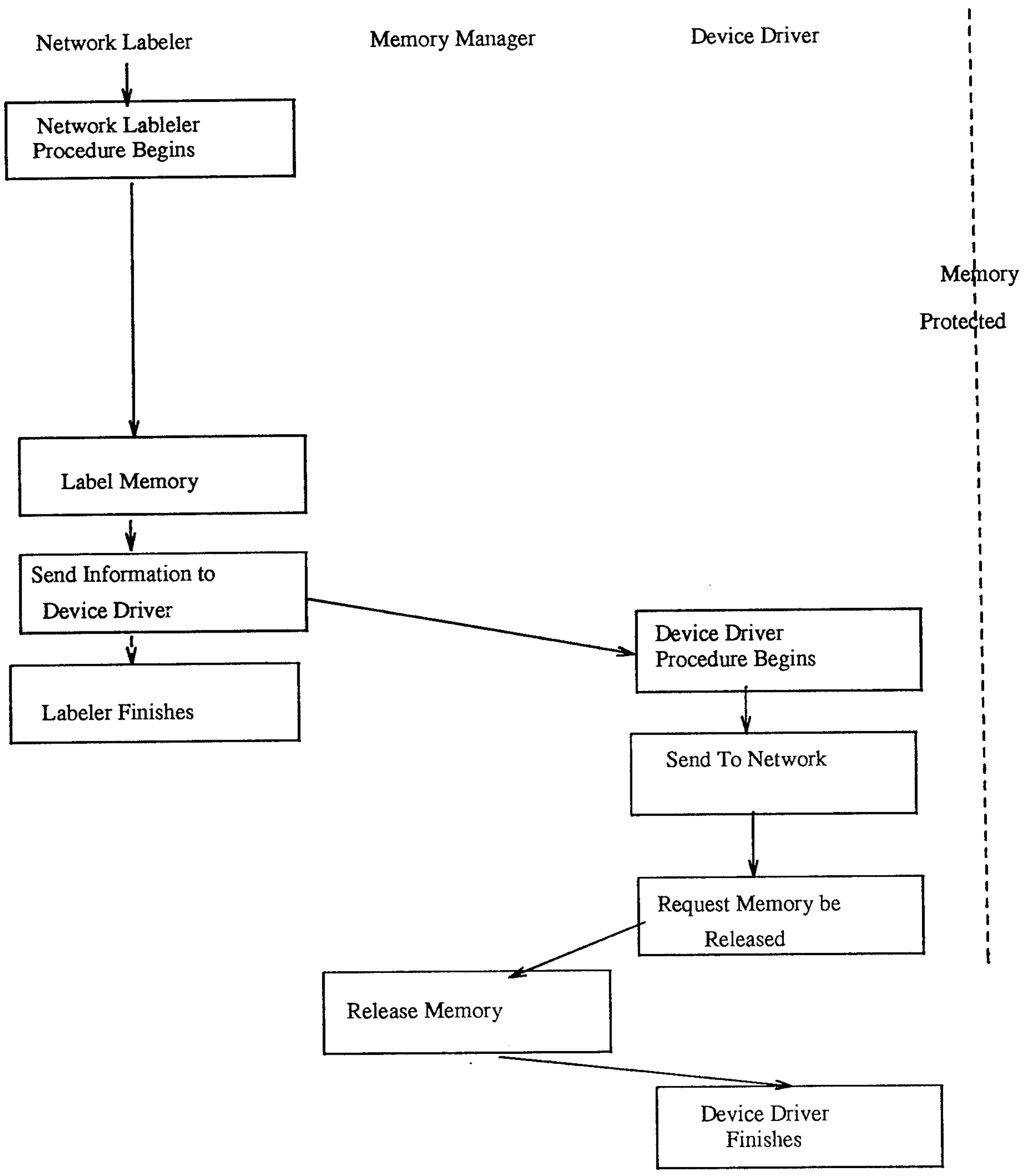

Figure 4.2: Memory Protection for Existing mbuf

secure labeler nor the device driver reset the label once it has been set and before it has been released.

So, hypothetically, the label will be correct when it is sent to the network. Unfortunately, in the real case, the memory manager will not really provide any reference mediation for trusted processes, only allocation and deallocation. Hence the other trusted processes must respect the allocation scheme.

4.2.3 Informal Description of Trusted Labeler

In this model, the trusted labeler is essentially just the labeler part of the protocol. Hence the description of the protocol for the labeler and the description of the labeler are essentially the same. In more complicated models, a separate description of the labeler would be useful.

In summary, the labeler gets protected memory (if needed), adds the correct label (using the level of the process), and sends the information to the driver. The labeler then returns.

4.2.4 Why the Trusted Labeler Meets Its Criteria

The potential information flow problems are that

- the wrong label is added, or
- information leaks through some internal variables, or
- there are failures in expected system support (e.g., mis-identification of the process level), or
- there may be timing channels (depending on process scheduling).

None of these potential problems are easily identifiable using only a high-level model. One should make these checks at the code level (or even at the machine level). However, in the model, these problems are partially addressed by the specification that the information going to the device driver is just the incoming mbuf with proper labeling added.

As noted above, the labeler functionality is just a part of the memory protocol. (For other kinds of models, showing that the procedure abides by the protocol may not be so trivial.)

No other protected memory is directly accessed by the labeler. Indirect accesses due to system procedure calls should not cause a problem by the assumption of the correctness of these calls. Note that this condition should be checked at the code level.

4.2.5 Constraints for Other Trusted Processes

Trusted processes must not write to protected memory that has been allocated solely for another process and has not yet been released. In particular, they should not write to mbufs allocated by the network IP protocol or the labeler that may contain the security label.

In the case that a new mbuf is not needed, we are assuming that IP properly allocated the packet (i.e., that it is already "protected" from other processes). This extra constraint is on IP.

4.3 Formal Version

It is typically beneficial to formalize protocols because the formalization may reveal some subtle error. In this particular case, the formalization is of more limited value because the example is simple. However, we will provide formalizations of the properties that should be satisfied and a specification of the top-level model. We will make this presentation in a variant of the Romulus SL specification language [18]. A mapping into a particular HOL theory should be straightforward.

For this model, we will need to keep track of the "relevant" events from the various processes and procedures, as well as the content of the allocatable memory. A trace `s` is a sequence of the relevant events. The expression `s(n)` means the nth element of the trace `s`.

The relevant events are internal events between different trusted processes:

- Procedure calls (or their equivalent) on the labeler, memory handler and the device driver from the process.

- Other requests to the memory handler from other untrusted and trusted processes.

- Return values for each of the invocations described above.

- Outputs of the device driver to the network.

We will represent the state of the protected memory by the variable `state`. The state together with the parameters of the procedure are the objects being protected. No local variables are explicitly surfaced in this model.

We will describe a number of predicates and functions for the memory protection. One of the more common arguments is `pid`, which means *process identifier*. Note that, in this example, all of the protocol layers (IP, labeler, and the device driver) have the same *pid*.

4.3.1 Security Properties

Events:

- `make_request(pid,size,initial)` – request for protected memory by a process with process id `pid` of size `size` with initial value pointed to by `initial`.

- `write_mem(pid,address,value)` – request by `pid` to set the `address` to `value`.

- `read_mem(pid,address)` – request by `pid` to read the value at `address`. (This operation is not used in this model.)

- `free_mem(user,address)` – request to free protected memory.

- `return_result(user,value)` – return value from call.

A `return_result` can happen only in response to some request.

We will define `corresponding_to(s,return_result(pid,value))`to be the corresponding request to the `return_result` in trace `s`. First we introduce the auxiliary definition `corresponding_to_n`.

```
corresponding_to_n(s,n,return(pid,value))=
  if is_request(s(n)) and theprocess(s(n))=pid
    then s(n)
  else if n>0
```

```
    corresponding_to_n(s,n-1,return(pid,value))
  else  /* not possible */
    null_event

corresponding_to(s,return(pid,value))=
  corresponding_to_n(s,length(s),return(pid,value))
```

We want to keep track of the protected memory at stage n. Next we introduce the "virtual variable" request_item to describe an item that has been allocated in response to a protection request.

The request_item(user,size,initial,memptr) variable represents the information consisting of the user that requested the memory, the size of the request, the initial value, and a pointer to the memory that is allocated.

The expression inrange(addr,reqitem) indicates when the address addr is in the range of reqitem.

The function Free removes an item from a list.

```
Free(pid,address,l)=
  if l=empty_list then l
  else if inrange(address,first(l)) and (first(l).pid=pid)
    then tail(l)
  else concat(first(l),Free(pid,address,l))
```

The predicate Protected indicates which memory is currently supposed to be protected.

```
Protected(s,n)=
  if n=0 then empty_list
  else if s(n)=return(pid,value) then
    let req= corresponding_to(s(n)) in
      if req ~= null_event then
        concat(request_item(pid,req.size,req.initial,value),
               Protected(s,n-1))
      else
        Protected(s,n-1)
  else if s(n)=freemem(pid,address) then
    Free(pid,address,Protected(s,n))
  else
    Protected(s,n-1)
```

We need to be able to tell which trusted processes should have access to which memory. In this model, only the process that allocated the memory should be able to write it. (The model does not contain any reads.)

```
notpermited(a,b)=  (a~=b)
```

We would like to require that protected memory is actually protected by the memory manager. Recall that we will use the expression `state` to refer to the protectable memory.

```
ProperlyProtected(s)=
  forall n
    if s(n) = set_mem(pid,address) and
      forsome reqitem,  (reqitem in Protected(s,n) and
        notpermited(reqitem.pid,pid) and
        inrange(address,reqitem) )
    then
      state(s,n+1)=state(s,n)
```

However, as the memory manager cannot enforce this property, we use an alternative: the memory manager should not allocate memory with overlapping address spaces.

```
ProperAllocation=
  forall s,n,pid,size,initial,value
  If s(n)=return_result(pid,value) and
     make_request(pid,size,initial)=corresponding_to(s,s(n))
  then
     inrange(address,requestitem(pid,size,initial,value) implies
       forall reqitem in Protected(s,n)
         not inrange(address,reqitem);
```

Trusted processes do not make improper requests. A process can become trusted depending upon what code it is executing. For simplicity we assume that a *pid* is either trusted or untrusted (we logically can append an extra attribute to the id to make this distinction). Note that this condition also applies to the labeler.

```
If s(n)=write_mem(pid,address,value) and
  iscurrentlytrusted(pid)
then foarll
    reqitem in Protected(s,n)
    if inrange(address,reqitem) then
      permitted(pid,reqitem.pid)
```

Other properties of the labeler are restrictiveness and proper labeling. These properties can be checked by a meta-level analysis of the specification, see section 4.3.3.

4.3.2 Specification

The labeler is specified as follows:

```
define get_protected(pid,labelsize)=
  make_request(pid,labelsize,nullvalue)

define setmem(ptr,labelfield,level)=
  write_mem(pid,ataddress(ptr,labelfield),level)

Kernel Procedure Labeler(membuf_ptr)=
  -- System Privileges are granted when this function is invoked
  -- Get the level of the process which invoked this process

  let level=levelprocess() in
    -- Get protected memory to add the label field
    if needspace(membuf_ptr) then
      let ptr=get_protected(getpid(),labelsize) in
        begin

        -- Handle the case that the procedure call failed
        if protection_failed(ptr)
          return;
        else
          begin
            -- Set the label field of the protected memory
            setmem(ptr,ptrfield,membuf_ptr);
            setmem(ptr,labelfield,level);

            -- Pass the information to the device driver
            -- the device_driver should release the memory
            call device_driver(ptr);
            -- Previous Privileges are restored when this
            --             function returns
            return;
          end
        end
    else
      begin
        setmem(mbuf_ptr,labelfield,level);
        call device_driver(mbuf_ptr);
        return;
      end
```

4.3.3 Formal Correctness Arguments for the Trusted Labeler

To check restrictiveness, we could formalize the labeler specification into HOL and apply security condition generation methods. However, as noted in the informal section, because of the simplicity of this model it would not provide much added insight. It might be worthwhile to use some code level verification techniques.

To show that the labeler abides by the protocol, one could convert the state machine specification into a trace specification and then check that the traces satisfied the correct labeling properties. Another alternative could be to construct a mechanism like SCG_TAC to check that labels were set at and only at the appropriate places.

One could also define the level of the output message to be based on the label field of the message. The labeling property would then essentially be met by restrictiveness. (This method is actually weaker then specified in the protocol, in that the label could be set and reset before it was sent out. However, in fact this weakening would be sufficient.)

On the other hand, as mentioned above, the model is sufficiently simple that a modeling check is not necessary. One should be careful to check that the labeling really is handled correctly at the code level.

To show that the labeler does not interfere with other trusted processes, one could examine the traces of the model to see that no other *setmem* requests to other processes' memory were issued. But this examination is probably not worth the effort at the modeling level. Again one should check this property at the code level. Note that there is a potential problem if the *setmem* for the label field overwrites part of the pointer to the next mbuf. However, the packet size should be sufficiently large that this problem will not happen.

4.3.4 Constraints for Other Trusted Processes

Any formal mechanisms for handling trusted process constraints are better done at the low-level design or coding stages.

Chapter 5

Authentication Protocol Models

5.1 Introduction

In this chapter we present an analysis of two authentication protocols. Authentication protocols are important contributors to integrity assurance because they are used to establish the correct identity of processes. Furthermore, authentication protocols are the basis of secure key distribution for encryption, and encryption is the method used to protect the integrity of data in transmission along a path that is not trusted to be private. Encryption is also an extremely powerful means of detecting transmission errors of non-malicious origin — random noise, electrical faults, etc. — and is thus used to fill one role of a checksum. Thus, the correctness of a protocol is also vital to secure data transmission.

An authentication protocol is an exchange of messages between a number of processes, called "principals". A typical aim of a protocol is to establish an encryption key shared by two principals. The message exchange often involves a third party — a "keyserver"— who is trusted to generate good keys and keep secrets. Protocol messages employ a variety of techniques, to ensure, for example, the identity of a principal, that the messages have been recently generated, and that the keys exchanged are protected. For more background on authentication protocols see Volume II and Volume IV of the Romulus Documentation Set.

We deal here with protocols modeled at the level of the messages between the processes, or "principals" involved, as is standard in this area. We use the Romulus implementation of authentication logic to state requirements on particular protocols and prove that they are correct, or in the case of an inadequate protocol, examine what the protocol *can* establish.

Full background and exposition of the logic of authentication used here can be found in Volume II of the Romulus Documentation Set. In the examples that follow, it is assumed that the reader is familiar with that material.

We will in following sections treat two well-known protocols as examples. For each, we present the description of the protocol and its specification, and we describe the proof of correctness. As mentioned above, we assume that the reader has familiarized himself with the logic, though looking at what follows alone should give the casual reader a rough idea of the method.

5.2 The Denning-Sacco Protocol

In this section, we specify and prove some aspects of the security of the Denning-Sacco authentication protocol [7].

5.2.1 Protocol Description and Specification

First, we load the theory of cryptographic protocols and declare the basic objects used in this particular protocol.

```
new_constant{Name = "A",
             Ty = ==`:principal`==}; % Wants new key between A and B %
new_constant{Name = "B",
             Ty = ==`:principal`==};
new_constant{Name = "Svr",
             Ty = ==`:principal`==}; % Keyserver %
new_constant{Name = "Ts",
             Ty = ==`:^textlist`==}; % Timestamp %
new_constant{Name = "Kas",
             Ty = ==`:^textlist`==}; % Existing key between A and S %
new_constant{Name = "Kbs",
             Ty = ==`:^textlist`==}; % Existing key between B and S %
new_constant{Name = "Kab",
             Ty = ==`:^textlist`==}; % New key between A and B %
```

Now we describe the actual protocol by defining the messages, together with whom they are to and from.

```
new_open_axiom("dsm1", --`send A Svr ((name A) APP (name B))`--);
new_open_axiom("dsm2", --`send Svr A (encrypt Kas ((name B) APP Kab APP
          Ts APP (encrypt Kbs ((name A) APP Kab APP Ts))))`--);
new_open_axiom("dsm3", --`send A B (encrypt Kbs ((name A) APP
          Kab APP Ts))`--);
```

It is assumed that A and B each have a key with which they can communicate securely with the server, Svr. A wishes to establish a new key to share with B, so that they may have a session using encryption. A sends a message, consisting of his and B's names, in the clear, to the server, the intent of which is understood by the server. The server sends the new key, Kab to A in a "certificate", containing a timestamp Ts. The timestamp ensures A that this message is fresh. The server signs the certificate by encrypting it with Kas, which is shared by it and A. This encryption also ensures that no one can read or tamper with the contents. The server avoids having to communicate with B by passing this responsibility to A: in his message to A, he includes an analogous certificate for B. In the third message, A passes this certificate on to B.

This protocol is efficient and simple, and it is commonly used. The use of timestamps is adequate to ensure freshness if process clocks are tightly synchronized. The protocol does not have a "handshake" at the end so that each of A and B can be sure that the other has successfully received the key.

We now give the specification. First we give the initial assumptions, or preconditions:

```
new_open_axiom("dsa1", --`theorem(believes A
                 (is_shared_secret A Svr Kas))`--);
new_open_axiom("dsa2", --`theorem(believes A (is_fresh Ts))`--);
new_open_axiom("dsa3", --`theorem(believes A (is_recog (name B)))`--);
new_open_axiom("dsa4", --`theorem(possesses A Kas)`--);
new_open_axiom("dsa5", --`theorem(possesses A (name A))`--);
new_open_axiom("dsa6", --`theorem(possesses A (name B))`--);

new_open_axiom("dsb1", --`theorem(believes B
                 (is_shared_secret B Svr Kbs))`--);
new_open_axiom("dsb2", --`theorem(believes B (is_fresh Ts))`--);
new_open_axiom("dsb3", --`theorem(believes B (is_recog (name A)))`--);
new_open_axiom("dsb4", --`theorem(possesses B Kbs)`--);

new_open_axiom("dss1", --`theorem(possesses Svr Kas)`--);
new_open_axiom("dss2", --`theorem(possesses Svr Kbs)`--);
new_open_axiom("dss3", --`theorem(possesses Svr Kab)`--);
new_open_axiom("dss4", --`theorem(possesses Svr Ts)`--);
new_open_axiom("dss5", --`theorem(possesses Svr (name A))`--);
new_open_axiom("dss6", --`theorem(possesses Svr (name B))`--);
new_open_axiom("dss7", --`theorem(believes Svr
                  (is_shared_secret Svr B Kbs))`--);
new_open_axiom("dss8", --`theorem(believes Svr
                  (is_shared_secret Svr A Kas))`--);
```

The logic we use is richer than the original BAN logic of protocols (see Volume II of the Romulus Documentation Set) and contains the machinery of extensions to enable us to model the intentions of a principal when it sends a message. This feature is used in the next example, but not in this one. Here, we show how to ignore this feature and prove simple properties without the need to discuss extensions. To this end, we set all extensions to be the most uninformative statement `nil`.

```
new_open_axiom("all_extensions_nil", --`!x. extension x = nil`--);
```

We now define some facts about the final state, the postcondition, which we wish to hold after execution.

```
new_definition ("postcond", --`postcondition =
     theorem(possesses A Kab) /\
     theorem(believes A (convey Svr ((name B) APP Kab APP Ts))) /\
     theorem(believes A (is_fresh ((name B) APP Kab APP Ts))) /\
     theorem(possesses B Kab) /\
     theorem(believes B (convey Svr ((name A) APP Kab APP Ts))) /\
     theorem(believes B (is_fresh ((name A) APP Kab APP Ts)))`--);
```

To demonstrate the possibilities for specifications, we have chosen to prove these facts about the final state: `A` and `B` possess the session key; it was the server who conveyed the important plaintext of the certificate to each; and this plaintext was fresh.

We note here that this specification is not the full statement of what is normally desired from the Denning-Sacco protocol. In addition to ensuring that `A` and `B` possess the session key as raw data, as stated here, we would also want them to believe that it constituted a good key, probably without even caring who conveyed it. For the full typical requirements of a protocol, see the Needham-Schroeder model in the next example.

5.2.2 Proof that the Protocol Meets Its Specification

The proof is more or less straightforward. We split the conjuncts off the postcondition and attack them one by one. Consider the first one,

```
theorem(possesses A Kab).
```

Looking at the rules governing possession (i.e., axioms of the authentication logic), we see that the rules

```
new_open_axiom("P3", --`!p x y. theorem(possesses p(x APP y))
                                ==> theorem(possesses p x)`--);
new_open_axiom("P4", --`!p x y. theorem(possesses p(x APP y))
                                ==> theorem(possesses p y)`--);
new_open_axiom("P5", --`!p x k. theorem(possesses p x) /\
                       theorem(possesses p k) ==>
                       theorem(possesses p (encrypt k x))`--);
```

apply. This way, with appropriate instantiations of the universally quantified variables, we reduce our goal to two subgoals. One is an obligation to show that A possesses the whole message sent to it by Svr. The other is to show that A possesses the key Kas that A uses to communicate with the keyserver. The second of these subgoals is an initial assumption and is proved quickly.

For the first subgoal, we see that the rules

```
new_open_axiom("P1", --`!p x. theorem(receive p x) ==>
                              theorem(possesses p x)`--);
new_open_axiom("R1", --`!p q x. send p q x /\ theorem (elig p x)
                                ==> theorem (receive q x)`--);
```

apply. The first says that to possess something, it is sufficient to receive it. The second says that receipt can come about if someone sent it (which occurred, by the protocol message description axioms) and moreover, *if they were eligible to send it.* This last condition of eligibility is our most important extension to the authentication logic GNY. It rules out certain impossible protocols that otherwise can be proved correct. This construct is adumbrated in Gong's paper [8].

Certain facts, which we call "lemmas", that are needed more than once in the proofs of the various subgoals, are proved separately and saved as HOL theorems, to be used when needed. For example, it was apparent that in both of the proofs of the first two conjuncts of the postcondition, we needed the fact that `Svr` was eligible to send its message to `A`. This lemma is the first thing we see in the proof transcript in Appendix 5.A.

Most of the proof is in the style described above — judiciously matching the current goal to the consequent of one of the implicative axioms. Continuing in this fashion, it is mostly straightforward to reduce our goals to known facts.

For completeness, we include the full proof transcript in Appendix 5.A. We remark that no attempt has been made to compress the proof in that appendix using the more powerful HOL tactics. Our aim in these first applications of the tool has been to understand the way the lower level proof works, as we intend to work on the design and automation of the whole verification process in a later version.

5.3 The Needham-Schroeder Protocol

In this section, we specify and prove the security of the Needham-Schroeder authentication protocol. The Needham-Schroeder protocol is a very interesting case study, as it is an important protocol, which influenced many later protocols. It also has a serious flaw and other lesser inadequacies. A logical analysis uncovers these flaws and indicates how the protocol needs to be fixed.

5.3.1 Protocol Description and Specification

First, we load the theory of cryptographic protocols and declare the basic objects used in this particular protocol.

```
new_constant{Name = "A",
             Ty = ==`:principal`==}; % Wants new key between A and B %
new_constant{Name = "B",
             Ty = ==`:principal`==};
```

```
new_constant{Name = "Svr",
             Ty = ==':principal'==}; % Keyserver %
new_constant{Name = "Na",
             Ty = ==':^textlist'==}; % Nonce generated by A %
new_constant{Name = "Nb",
             Ty = ==':^textlist'==}; % Nonce generated by B %
new_constant{Name = "Kas",
             Ty = ==':^textlist'==}; % Existing key between A and S %
new_constant{Name = "Kbs",
             Ty = ==':^textlist'==}; % Existing key between B and S %
new_constant{Name = "Kab",
             Ty = ==':^textlist'==}; % New key between A and B %
```

Now we describe the actual protocol by defining the messages, together with whom they are to and from.

```
new_open_axiom("nsm1", --'send A Svr ((name A) APP (name B) APP Na)'--);
new_open_axiom("nsm2", --'send Svr A (encrypt Kas (Na APP (name B)
                  APP Kab APP (encrypt Kbs (Kab APP (name A)))))'--);
new_open_axiom("nsm3", --'send A B  (encrypt Kbs (Kab APP (name A)))'--);
new_open_axiom("nsm4", --'send B A (encrypt Kab Nb)'--);
new_open_axiom("nsm5", --'send A B (encrypt Kab (feas Nb))'--);
```

It is assumed that A and B each have a key with which they can communicate securely with the server, Svr. A wishes to establish a new key to share with B, so that they may engage in a session using encryption. A sends a message, consisting of his and B's names, and a nonce Na (a fresh random number) in the clear, to the server, the intent of which is understood by the server. The server sends the new key Kab to A in a "certificate", containing A's nonce Na. This nonce ensures A that this message is fresh. The server signs the certificate by encrypting it with Kas, which is shared by it and A. This encryption also ensures that no one can read or tamper with the contents. The server avoids having to communicate with B by passing this responsibility to A: in his message to A, he includes a certificate for B. This certificate is not fully analogous to that sent for A: there is no nonce in it. We will return to this point during the discussion of the proof below. In the third message, A passes this certificate on to B.

By now A and B possess the new key. The final two messages are an attempt at a "handshake", whereby each would like to establish for himself that the other possesses the key, and B would also like to determine that the key is fresh. The function `feas` is typically "subtract one" and is a standard way for a principal to prove that he possesses the argument.

As mentioned, this protocol has a number of flaws. When amended, it is a useful protocol that incorporates a handshake so that each of A and B can be sure that the other has successfully received the key.

We now give the specification. First we give the initial assumptions, or preconditions:

```
new_open_axiom("nsa1", --`theorem(believes A
                  (is_shared_secret A Svr Kas))`--);
new_open_axiom("nsa2", --`theorem(believes A (is_fresh Na))`--);
new_open_axiom("nsa3", --`theorem(believes A (is_recog (name B)))`--);
new_open_axiom("nsa4", --`theorem(possesses A Na)`--);
new_open_axiom("nsa5", --`theorem(possesses A Kas)`--);
new_open_axiom("nsa6", --`theorem(possesses A (name A))`--);
new_open_axiom("nsa7", --`theorem(possesses A (name B))`--);
new_open_axiom("nsa8", --`theorem(believes A (juris Svr
                  (is_shared_secret A B k)))`--);
new_open_axiom("nsa9", --`theorem(believes A (juris_star Svr))`--);
new_open_axiom("nsa10", --`theorem(believes A (juris_star B))`--);

new_open_axiom("nsb1", --`theorem(believes B
                  (is_shared_secret B Svr Kbs))`--);
new_open_axiom("nsb2", --`theorem(believes B (is_fresh Nb))`--);
new_open_axiom("nsb3", --`theorem(believes B (is_recog Nb))`--);
new_open_axiom("nsb4", --`theorem(believes B (is_recog (name A)))`--);
new_open_axiom("nsb5", --`theorem(possesses B Nb)`--);
new_open_axiom("nsb6", --`theorem(possesses B Kbs)`--);
new_open_axiom("nsb7", --`theorem(believes B (juris Svr
                                       (is_shared_secret A B k)))`--);
new_open_axiom("nsb8", --`theorem(believes B (juris_star Svr))`--);
new_open_axiom("nsb9", --`theorem(believes B (juris_star A))`--);

% The following assumption is needed owing to a flaw in the protocol%

new_open_axiom("nsb10_dubious_assumption",
         --`theorem(believes B (is_fresh Kab))`--);

new_open_axiom("nss1", --`theorem(possesses Svr Kas)`--);
new_open_axiom("nss2", --`theorem(possesses Svr Kbs)`--);
new_open_axiom("nss3", --`theorem(possesses Svr Kab)`--);
new_open_axiom("nss6", --`theorem(believes Svr
                  (is_shared_secret Svr B Kbs))`--);
```

```
new_open_axiom("nss7", --`theorem(believes Svr
                   (is_shared_secret Svr A Kas))`--);
new_open_axiom("nss8", --`theorem(believes Svr
                   (is_shared_secret A B Kab))`--);
```

In the treatment of this protocol (unlike in the previous example), we use an extension construct of the logic, which is needed to establish the postcondition.

```
new_open_axiom("nse1", --`extension (encrypt Kbs (Kab APP (name A))) =
                            believes Svr (is_shared_secret A B Kab)`--);
new_open_axiom("nse2", --`extension (encrypt Kas (Na APP (name B) APP
                   Kab APP (encrypt Kbs (Kab APP (name A))))) =
                   believes Svr (is_shared_secret A B Kab)`--);
new_open_axiom("nse3", --`extension (feas Nb) =
                            believes Svr (is_shared_secret A B Kab)`--);
new_open_axiom("nse4", --`extension
            (Na APP ((name B) APP (Kab APP
                  (encrypt Kbs(Kab APP (name A)))))) = nil`--);
```

We now define the postcondition that we want to hold after the completion of the protocol.

```
new_definition ("postcond", --`postcondition =
     theorem(possesses A Kab) /\
     theorem(believes A (is_shared_secret A B Kab)) /\
     theorem(possesses B Kab) /\
     theorem(believes B (is_shared_secret A B Kab))`--);
```

This postcondition states that A and B end up with a session key as desired. There are two parts. The principals need to possess the raw data — the bits that comprise the key. But before using this data as a key, they need also to believe that this data can be trusted as a key.

Owing to several flaws in the protocol, the handshake at the end of the protocol is not adequate, and we do not state and attempt to prove the intended further consequences. Our logic does enable us to explain fully these shortcomings, though here we confine ourselves to investigating the problems that the protocol has in meeting the basic specification given just above.

5.3.2 Proof that the Protocol Meets Its Specification

The proof style is more or less straightforward, as in the previous case for Denning-Sacco. We proved the four conjuncts separately as lemmas, and at the end, we proved the postcondition from the lemmas. One part of the proof warrants close examination. In the proof of the fourth conjunct,

```
theorem(believes B (is_shared_secret A B Kab))‘--);
```

we must show that B attains this belief by trusting the server. To show this, we use the rules

```
new_open_axiom("J1", --‘!p q s.
          theorem(believes p (juris q s))    /\
          theorem(believes p (believes q s)) ==>
          theorem(believes p s)‘--);

new_open_axiom("J2", --‘!p q x s.
               theorem(believes p (juris_star q))   /\
               theorem(believes p (convey q x))     /\
               theorem(believes p (is_fresh x))     /\
               (extension x = s)                        ==>
               theorem(believes p (believes q s))‘--);

new_open_axiom("J3", --‘!p q s.
            theorem(believes p (juris_star q))              /\
            theorem(believes p (believes q (believes q s))) ==>
            theorem(believes p (believes q s))‘--);
```

The first of these rules says that for p to acquire a belief s from q, he must trust q and believe that q himself believes s. To establish that q himself believes s, p needs to see a fresh message from q whose extension is that belief. The last of these rules is a technicality, to deal with situations like we have here — we need to go from the fact that B believes the extension

```
believes Svr (is_shared_secret A B Kab)
```

to that B believes

```
is_shared_secret A B Kab.
```

We see then that there is a freshness requirement on the information B sees passed to him by A, originating from Svr. Indeed in the proof as performed, the following subgoal arose:

```
theorem(believes B(is_fresh(Kab APP (name A)))).
```

It was impossible to proceed from there, as the text contained no nonce known to B to be fresh.

This problem is now well known; it was discovered some time after the protocol was published. Interestingly, we had forgotten about it until we came up against it in the attempted proof. Rather than leave the proof in this state, we can better understand the protocol by adding the following unjustified assumption to the initial assumptions of the protocol.

```
new_open_axiom("nsb10_dubious_assumption",
          --`theorem(believes B (is_fresh Kab))`--);
```

Given this assumption, the proof continues, and the postconditions are all proved. Thus we can say that provided B makes the extra assumption that the new session key it gets sent from the server is fresh, the protocol achieves its goal. This assumption is a dangerous one, as the session key Kab is for an encryption method (e.g., DES), for which it is assumed feasible for an adversary to discover a key, given sufficient time. An adversary may have determined the old key, and while unable to construct message 3 himself, replayed the old message 3, spoofing A and thereby inducing B to use the compromised key. Other dangerous consequences of this scenario are described in [7, 2].

This weakness was corrected in a modified version of the protocol published several years later. The solution is of course for B to send a nonce to be included in the message to him.

The full HOL proof script of this protocol is given in Appendix 5.B. Note again that we have not compressed the proof by using the more powerful HOL tactics.

5.A Appendix: Proof Transcript for the Denning-Sacco Protocol

```
new_theory("ds_proof_90");

new_parent "ds_90";

(*---------------------Lemma-------------------------------*)

set_goal([], --`theorem
 (elig
  Svr
  (encrypt
   Kas
   (((name B) APP Kab) APP
     (Ts APP (encrypt Kbs((name A) APP (Kab APP Ts)))))))`--);

e(MATCH_MP_TAC (theorem "crypto_90" "elig_encr")  );

e(EXISTS_TAC (--`A`--));

e(CONJ_TAC);

e(MATCH_MP_TAC (axiom "crypto_90" "E3"));

e(CONJ_TAC);

e(MATCH_MP_TAC (axiom "crypto_90" "E3"));

e(CONJ_TAC);

e(MATCH_MP_TAC (axiom "crypto_90" "E1"));

e(ASSUME_TAC (axiom "ds_90" "dss6"));

e(PURE_ASM_REWRITE_TAC[]);

(*Another historic moment*)

e(MATCH_MP_TAC (axiom "crypto_90" "E1"));

e(ASSUME_TAC (axiom "ds_90" "dss3"));
```

```
e(PURE_ASM_REWRITE_TAC[]);

(*History repeats itself*)

e(MATCH_MP_TAC (axiom "crypto_90" "E3"));

e CONJ_TAC;

e(MATCH_MP_TAC (axiom "crypto_90" "E1"));

e(ASSUME_TAC (axiom "ds_90" "dss4"));

e(PURE_ASM_REWRITE_TAC[]);

(*History repeats itself*)

e(MATCH_MP_TAC (theorem "crypto_90" "elig_encr"));

e(EXISTS_TAC (--`B`--));

e(REPEAT CONJ_TAC);

e(MATCH_MP_TAC (axiom "crypto_90" "E1"));

e(MATCH_MP_TAC (axiom "crypto_90" "P2"));

e CONJ_TAC;

e(ASSUME_TAC (axiom "ds_90" "dss5") THEN PURE_ASM_REWRITE_TAC []);

(*subgoal proved*)

e(MATCH_MP_TAC (axiom "crypto_90" "P2"));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ds_90")));

(*subgoal proved; using list of all ds theorems seen as useful*)


e(REWRITE_TAC (map snd (axioms "ds_90")));

e(REWRITE_TAC (map snd (axioms "ds_90")));
```

```
e(REWRITE_TAC (map snd (axioms "ds_90")));

e(REWRITE_TAC (map snd (axioms "ds_90")));

e(REWRITE_TAC (map snd (axioms "crypto_90")));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ds_90")));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ds_90")));

(*subgoal proved*)

e(REWRITE_TAC (map snd (axioms "ds_90")));

e(REWRITE_TAC (map snd (axioms "crypto_90")));

(*end subproof*)

save_top_thm "Lemma_Svr_elig";

(*------------------Lemma---------------------------------------*)

set_goal([], --`theorem
   (possesses
    A
    (((name B) APP Kab) APP
     (Ts APP (encrypt Kbs((name A) APP (Kab APP Ts))))))`--);


e(ONCE_ASM_REWRITE_TAC[
  SYM (SPEC_ALL (SPEC (--`Kas`--)
     (GENL[--`k:^textlist`--,--`x:^textlist`--]
         (SPEC_ALL(axiom "crypto_90" "Y1")))))]);

e(MATCH_MP_TAC (axiom "crypto_90" "P7"));

e(CONJ_TAC);

rotate 1;
```

```
e(ASM_REWRITE_TAC [axiom "ds_90" "dsa4"]);

(* Historic moment:  successful proof of a subgoal*)

e(MATCH_MP_TAC (axiom "crypto_90" "P1"));

e(MATCH_MP_TAC (axiom "crypto_90" "R1"));

e(EXISTS_TAC (--`Svr`--));

e(CONJ_TAC);

e(PURE_REWRITE_TAC [definition "crypto_90" "APP_def"]);

e(ASSUME_TAC (axiom "ds_90" "dsm2"));

e(RULE_ASSUM_TAC
     (PURE_REWRITE_RULE [definition "crypto_90" "APP_def"]));

e(PURE_REWRITE_TAC [theorem "list" "APPEND_ASSOC"]);

e(RULE_ASSUM_TAC (PURE_REWRITE_RULE [theorem "list" "APPEND_ASSOC"]));

e(PURE_ASM_REWRITE_TAC[]);

e(REWRITE_TAC (map snd (theorems "ds_proof_90")));

(*Assumes that the lemmas are in theory ds_proof_90*)

save_top_thm "Lemma_A_poss";


(*-----------Finally, the proof of ds postcondition---------------*)

set_goal([], --`postcondition`--);

e(PURE_ONCE_REWRITE_TAC [definition "ds_90" "postcond"]);

e(CONJ_TAC);

e(MATCH_MP_TAC (axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`(name B)`--));
```

```
e(MATCH_MP_TAC (axiom "crypto_90" "P3"));

e(EXISTS_TAC (--`Ts APP (encrypt Kbs ((name A) APP Kab APP Ts))`--));

e(REWRITE_TAC (map snd (theorems "ds_proof_90")));

(*The last dealt with both the poss and elig bits using both the
     lemmas*)

e CONJ_TAC; (*Begin proof of second postcond conjunct: bel A conv ...*)

e(MATCH_MP_TAC (axiom "crypto_90" "M5"));

e(EXISTS_TAC (--`encrypt Kbs ((name A) APP Kab APP Ts)`--));

e(MATCH_MP_TAC (axiom "crypto_90" "M1"));

e(EXISTS_TAC (--`Kas`--));

e CONJ_TAC; (*Begin proof that: A recv encr....*)

e(MATCH_MP_TAC (axiom "crypto_90" "R1"));

e(EXISTS_TAC (--`Svr`--));

e CONJ_TAC;(*Begin proof that Svr sent it*)

e(PURE_REWRITE_TAC [definition "crypto_90" "APP_def"]);

e(ASSUME_TAC (axiom "ds_90" "dsm2"));

e(RULE_ASSUM_TAC
     (PURE_REWRITE_RULE [definition "crypto_90" "APP_def"]));

e(PURE_REWRITE_TAC [theorem "list" "APPEND_ASSOC"]);

e(RULE_ASSUM_TAC (PURE_REWRITE_RULE [theorem "list" "APPEND_ASSOC"]));

e(PURE_ASM_REWRITE_TAC[]);

(*The above used again soon so could be a lemma, but it's short*)

(*Proved: send Svr A (encrypt Kas... *)
(*New subgoal: elig Svr (encrypt Kas... *)

e(PURE_REWRITE_TAC [definition "crypto_90" "APP_def"]);

e(ASSUME_TAC (theorem "ds_proof_90" "Lemma_Svr_elig"));

e(RULE_ASSUM_TAC
     (PURE_REWRITE_RULE [definition "crypto_90" "APP_def"]));

e(PURE_REWRITE_TAC [theorem "list" "APPEND_ASSOC"]);

e(RULE_ASSUM_TAC (PURE_REWRITE_RULE [theorem "list" "APPEND_ASSOC"]));

e(PURE_ASM_REWRITE_TAC[]);

(*Proved: elig Svr (encrypt Kas... *)
(*Thus proved: (receive A (encrypt Kas... ) *)

e CONJ_TAC;

(*Subgoal: `theorem (possesses A Kas)`*)

e(REWRITE_TAC (map snd (axioms "ds_90")));

(*Proved it.*)

e CONJ_TAC;

(*subgoal: believes A (is_shared_ ... *)

e(REWRITE_TAC (map snd (axioms "ds_90")));

(*Proved: believes A (is_shared_ ... *)

e CONJ_TAC;

(*Next prove: believes A (is_recog ((name B... *)

e(MATCH_MP_TAC (axiom "crypto_90" "G1"));

e(MATCH_MP_TAC (axiom "crypto_90" "G1"));

e(REWRITE_TAC (map snd (axioms "ds_90")));
```

```
(*Proved: believes A (is_recog ((name B... *)
(*New subgoal: believes A (is_fresh (((name B ... *)

e(MATCH_MP_TAC (axiom "crypto_90" "F1"));

e(MATCH_MP_TAC (axiom "crypto_90" "F1"));

e(MATCH_MP_TAC (axiom "crypto_90" "F2"));

e(MATCH_MP_TAC (axiom "crypto_90" "F2"));

e(REWRITE_TAC (map snd (axioms "ds_90")));

(*subgoal proved*) (*Next a bunch of conjuncts*)

e(CONJ_TAC);(*Next: believes A (is_fresh ... *)

e(MATCH_MP_TAC (axiom "crypto_90" "F2"));

e(MATCH_MP_TAC (axiom "crypto_90" "F2"));

e(REWRITE_TAC (map snd (axioms "ds_90")));

(*subgoal proved*) (*Next a bunch of conjuncts*)

e(CONJ_TAC);(*Next: possesses B Kab *)

e(MATCH_MP_TAC (axiom "crypto_90" "P1"));

e(MATCH_MP_TAC (axiom "crypto_90" "R2"));

e(EXISTS_TAC (--‘Ts‘--));

e(MATCH_MP_TAC (axiom "crypto_90" "R3"));

e(EXISTS_TAC (--‘name A‘--));

e(MATCH_MP_TAC (axiom "crypto_90" "R5"));

e(EXISTS_TAC (--‘Kbs‘--));

e(CONJ_TAC);

(*Next subgoal: receive B (encrypt Kbs ... *)
e(MATCH_MP_TAC (axiom "crypto_90" "R1"));

e(EXISTS_TAC (--`A`--));

e(CONJ_TAC);

(*Next subgoal: send A B (encrypt Kbs ... *)

e(REWRITE_TAC (map snd (axioms "ds_90")));

(*Proved*)

(*Next: elig A (encrypt Kbs ... *)

e(MATCH_MP_TAC (axiom "crypto_90" "E1"));

e(MATCH_MP_TAC (axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`Ts`--));

e(MATCH_MP_TAC (axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`(name B) APP Kab`--));

e(REWRITE_TAC (map snd (theorems "ds_proof_90")));

e(REWRITE_TAC (map snd (axioms "ds_90")));

e CONJ_TAC;

(*subgoal: believes B (convey Svr (name A ... *)

e(MATCH_MP_TAC (axiom "crypto_90" "M1"));

e(EXISTS_TAC (--`Kbs`--));

e CONJ_TAC;

e(MATCH_MP_TAC (axiom "crypto_90" "R1"));

e(EXISTS_TAC (--`A`--));

e CONJ_TAC;
```

```
e(REWRITE_TAC (map snd (axioms "ds_90")));

(*Proved: send A B (encrypt Kbs ... *)
(*New subgoal: elig A (encrypt Kbs ...*)

e(MATCH_MP_TAC (axiom "crypto_90" "E1"));

e(MATCH_MP_TAC (axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`Ts`--));

e(MATCH_MP_TAC (axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`(name B) APP Kab`--));

e(REWRITE_TAC (map snd (theorems "ds_proof_90")));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ds_90")));

e CONJ_TAC;

(*subgoal: believes B (is_shared_secret ... *)

e(REWRITE_TAC (map snd (axioms "ds_90")));

e CONJ_TAC;

e(MATCH_MP_TAC (axiom "crypto_90" "G1"));

e(REWRITE_TAC (map snd (axioms "ds_90")));

e(MATCH_MP_TAC (axiom "crypto_90" "F1"));

e(MATCH_MP_TAC (axiom "crypto_90" "F2"));

e(MATCH_MP_TAC (axiom "crypto_90" "F2"));

e(REWRITE_TAC (map snd (axioms "ds_90")));

e(MATCH_MP_TAC (axiom "crypto_90" "F2"));
```

```
e(MATCH_MP_TAC (axiom "crypto_90" "F2"));

e(REWRITE_TAC (map snd (axioms "ds_90")));

(*---------------End of proof of postcondition----------------*)

export_theory();

close_theory();
```

5.B Appendix: Proof Transcript for the Needham-Schroeder Protocol

```
new_theory("ns_proof_90");

new_parent "ns_90";

(*---------------------Lemma-------------------------------*)

set_goal([],
    --`theorem(possesses Svr((name A) APP ((name B) APP Na)))`--);


e(MATCH_MP_TAC (axiom "crypto_90" "P1"));

e(MATCH_MP_TAC (axiom "crypto_90" "R1"));

e(EXISTS_TAC (--`A`--));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e(MATCH_MP_TAC (axiom "crypto_90" "E1"));

e(MATCH_MP_TAC (axiom "crypto_90" "P2"));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e(MATCH_MP_TAC (axiom "crypto_90" "P2"));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e(REWRITE_TAC (map snd (axioms "ns_90")));

save_top_thm "Lemma_Svr_poss";

(*End proof*)
```

```
(*--------------------Lemma------------------------------*)

set_goal([], --`theorem
 (elig Svr (encrypt Kas (Na APP (name B) APP Kab APP
                        (encrypt Kbs (Kab APP (name A))))))`--);

(* For some parts of this proof, we have two choices -- 1) to prove
 possession of the names of A and B by virtue of an initial
 assumption of possession, or 2) to prove possession of A and B by
 virtue of receiving them from A.  The logic exposes this aspect.
*)

e(MATCH_MP_TAC (theorem "crypto_90" "elig_encr")  );

e(EXISTS_TAC (--`A`--));

e CONJ_TAC;

e(MATCH_MP_TAC (axiom "crypto_90" "E3"));

e CONJ_TAC;

e(MATCH_MP_TAC (axiom "crypto_90" "E1"));

(*Now proving that --`theorem(possesses Svr Na)".
 We will APP to get the whole message sent by P.
 Then use Lemma_Svr_poss.
*)

e(MATCH_MP_TAC (axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`name B`--));

e(MATCH_MP_TAC (axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`name A`--));

e(MATCH_ACCEPT_TAC (theorem (current_theory()) "Lemma_Svr_poss"));

(*Using current_th above may not be as good as ns-proof*)

(*Now proceed to proof of
 (elig Svr((name B) APP (Kab APP (encrypt Kbs(Kab APP (name A))))))
 where we will immediately resort to possession before splitting into
 components, as it works and there are less steps (no extensions).
 It is more powerful in general to keep to elig though we don't here.
*)

e(MATCH_MP_TAC (axiom "crypto_90" "E1"));

e(MATCH_MP_TAC (axiom "crypto_90" "P2"));

e CONJ_TAC;

e(MATCH_MP_TAC (axiom "crypto_90" "P3"));

e(EXISTS_TAC (--`Na`--));

e(MATCH_MP_TAC (axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`name A`--));

e(MATCH_ACCEPT_TAC (theorem (current_theory()) "Lemma_Svr_poss"));

(*Now to prove possession of Kab, the next component.*)

e(MATCH_MP_TAC (axiom "crypto_90" "P2"));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

(*Now prove Svr poss of the encrypted final part of the message to B.*)

e(MATCH_MP_TAC (axiom "crypto_90" "P5"));

e CONJ_TAC;

e(MATCH_MP_TAC (axiom "crypto_90" "P2"));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e(MATCH_MP_TAC (axiom "crypto_90" "P3"));

e(EXISTS_TAC (--`(name B) APP Na`--));
```

```
e(MATCH_ACCEPT_TAC (theorem (current_theory()) "Lemma_Svr_poss"));

e(REWRITE_TAC (map snd (axioms "ns_90")));

(*End of the possession proofs, probably.*)

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e(PURE_REWRITE_TAC [axiom "ns_90" "nse4"]);

e(REWRITE_TAC (map snd (axioms "crypto_90")));

(*End of proof*)

save_top_thm "Lemma_Svr_elig";

(*End proof*)

(*---------------------Lemma--------------------------------*)

set_goal([], --`theorem(possesses A (Na APP ((name B) APP
                    (Kab APP (encrypt Kbs(Kab APP (name A)))))))`--);


e(MATCH_MP_TAC (axiom "crypto_90" "P6"));

e(EXISTS_TAC (--`Kas`--));

e CONJ_TAC;

e(MATCH_MP_TAC (axiom "crypto_90" "P1"));

e(MATCH_MP_TAC (axiom "crypto_90" "R1"));

e(EXISTS_TAC (--`Svr`--));
```

```
e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e(MATCH_ACCEPT_TAC (theorem (current_theory()) "Lemma_Svr_elig"));

(*Proved elig of Svr using lemma*)

e(REWRITE_TAC (map snd (axioms "ns_90")));

save_top_thm "Lemma_A_poss";

(*End of proof*)

(*-----------Proof of the parts of the Postcondition--------------*)

(*-----------Lemma: conjunct 1 of postcondition----------------------*)

set_goal([], --`theorem(possesses A Kab)`--);

e(MATCH_MP_TAC(axiom "crypto_90" "P3"));

e(EXISTS_TAC (--`(encrypt Kbs(Kab APP (name A)))`--));

e(MATCH_MP_TAC(axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`name B`--));

e(MATCH_MP_TAC(axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`Na`--));

e(MATCH_ACCEPT_TAC (theorem (current_theory()) "Lemma_A_poss"));

save_top_thm "Lemma_A_poss_Kab";

(*End of proof*)

(*-----------Lemma: Conjunct 2 of postcondition----------------------*)

set_goal([], --`theorem(believes A
                                (is_shared_secret A B Kab))'--);

e(MATCH_MP_TAC(axiom "crypto_90" "J1"));

e(EXISTS_TAC (--'Svr'--));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

(*right here we have to prove
 --'theorem(believes A(believes Svr(is_shared_secret A B Kab))"
 see file ns.ml for desired alteration to GNY logic.
 We have to go through J3 as follows.
*)

e(MATCH_MP_TAC(axiom "crypto_90" "J3"));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

(*Now we're back to the same proof rules as with the alternative
 description, with different extensions as suggested.
*)

e(MATCH_MP_TAC(axiom "crypto_90" "J2"));

e(EXISTS_TAC (--'(encrypt Kas (Na APP (name B) APP Kab APP
                          (encrypt Kbs (Kab APP (name A)))))'--));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e CONJ_TAC;

(*Now rotate and separate conjuncts in order to get at the extension
    part.*)

rotate(1);

e CONJ_TAC;
```

```
rotate(1);

e(REWRITE_TAC (map snd (axioms "ns_90")));

(*This has proved the extension part.*)

e(MATCH_MP_TAC (axiom "crypto_90" "F3"));

e CONJ_TAC;

e(MATCH_MP_TAC (axiom "crypto_90" "F1"));

e(REWRITE_TAC (map snd (axioms "ns_90")));

e(REWRITE_TAC (map snd (axioms "ns_90")));

(*Now we prove (believes A (convey Svr (encrypt Kas (Na APP.....))))*)

(*Now we use M2*)

e(MATCH_MP_TAC (axiom "crypto_90" "M2"));

(*Done.  Now for the antecedents of M2*)

e CONJ_TAC;

e(MATCH_MP_TAC (axiom "crypto_90" "R1"));

e(EXISTS_TAC (--`Svr`--));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e(MATCH_ACCEPT_TAC (theorem (current_theory()) "Lemma_Svr_elig"));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));
```

```
e CONJ_TAC;

(*Next the proof that
    A recog (Na APP ((name B) APP (Kab APP (encrypt Kbs...*)

e(MATCH_MP_TAC (axiom "crypto_90" "G2"));

e(MATCH_MP_TAC (axiom "crypto_90" "G1"));

e(REWRITE_TAC (map snd (axioms "ns_90")));

(*Next the proof that
    A bel fresh (Na APP ((name B) APP (Kab APP (encr...*)

e(MATCH_MP_TAC (axiom "crypto_90" "F1"));

e(MATCH_MP_TAC (axiom "crypto_90" "F1"));

e(REWRITE_TAC (map snd (axioms "ns_90")));

save_top_thm "Lemma_A_secret_Kab";

(*End of proof*)

(*-----------Lemma: Conjunct 3 of postcondition-------------------*)

set_goal([], --`theorem(possesses B Kab)`--);

e(MATCH_MP_TAC(axiom "crypto_90" "P3"));

e(EXISTS_TAC (--`name A`--));

e(MATCH_MP_TAC(axiom "crypto_90" "P6"));

e(EXISTS_TAC (--`Kbs`--));

e CONJ_TAC;

e(MATCH_MP_TAC(axiom "crypto_90" "P1"));

e(MATCH_MP_TAC(axiom "crypto_90" "R1"));

e(EXISTS_TAC (--`A`--));
```

```
e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

(*Now for the proof that A was elig to send to B*)

e(MATCH_MP_TAC(axiom "crypto_90" "E1"));

e(MATCH_MP_TAC(axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`Kab`--));

e(MATCH_MP_TAC(axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`name B`--));

e(MATCH_MP_TAC(axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`Na`--));

e(MATCH_ACCEPT_TAC (theorem (current_theory()) "Lemma_A_poss"));

e(REWRITE_TAC (map snd (axioms "ns_90")));

save_top_thm "Lemma_B_poss_Kab";

(*End of proof*)


(*-----------Lemma: Conjunct 4 of postcondition-----------------------*)

set_goal([], --`theorem(believes B
                            (is_shared_secret A B Kab))`--);


e(MATCH_MP_TAC(axiom "crypto_90" "J1"));

e(EXISTS_TAC (--`Svr`--));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

(*Next prove (believes A(believes Svr(is_shared_secret A B Kab)))
 We have to go through J3 as follows.
*)

e(MATCH_MP_TAC(axiom "crypto_90" "J3"));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e(MATCH_MP_TAC(axiom "crypto_90" "J2"));

e(EXISTS_TAC (--`encrypt Kbs (Kab APP (name A))`--));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e CONJ_TAC;

(*Now rotate and separate conjuncts in order to get at the extension
     part.*)

rotate(1);

e CONJ_TAC;

rotate(1);

e(REWRITE_TAC (map snd (axioms "ns_90")));

(*This has proved the extension part.*)

e(MATCH_MP_TAC (axiom "crypto_90" "F3"));

e CONJ_TAC;

(*Next proof obligation:
 --`theorem(believes B(is_fresh(Kab APP (name A))))"
 Here's the weakness in the N-S protocol.  It is impossible to prove
 this.  One thing is to add to the assumptions
 --`theorem(believes B(is_fresh(Kab )))",
 as they did in BAN, and point out that it is a dubious assumption:
 B just gets some key and assumes that it is fresh.  See file ns.ml
*)
```

```
e(MATCH_MP_TAC (axiom "crypto_90" "F1"));

e(MATCH_ACCEPT_TAC (axiom "ns_90" "nsb10_dubious_assumption"));

(*Proved the bit requiring freshness*)

e(REWRITE_TAC (map snd (axioms "ns_90")));

(*Now we prove
     (believes B (convey Svr (encrypt Kbs(Kab APP (name A)))))*)

(*Now we need to use M2 *)

e(MATCH_MP_TAC (axiom "crypto_90" "M2"));

(*Done.  Now for the antecedents of M2*)

e CONJ_TAC;

e(MATCH_MP_TAC (axiom "crypto_90" "R1"));

e(EXISTS_TAC (--`A`--));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e(MATCH_MP_TAC (axiom "crypto_90" "E1"));

e (MATCH_MP_TAC (axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`Kab`--));

e (MATCH_MP_TAC (axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`name B`--));

e (MATCH_MP_TAC (axiom "crypto_90" "P4"));

e(EXISTS_TAC (--`Na`--));

e(MATCH_ACCEPT_TAC (theorem (current_theory()) "Lemma_A_poss"));
```

```
e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e CONJ_TAC;

e(REWRITE_TAC (map snd (axioms "ns_90")));

e CONJ_TAC;

(*Next the proof that
     A recog (Na APP ((name B) APP (Kab APP (encrypt Kbs...*)

e (MATCH_MP_TAC (axiom "crypto_90" "G2"));

e(REWRITE_TAC (map snd (axioms "ns_90")));

(*Next the proof that B bel fresh ((Kab APP (name A)) APP Kbs)*)

e(MATCH_MP_TAC (axiom "crypto_90" "F1"));

e(MATCH_MP_TAC (axiom "crypto_90" "F1"));

e(MATCH_ACCEPT_TAC (axiom "ns_90" "nsb10_dubious_assumption"));

save_top_thm "Lemma_B_secret_Kab";

(*End of proof*)


(*---------Proof of postcondition-------------------------------*)

set_goal([], --`postcondition`--);

e(PURE_ONCE_REWRITE_TAC [definition "ns_90" "postcond"]);

e(CONJ_TAC);

e(MATCH_ACCEPT_TAC (theorem (current_theory()) "Lemma_A_poss_Kab"));

e(CONJ_TAC);

e(MATCH_ACCEPT_TAC (theorem (current_theory()) "Lemma_A_secret_Kab"));
```

```
e(CONJ_TAC);

e(MATCH_ACCEPT_TAC (theorem (current_theory()) "Lemma_B_poss_Kab"));

e(MATCH_ACCEPT_TAC (theorem (current_theory()) "Lemma_B_secret_Kab"));

(*Proof of the postcondition successful*)

export_theory();

close_theory();
```

Chapter 6

Multilevel Distributed Database Integrity Model

6.1 Introduction

In this section, we describe a method of building multilevel databases that handles both integrity and nondisclosure, and we provide a formal specification of the trusted part of the method. Rather than deriving a new solution to this problem, this model is based on the recent unpublished work "Data Replication and Multilevel-Secure Transaction Processing" by Boris Kogan and Sushil Jajodia [11]. In their paper, they present a method that constructs a database architecture that is restrictive and satisfies the integrity property, one-copy serializability. This method appears to be a practical way of constructing multilevel replicated databases. [1] For further discussion of integrity and nondisclosure theories, see Volume II of the Romulus Documentation Set.

The theoretical content of our model is directly derived from [11] and, in this regard, it is not an extension of their results. We have added a formal specification for the trusted part of their protocol. In the paper, the authors present both the centralized and completely distributed form. Here, we present a minor generalization that permits any number of individual databases to reside at a particular host.

[1]Called distributed DBMS in [1].

6.2 Serializability

Serializability is the standard integrity property for concurrency control in databases. Serializability ensures that if a number of transactions are interleaved, then the effect will be equivalent to some scenario in which the transactions are not interleaved.

Here is an intuitive explanation of why this property is desired. Suppose each transaction leaves the database in a "good" state. Any possible history on a serializable system will be equivalent to some non-interleaved sequence of transactions, and hence, by induction, we can conclude that the database is left in a "good" state. Note that serializability by itself only handles part of the integrity problem. There is still the obligation for each individual transaction to be sufficiently correct.

When dealing with distributed databases, we need a modified form of serializability, to handle the fact that there are multiple versions of the same information on different hosts. This form of serializability is called one-copy serializability.

In their paper, Kogan and Jajodia show that their protocol is one-copy serializable. This model is a formalization of the trusted part of that protocol.

It is worth noting that in some cases serializability is not the appropriate integrity property. It may be that the concept of transaction is simply not present. For example, a database may continuously receive the results of some sensor, and hence there is no useful notion of transaction. Or, it may be that orderings of reads and writes do not impact the desired correctness conditions. However, serializability is usually the appropriate property.

For a good introduction to the field see [5].

6.3 The Architecture

The replicated database consists of a number of different interconnected databases called *containers* in the paper (see Figure 6.1). Each individual database contains information at or below some security level. In fact, in this model it contains all of the information at or below that level (data at lower levels is passed up, i.e., replicated on the machines of higher security levels). In Romulus terminology, each container is single-level. It is not trusted to keep information segregated by level.

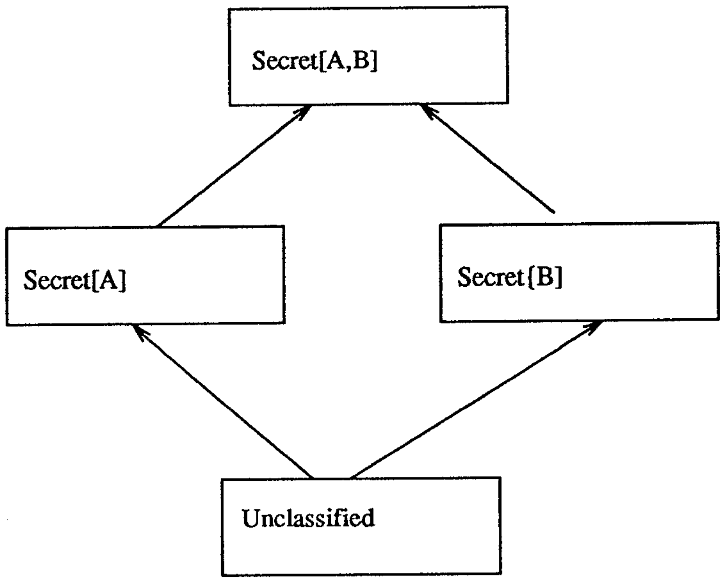

Figure 6.1: Data Flow Diagram of Typical Configuration

Information can flow up from a low- to a high- level database, but it is not permitted to flow in the opposite direction. The secure network and/or the operating TCB prevents the flow from going in the wrong direction.

If each database is on a different host, then from the point of view restrictiveness, this kind of system is manifestly secure. All of the nondisclosure security is handled by the network. If multiple databases are on one machine, we can handle the synchronization between databases by single level processes (one for each level). In this case, the system is also manifestly secure.

An alternative approach is to group all of the synchronization responsibility between databases on a particular host into one process. In this case, the synchronization process must be trusted (MLS), but it may be able to more efficiently manage shared updates. This model uses this MLS approach.

6.4 The Integrity and Security Properties

6.4.1 Integrity

For integrity, we want some control on propagation of information from low to high levels so as to preserve database consistency (i.e., one-copy serializability). For security reasons, we do not want communication to databases at lower levels, so it is not trivial to synchronize the updates. Here we just discuss the trusted part of the protocol.

First, transactions on a given MLS subsystem must be passed to the TCB and then relayed to the appropriate container. Secondly, updates from lower level containers must be queued on arrival and then delivered to the appropriate container if they are not blocked.

The queue handling is accomplished as follows. If an update is received for the subsystem from a container at level m (i.e., C_m), then that update is placed on each queue $Q_{m,j}$, where j is a security level of the subsystem that immediately dominates m. (The level j immediately dominates m, provided j is a more sensitive security classification and there is no other level between j and m.)

The subsystem will periodically send the TCB protocol wake-up signals to forward any unblocked updates. If an update is unblocked, it will be removed from the queue and forwarded to the destination container (container j if it was on $Q_{i,j}$).

We next describe when a queue is unblocked. Suppose a is at the head of $Q_{i,k}$. It can be sent, provided that all other queues $Q_{j,k}$ that might contain a (i.e., for which the level of a is immediately dominated by j) have a at the front of their list.

6.4.2 Nondisclosure

There are two different parts to the overall nondisclosure security of the database.

First, the message handling and delivery part of system must properly screen outgoing messages and be able to identify the level of incoming messages. This is part of the underlying TCB of the system, but not part of this model. If the system is distributed, the screening of outgoing messages could be handled by the network.

Second, the secure part of the protocol must place and remove updates from the queues, and send messages to containers at the right levels. The security justification is straightforward, as information is stored and forwarded with a possible upgrade of levels in-between. This nondisclosure property, and the absence of potential covert storage channels, can be demonstrated by showing that the protocol is restrictive.

6.5 Formal Specification of the TCB Extension

Part of the protocol must be handled by the TCB. (It is called *P2* in the paper and is informally discussed in section 6.4.1.) Here we describe a Romulus style model of the protocol.

This HOL specification is for the trusted part of the database protocol of Kogan and Jajodia 1993. It is for a distributed database and only assumes that some of the containers are on any given subsystem.

It uses the definitions and theorems about recursive-transitive closures.

```
load_theory "rtc";
new_theory "db_spec";
```

We use a number of abbreviations for the presentation of the specification that can be automatically expanded before theorem proving. These are `all`, `exists`, `some`, `false`, and `true`, which respectively stand for the HOL symbols `!`, `?`, `@`, `F`, and `T`. We also introduce the ML function `defcontype` as an alternative to the HOL `define_type`, due to a bug in the current version of HOL. It is similar to *romcontype* used in other Romulus specifications.

```
use "style.sml";
```

This is a specification of the secure part of the protocol for a subsystem (collection of databases on a single host). The collection of these subsystems forms the distributed database.

We assume

- that other parts of the operating system TCB can assure that the security level of local transactions can be correctly identified (if the subsystem is multilevel),
- that the network TCB will properly identify the security of messages to the protocol TCB, and
- that the network TCB and/or the OS TCB can screen out container messages to subsystems at lower levels.

We first introduce the basic security functions and the objects to be handled. The levels `level` and `dom` are the MAC security levels and their ordering. The set of levels is unconstrained.

```
new_type{Arity= 0, Name= "level"};
```

The primitive dominates is for adjacent levels in the dominates lattice.

```
new_constant{Name="dom_primitive",
             Ty = ==`:level#level -> bool`==};
```

We introduce an axiom requiring that it is really primitive.

```
expand_axiom("dom_prim",
   --`all i. all j. (dom_primitive(i,j) /\ dom_primitive(j,k)
      ==> ((i=j) \/ (j=k)))`--);
```

We introduce curried versions of functions to facilitate theorem proving.

```
new_definition("domcp",
   --`domcp (a:level) (b:level)= dom_primitive(a,b)`--);
```

We build the recursive transitive closure.

```
new_definition("domc",--`domc (a:level) (b:level)=(RTC domcp a b)`--);
new_definition("dom", --`dom ((a:level),(b:level))=domc a b`--);
```

Each database at a particular level is called a container.

```
define_contype "containertype"
   [
      ("container", [==`:level`==])
   ];

new_recursive_definition {
 name = "containerlevel",
 fixity = Prefix,
 rec_axiom = (theorem "-" "containertype_def"),
 def=
     (--`containerlevel (container l) =l`--)
   };
```

Each container is on some subsystem. There may be one or multiple subsystems.

```
new_type{Arity= 0, Name= "subsystem"};
```

`leveltosystem` identifies the subsystem where the container at level `l` resides.

```
new_constant{Name= "leveltosystem", Ty= ==`:level -> subsystem`==};
```

The two kinds of transaction operations are reads and writes; they operate on the data of some container.

```
new_type{Arity= 0, Name= "datatype"};
```

We use the meta-language function `define_contype` that is defined in `style.sml` instead of `define_type`, because of a bug in HOL.

Operations on data are either reads or writes. We also explicitly identify the level of request.

```
define_contype "op"
   [("read", [==`:level#datatype`==]),
    ("write",[==`:level#datatype`==])];

new_recursive_definition {
   name = "oplevel",
   fixity = Prefix,
   rec_axiom = (theorem "-" "op_def"),
   def=
     (--`(oplevel (read a) =(FST a)) /\
         (oplevel (write a) =(FST a))`--)
   };
```

The Events are

- Updates are received from a container at some level and sent to a container at some level. Note that all updates should be write operations.
- Transactions from and to container k. (They are always local.)
- Timing signals.

```
define_contype "events"
   [
      ("update", [==`:level#op`==]),
      ("transaction",[==`:level#op`==]),
      ("time",[])
   ];
```

We introduce projection function names for extracting event information.

```
new_definition("levelof_def",
   --`levelof (e:level#op) =FST(e)`--);
new_definition("transop_def",
   --`opof (e:level#op) =SND(e)`--);
```

We need to distinguish the three kinds of events.

```
define_contype "eventkinds"
   [
      ("updatekind",[]),
      ("transactionkind",[]),
      ("timekind",[])
```

```
    ];

  new_recursive_definition {
     name = "kindof",
     fixity = Prefix,
     rec_axiom = (theorem "-" "events_def"),
     def=
        (--`(kindof (update a) = updatekind ) /\
        (kindof (transaction b) = transactionkind) /\
        (kindof (time)= timekind)`--)
     };
```

We introduce a convenient function for extracting the information from update events. In order to make it a definition on the entire type we extend it to the other cases.

```
  new_recursive_definition {
     name = "extractinfo",
     fixity = Prefix,
     rec_axiom = (theorem "-" "events_def"),
     def=
        (--`(extractinfo (transaction a) =a) /\
            (extractinfo (update b) =b) /\
            (extractinfo (time)=(some x:(level#op).true))`--)
     };
```

We introduce some constant levels. The levels of this subsystem are between subsystemhigh and subsystemlow.

```
  new_constant{Name="subsystemlow",Ty = ==`:level`==};
  new_constant{Name="subsystemhigh",Ty = ==`:level`==};
  expand_axiom("subsystemlevels",
      --`all l. (dom(l,subsystemlow) /\ dom(subsystemhigh,l))`--);
```

Timing signals are the lowest level event that can occur on the specified subsystem of containers.

The level of an update event will be considered to be the write-up level.

```
  new_recursive_definition {
     name = "evlevel",
     fixity = Prefix,
     rec_axiom = (theorem "-" "events_def"),
     def=
        (--`(evlevel (transaction a) =(levelof(a))) /\
            (evlevel (update b) = (levelof(b))) /\
            (evlevel (time)= subsystemlow)`--)
     };
```

We introduce functions `Input` and `Output` to distinguish events coming in and going out. We extend the level definitions to these types.

```
  define_contype "InEv"
     [
        ("Input", [==`:events`==])
     ];

  new_recursive_definition {
     name = "inputof",
     fixity = Prefix,
     rec_axiom = (theorem "-" "InEv_def"),
     def=
        (--`inputof (Input e) =e`--)
     };

  define_contype "OutEv"
     [
        ("Output", [==`:events`==])
     ];

  new_recursive_definition {
     name = "outputof",
     fixity = Prefix,
     rec_axiom = (theorem "-" "OutEv_def"),
     def=
        (--`outputof (Output e) =e`--)
     };

  new_recursive_definition {
     name = "inputlev",
     fixity = Prefix,
     rec_axiom = (theorem "-" "InEv_def"),
     def=
        (--`inputlev (Input e) = evlevel(e)`--)
     };

  new_recursive_definition {
    name = "outputlev",
    fixity = Prefix,
    rec_axiom = (theorem "-" "OutEv_def"),
    def=
       (--`outputlev (Output e) = evlevel(e)`--)
    };

(*-----------------------------------------------------------*)
```

For composition analysis, we associate ports with these events. They indicate from where and to where the events are passing.

```
define_contype "porttype"
   [
   (* The input port for this part of the TCB *)
      ("tcb_inport",[]) ,
   (* Ports to containers *)
      ("to_container",[==`:level`==]),
   (* Incoming internal timing signal *)
      ("timeport",[]),
   (* time_events are actually never sent out,
         but we use a place holder *)
      ("dummyport",[])
   ];


new_recursive_definition {
   name = "outportof",
   fixity = Prefix,
   rec_axiom = (theorem "-" "events_def"),
   def=
      (--`(outportof (update a) = to_container(levelof(a))) /\
          (outportof (transaction b) = to_container(levelof(b))) /\
          (outportof (time)= dummyport)`--)
   };

(*-----------------------------------------------------------*)
```

The object being maintained is a collection of queues of update requests from container `i` to container `j`. It is of type `level # level  -> op list`.

```
new_type{Arity= 0, Name= "queue"};
```

We would like to use type_abbreviations, but they are not currently supported. We use an ML abbreviation for queues and later use antiquotation when it needs to be used.

```
new_type_abbrev("queues",==`: level # level -> op list`==);
val queues= ty_antiq(==`:(level # level -> op list)`==);
```

There is not a list for every pair of indices. It is determined by the set of supported levels. Indeed (i,j) can only be in the domain if dom_primitive(j,i); so we introduce the predicate exists_queue.

```
new_definition("exists_queue",
    --`exists_queue(i,j) = dom_primitive(j,i)`--);
```

The Queues are the persistent data of the processes, so we need to introduce a security projection function for them. Again we introduce auxiliary curried definitions.

The queue, Q(i,j), contains information destined for level j. It can be considered to only be in the view of j or above. That is, when an update is added to Q(i,j) it can already be considered written-up. So it suffices to project out entire queues.

```
new_definition ("projcur",
   --`projc Q l (a:(level # level)) =
      (dom( l,(SND a)) =>
         (Q a)
      |
         []:(op list))`--);

new_definition("proj",
   --`proj(Q,l) = (projc Q l)`--);

(*-------------------------------------------------------------*)
```

We now introduce auxiliary functions for describing the TCB extension for the database.

Incoming updates must be added to the appropriate queue.

```
new_definition("enqueue_c",
   --`enqueue_c (a:level) (e:op) (Q:^queues) (b:level#level) =
      (((exists_queue(b)) /\ (a=(FST b))) =>
           (CONS e (Q(b)))
        |
           Q(b))`--);

new_definition("enqueue_def",
   --`enqueue(a,e,Q)=
      enqueue_c a e Q `--);
```

We use the following list operations: `inlist`, `last`, and `butlast`.

```
new_list_rec_definition ("inlist_def",
   --`(inlist (NIL) (a:'t) =false) /\
      (inlist (CONS (h:('t)) l) a = ((a=h) \/ inlist l a))`--);

(* Note that the last item that was put in, is the first item out.
    It is a queue. *)
new_list_rec_definition("last_def",
   --`(last (NIL:('t list))  = (some x:'t. true)) /\
      (last (CONS (h:'t) (l:('t list))) =
         ((l=NIL) => h | (last  l)  ))`--);

(* A queue after the last item is removed *)
new_list_rec_definition("butlast_def",
   --`(butlast (NIL:('t list))  = []) /\
      (butlast (CONS (h:'t) (l:('t list))) =
         ((l=NIL) => [] | (CONS h (butlast l)  )))`--);

new_list_rec_definition("removeduplicates",
   --`(removedup ([]:('t list))=[]) /\
      (removedup (CONS (a:'t) (l:('t list)))  =
         ((inlist l a) => l | (CONS a l)))`--);
```

When we get a timing signal, we must check the queues and send out the front update transactions that are not blocked.

Suppose `l` is at the head of `Q(i,k)`. It can be sent, provided that all other queues `Q(j,k)` that might contain `l`, have `l` at the front of their list.

We introduce the function `list_of_indices` to describe which items can be removed from the front of the queues. (The definition impacts only integrity, not nondisclosure.)

```
new_constant{Name="list_of_indices",
             Ty=  ==`:^queues -> ((level # level) list)`==};
```

```
expand_axiom("list_of_indices_ax",
   --`all i.  all k.
      ( (inlist  (list_of_indices(Q)) (i,k)) =
         ( (~(Q(i,k)=[])) /\
            (let o=(last (Q(i,k))) in
               (let l=(oplevel(o)) in
                  (all j.
                      (dom(j,l) ==> (o=last (Q(j,k))))))))`--);
```

We describe the list of outputs that should be sent for the specified list of queues. It is slightly over specified in that the update transactions to container i need not come before the update transactions to container j, when dom(j ,i). We define the list with duplications and then remove the duplicates.

```
new_list_rec_definition("formdupmsglist",
   --`(formdupmsglist ([]:((level # level) list)) (Q:^queues) =[]) /\
      (formdupmsglist (CONS a l) Q=
         ((CONS (Output (update((SND(a)),(last(Q(a))))))
            (formdupmsglist l Q))))`--)
   handle e => Raise e;

new_list_rec_definition("formmsglist",
   --`((formmsglist ([]:((level # level) list)) (Q:^queues) =[]) /\
       (formmsglist (CONS a l) Q=
          (removedup (formdupmsglist l Q))))`--);
```

We also describe the queues after the items have been removed.

```
new_definition("dequeue_cur",
   --`dequeue_cur (l:((level#level) list))  (Q:^queues)
      (a:(level#level)) =
         ((inlist l a) => (butlast(Q a)) | (Q a))`--)
   handle e => Raise e;

new_definition("dequeue_def",
   --`dequeue(l,Q)=dequeue_cur l Q`--);

(*----------------------------------------------------------*)
```

We will use the the PSL theory of states to describe the database.

```
(* This definition is taken from romproc.sml *)
(define_type {
   name = "PSL_Def",
   type_spec =
      `process = Skip |
          ;; of process#process |
          Orselect of process#process |
          If of bool#process#process |
          Send of 'outev |
          Receive of ('inev -> bool)#('inev -> 'invoc) |
          Call of 'invoc |
          Buffered of ('inev -> bool)#('inev)list#process`,
   fixities =
      [Prefix,Infix 1000,Prefix,Prefix,Prefix,Prefix,Prefix,Prefix]});
```

We specialize the polymorphic process definition to the actual Input events, Output events, and process_calls.

We need names for the processes/procedures that will be invoked.

In the current theory of PSL we must separate input and output handling, so that the process is split into two parts: getting and sending items from the buffer (db), and the event processing (db_action). We must also distinguish between the name of the action to be applied and the actual function. We first describe the names and the types that the function of that name uses. They are called invocations.

```
define_contype "Invoc"
   [
   (* get next db events *)
      ("apply_db", [==`:^queues`==]) ,
   (* handle db events *)
      ("apply_db_action",[==`:^queues`==, ==`:InEv`==])
   ];
```

We now remove the polymorphism. Alternatively, we could have built a process theory specifically for the input events, output events, and invocations.

```
val db_process = ty_antiq(==`:(OutEv,InEv,Invoc)process`==);
```

We introduce then and else to clarify the specification.

```
new_definition("then",
   --`then(P:^db_process)=P`--);
new_definition("else",
   --`else(P:^db_process)=P`--);

(*-----------------------------------------------------------*)
```

We augment the language with the `multisend` construct, which sends out all of the items on a list.

```
new_list_rec_definition("multisend",
   --`(multisend []=Skip) /\
      (multisend (CONS (a:OutEv) l)=
         ((Send a ;; multisend l):(^db_process)))`--)
   handle e => Raise e;

(*-----------------------------------------------------------*)
```

The definition of the database protocol gets the next db event and calls the handler.

```
new_definition("db_def",
   --`db(Q:^queues) =
      ((Receive (\x:InEv.true) (apply_db_action Q)):(^db_process))`--);
```

We next actually process the db event, describing the response for each of the three kinds of events.

```
new_definition("db_action_def",
  --`db_action (Q:^queues) (ine:InEv)=
     ((let e=(inputof(ine)) in
     (If (kindof(e)=transactionkind)
        (then
           (Send(Output e) ;;  (* Pass transaction on to the database *)
           Call(apply_db Q)))
        (else
           (If (kindof(e)=updatekind) (* add updates to the queues*)
              (then
                 ((let e1=(extractinfo(e)) in
                   (let Q1=(enqueue(levelof(e1),opof(e1),Q)) in
                      (Call(apply_db Q1))))))

              (else
(If (kindof(e)=timekind)  (* timing signal to cause
                               handling of updates *)
   (then
      (let l=(list_of_indices(Q)) in
      (let msgs=((formmsglist l Q)) in
         (let Q2=(dequeue(l,Q)) in
         (
            (multisend(msgs));;
            (Call (apply_db Q2)))))))
   (else              (* impossible case *)
      (Call (apply_db(Q)))))))))):(^db_process))'--);
```

6.A Appendix: style.sml

This is the file style.sml used in this chapter.

```
(* some definitions for specification clarity *)
(* all x,exists x,some x, false,true *)
new_binder_definition("all_def",(--`$all = \P:('a->bool).($! P)`--));
new_binder_definition("exists_def",
                                    --`$exists = \P:('a->bool).($? P)`--);
new_binder_definition("some_def",--`$some = \P:('a->bool).($@ P)`--);
new_definition("falsedef", --`false=F`--);
new_definition("truedef", --`true=T`--);

(* defcontype is a simple form of Steve Brackin's romcontype.
   It could eventually be replaced by the HOL define_type,
    when the error in HOL is corrected *)

fun define_contype typename [] = raise Bind
  | define_contype typename constructorlist =
 let
  fun makeclauses [] = []
    | makeclauses ((name,argtypes)::restlist) =
       {constructor = name,
        fixity = Prefix,
        arg = map Parse_support.Hty argtypes}::(makeclauses restlist);
  in
   dtype
    {ty_name = typename,
     save_name = typename^"_def",
     clauses = makeclauses constructorlist}
  end;

(* expand_axiom  -- forms new_open_axiom and
    replaces the style abbreviations with their primitive forms *)

 fun expand_axiom (name,arg) =
 BETA_RULE (
    REWRITE_RULE [(definition "-" "all_def"),
                  (definition "-" "exists_def"),
                  (definition "-" "some_def"),
                  (definition "-" "falsedef"),
                  (definition "-" "truedef")]
             (new_open_axiom(name,arg)));
```

Chapter 7

The Fault Tolerant Reference Monitor Model

7.1 Introduction

The Fault Tolerant Reference Monitor (FTRM) is designed to support multiple clients with arbitrary security classifications accessing data in a hierarchical file system with files and directories at arbitrary security classifications. It is intended to implement an access control policy that allows clients to obtain information only at an equal or lower security level, or output information only at an equal or greater security level. Further, the FTRM design is intended to block so-called "covert storage channels". Such channels are ways of signaling classified information to a client who is not cleared for it by using aspects of the system not normally considered communication media, such as locks on shared resources. The FTRM blocks such channels by a combination of special provisions in the access control policy (described in section 7.2.3) and use of asynchronous communication protocols (described in section 7.2.6).

The FTRM is designed to withstand various kinds of faults in the nodes on the network, including nodes crashing, disks and other devices for maintaining the file system crashing, and corruption of data (either in ordinary files or in files containing records of client and file security levels) on disks.

The FTRM accomplishes fault tolerant multilevel security by replicating files and tables of security levels on multiple network nodes. The distributed data is managed consistently using the ISIS system for fault tolerant communication (described in section 7.2.6). The ISIS protocols mask the various kinds of failures that the system is meant to withstand. The replicated data is used to mask data corruption faults by implementing a collection of voting algorithms on top of ISIS. The FTRM is designed so that any number of faults can be withstood if there are sufficiently many nodes and sufficient replication of file system data and security level tables. In particular, secure mediation of access to data will be unaffected by faults if there is a sufficient degree of replication.

To increase assurance that the FTRM design blocks covert storage channels in the presence of faults, we develop a *formal model* of the FTRM design and specify its security using an information-theoretically based mathematical property called *restrictiveness* (see section 7.3). By making the design of the FTRM and the security property it is intended to satisfy mathematically precise, we have greater confidence that the system is correct and secure.

For further discussion of the availability theory, see Volume II of the Romulus Documentation Set.

Much of the text in this section and the succeeding sections on the Fault Tolerant Reference Monitor, its design, and its security model was drawn from an earlier ORA report on the FTRM [20]. The results presented here primarily differ from the earlier work in the following ways:

- The FTRM's formal model is implemented for HOL90 in Standard ML rather than for HOL88 in pre-standard ML.

- The formal model is complete, including formal specifications of projection, invariant, and initial-state functions that were described only informally in the earlier work.

- The formal model corrects errors in the earlier work, particularly an error that caused later client-process requests to overwrite earlier ones.

- The formal model uses newly developed Romulus techniques and utilities that impose strong type checking on state parameters and message contents; this strong type checking protects against errors in the model, reduces clutter, and provides the basis for easier proofs of security properties.

7.2 FTRM Design

A Fault Tolerant Reference Monitor mediates accesses to a file system in a way that is both secure *and* fault tolerant. The model of file system data assumed by the FTRM is well known. The file system contains files that are classified and labeled at different sensitivity levels. Users are assigned different security clearance levels. The file system is accessed by processes called *clients*. Clients access the file system on behalf of users. A client inherits the security level of the user it is acting for. The FTRM permits or refuses client requested file accesses according to a multilevel mandatory access control policy (and possibly a discretionary policy as well). The access control policy is enforced even in the face of certain failures.

This section describes the overall architecture of the FTRM in informal terms. This description includes the kinds of failure the FTRM will tolerate (usually referred to as the system's *failure semantics*), the data model, the access control policy, the general method for tolerating faults, and the rationale for the design.

7.2.1 Architecture

The FTRM design achieves fault tolerance by replication. The novelty of this design is that it *combines* fault tolerance and security in a single system.

A system built on the FTRM would consist of a number of sites running the FTRM server monitor software and a number of sites that have installed the FTRM user monitor software. A site would typically consist of one machine. for example a Sun workstation, but the possibility of further replication at each site is not ruled out. In addition, each server site has a storage device, for example a hard disk, which stores all or part of the file system, including normal data such as files and security data such as file level labels and user level labels. The simplest instance of this configuration would have a complete copy of the file system residing at every server site, but the FTRM design allows a server site to have only part of a file system in residence. The sites are connected by a network. either local or longhaul, which we will refer to as the *intersite network*. The sites are assumed to be fully connected, that is, there is a communication path between any two sites (which may, in general, pass through other sites) when there is no fault.

7.2.2 Failure Semantics

A system can only be meaningfully described as "fault tolerant" with respect to certain types of failure. The failures a system is intended to withstand are referred to as the system's *failure semantics.* It is important to specify the failure semantics of a fault tolerant system so as not to give the users of the system false confidence in the system's reliability. It is also important because the more general the kinds of fault a system is meant to withstand, the more expensive it is to build and run.

In some fault tolerant systems, if there is sufficient replication, it is possible to construct the system so that it can withstand a certain number of arbitrary, or (in the jargon of fault tolerance) *Byzantine*, faults and still function normally. In a system where "normal functioning" implies "secure functioning", it is much more difficult to withstand even one Byzantine fault in a processor that is running the software intended to enforce security. If a site handling classified data has any direct link to its environment, such as a terminal or phone link, a Byzantine fault in the security function could allow classified data to be released to people with insufficient classification to see it. This compromise can also happen if a site has an indirect connection to its environment, such as a connection to a phone link or other "dumb" device through the intersite network. Even if a site's only connection to its environment is through the intersite network, and the only devices connected to the intersite network are other sites running secure, fault tolerant software, a Byzantine fault in the security function of a single site could still compromise security if the intersite network was not secure. For example, if the intersite network was subject to eavesdropping, a Byzantine fault in a site's security function could allow classified data to be released to an eavesdropper.

The main failure the FTRM is designed to tolerate is arbitrary data corruption caused by disk failures. For example, the classification label of a file (a special kind of data) can be corrupted on some servers, and the FTRM will not grant access to the file in violation of the access control policy. The other failures tolerated are machine crashes. By "machine crashes" we mean that a machine can fail, but is assumed to stop when a failure occurs. This failure semantics is commonly assumed, and it is adequate for many applications.

In addition to the types of failure, the maximum number of failures that can be tolerated is also vital. If all disks can be corrupted at one time (if for example a file's label is changed on all servers) there is no way to know, within the system, what the actual label should be. If f disks can be arbitrarily corrupted at any time, there must be at least $f + 1$ uncorrupted disks available at any time to make it possible, using majority voting, for the FTRM system to provide the file service and also enforce the access control policy. More specifically, suppose out of a total of t servers, there can be u unavailable servers (due to server machine crash, disk crash, or the crash of the link between a disk and its server machine or between a server site and a user site), f available servers with corrupted disks, and w available servers with uncorrupted disks. For the system to correctly enforce security in the presence of these faults, it must be the case that $t = u + f + w$ and that $w > f$ holds at all times.

The total number of available servers is $f + w$. If f is a fixed maximum number of corruption failures that the system is expected to be able to tolerate, the FTRM will grant access (i.e., operate) only when $f + w > 2f$, or $w > f$, so that the voting algorithm can always yield correct answers. It is important to note that more crash failures (i.e., when $w < f + 1$) will only render the file service unavailable, but the access control policy will not be violated.

7.2.3 Data Model and Access Control Policy

The objects the FTRM is concerned with are clients and files. Each copy of a file F is assigned a security level $level(F)$ from some collection of levels specified by the application. We assume that a user may log in and use the system at various levels in the same collection, reflecting the fact that users can act in different capacities at different times. Such a user must specify, during the login procedure, at which level he will act. The levels a particular user can log in at is decided by a policy external to the FTRM. A client (a process running on behalf of a user) inherits the security level at which the user is logged in. Thus each client C is assigned a clearance level $level(C)$.

There is a *dominates* relation between security levels, denoted by $\geq$, which is transitive and reflexive: $a \geq a$; $a \geq b$ and $b \geq c$ implies $a \geq c$. Equality is denoted by $a = b$. The access control policy of the FTRM is based on the widely applied Bell-LaPadula style multilevel access control policy [4, 3]:

- A client C may read from a file F only if $level(C) \geq level(F)$.

- A client C may write to a file F only if $level(F) \geq level(C)$.

To illustrate, suppose the security levels are the standard four classifications: unclassified, confidential, secret, and top-secret. The $\geq$ relation could be defined as

(1) confidential $\geq$ unclassified
(2) secret $\geq$ confidential
(3) top-secret $\geq$ secret
(4) $a \geq a$
(5) $a \geq b$ and $b \geq c$ implies $a \geq c$

7.2.4 File System and Access Control Policy

The file system has a typical UNIX hierarchical organization. The starting reference point of the file system is a root directory called /. Within a directory, there can be subdirectories and files. A directory is treated as a file with a certain kind of structure. A directory contains a set of entries, each representing a subdirectory or a file, and is of the form

(name, owner, access mode, size, date-of-last-modification)

There are three types of access, *read*, *write*, and *execute*. An access mode describes some combination of the three. Writing to a directory amounts to the creation or deletion of subdirectories or files.

To instantiate the multilevel access control policy described earlier, the following access rules are enforced. Let C denote a client, D a directory, and F a subdirectory or a file in directory D.

1. $level(/)$ = system low (e.g., unclassified).
2. $level(F) \geq level(D)$.
3. C can create F if and only if $level(F) \geq level(C) = level(D)$.
4. C has read access to F if and only if $level(C) \geq level(F)$.
5. C has write access to F if and only if $level(F) \geq level(C) = level(D)$.

It is straightforward to check that the rules enforce the access control policy because they *are* the access control policy. Note that from rules 2 and 4, C has read access to a directory (whose level is dominated by $level(C)$) that can contain a file or directory F whose level dominates $level(C)$. Thus C will know the existence of F even if $level(F) \geq level(C)$. This knowledge is not a security violation since the existence of a file is information that resides in the directory rather than in the file itself, and by rule 3, directories can only be written by clients of lower or equal level. This access control policy blocks covert channels through creating and deleting high-level files. In addition, by using this access control policy, the FTRM is not forced to use the complicated polyinstantiation strategies widely used in multilevel secure database systems [10, 13].

7.2.5 Synchronizing Replicated FTRM Server Monitors

The file system primitives can tolerate failures (host crashes, data corruption) up to a certain threshold number by broadcasting each request to a group of server monitors so that as long as a sufficient number of servers are available, the service is available. Majority voting is used on the returned result to mask failures, particularly data corruption.

To achieve this, it is insufficient just to hook up a number of the replicated FTRM server monitors and let them operate concurrently. They must coordinate among themselves to maintain data consistency and to behave like a single unreplicated file system.

To illustrate the necessity of coordination, consider the following scenario. Suppose the file system contains a file X, whose contents are initially the string "a". Suppose three different users issue three calls: the first writes the string "b" into the file, the second writes the string "c" into the file, and the third reads the first byte of the file. Now a server monitor that receives the calls in the order 1-2-3, will return value "c" upon read. A server monitor that gets the calls in the order of 1-3-2 will return value "b". Another server monitor that receives in the order of 3-1-2 will return value "a". If there are a total of three server monitors, then no majority value can be decided and the service is treated as unavailable. If the second server monitor receives in the same order as the third server monitor, then the majority value is the wrong value, "a". Therefore, it is possible for the FTRM to be in a situation where the file service is unavailable, or worse, the return value is wrong, even when no failure at the servers has occurred.

Such things can happen because in a typical network (such as the Internet) messages are not guaranteed to be delivered at all, or if delivered, are not guaranteed to be delivered in the order in which they were sent. Therefore, the FTRM system can behave correctly only if the following additional requirement is met: messages sent to the group of replicated server monitors will be delivered and processed at all operational servers in the same order. (Note that it is *not* necessary that this order be the same order in which they were sent; it is only necessary that it be the same order at each server).

Meeting this requirement is a nontrivial task, and this task and related issues are ongoing research topics in the fields of distributed systems and databases. Fortunately, there is a toolkit for distributed computing, called ISIS, developed at Cornell University, that makes this effort much easier. Details of ISIS are introduced in section 7.2.6. For the discussion here, it is sufficient to know that if the broadcast call in the FTRM uses the primitive ISIS call `bcast()`, then ISIS will take care of the message ordering in a fault tolerant and transparent manner.

7.2.6 ISIS

ISIS is a group-oriented distributed computing system that provides several forms of support for fault-tolerant computing. The development of ISIS is led by Professor Kenneth Birman at Cornell University. The latest version of ISIS (V3.0) is a commercial product release. It currently runs on a number of platforms including Sun OS (UNIX). Major efforts are underway to port ISIS to other platforms, including the Mach operating system developed at Carnegie-Mellon University. Commercial versions of ISIS are marketed by ISIS Distributed Systems, Inc., while academic versions are free of charge.

ISIS can be used in a network of computers linked by either local area or wide area networks. Initial performance measurements on prototypes show that ISIS is an efficient mechanism to support fault tolerant computing.

ISIS Abstractions and Guarantees

At the abstract level, ISIS transforms nondistributed abstract type specifications into fault tolerant, distributed implementations, called *resilient objects*. Such objects achieve fault tolerance by replicating the code and data managed by the object at more than one site. The resulting components synchronize their actions to provide the effect of a single site object. In the event of a failure, any ongoing operation at a failed component is continued by an operational one. A resilient object continues to accept and process new operations as long as at least one component is operational. Failed components recover automatically when the site at which they reside is restarted.

At the system level, ISIS supports an abstraction mechanism called *fault tolerant process groups*. ISIS guarantees that all processes belonging to the same group will observe consistent orderings of events affecting the group as a whole, including process failures, recoveries, migration, and dynamic changes to group properties such as group membership and member rankings. These orderings are achieved through site managers running a set of protocols such as agreement and failure detection.

At the communication layer, ISIS has a number of primitives to support reliable communication even in the presence of failures. They include group broadcast, atomic broadcast, causal broadcast, and minimal broadcast. Each of these primitives supports a slightly different broadcast semantics. For example, atomic broadcast guarantees that, if a message to a process group is delivered at one member site, it is delivered to all member sites. Also, messages are processed in the same order at all sites.

ISIS supports many other features, such as causality across the boundary of groups.

7.3 Formal Security Theory

The access control policy stated above *seems* to capture the basic security requirements of the FTRM and even addresses some subtleties. The most obvious security threat is the threat of a user or process with a low security level being able to read a file of high security level. This threat is addressed in a straightforward way by the mediation logic in the FTRM open call, which does not allow a client to open a file unless the security level of the client dominates that of the file.

A more subtle (albeit long-recognized) security threat is that of a so-called "Trojan horse" client that has a high security level inherited from a trusted user, but that contains untrustworthy code.[1] One of the ways such a client can compromise security is by reading high-level data (which it is permitted to do by virtue of its high security level), and then, unbeknownst to the user it belongs to, writing that data into a file with a low security level, where it may later be read by a low-level user who is not trusted. This threat is addressed by the access control policy using the standard technique described in [4] and [3]. The mediation logic in the `ft_open` call does not allow a client to open a file for writing unless the security level of the client *is dominated by* the security level of the file. This means that a high-level client, even if it is a Trojan horse, can only write to high-level files, and so cannot compromise security by a "write down" to a low-level file.

An even more subtle security threat is that of a Trojan horse client signaling information to a low-level client by creating files. For example, a high-level Trojan horse client might signal high-level information to a low-level client by creating a low-level file that the low-level client could then see. The information might be encoded in the name of the file. The access control policy blocks this threat by giving newly created files the level of the creating client, so high-level clients can only create high-level files. Another version of the same threat, which is *not* blocked by this provision, is a high-level Trojan horse client creating a *high* level file whose existence could be detected by a low-level client by the low level client attempting to open the file for "write". This threat is blocked by the way the access control policy treats the level of directories. Since a directory is a file like any other, and file creation is a write to the directory in which the file is created, a high-level Trojan horse client can only write to high-level directories. Thus, a high-level client can only create a file in a high-level directory. Checking whether a file exists in a directory is a read from the directory, so a low-level client cannot check whether a file exists in a high-level directory. Any attempt to do so results in an error return, regardless of whether the file exists or not. Thus, a low-level

[1]The untrustworthy code may be deliberately, maliciously introduced, or it may just be a bug, or even a feature of the application. For example, the emacs editor creates "checkpoint" files containing the latest state of its internal text buffers in case the editor is crashed without saving the buffers. In certain cases, the checkpoint files can have more liberal permissions than the file being edited.

client cannot see anything about files that any high-level client creates.

As the previous paragraph demonstrates, making sure that a system blocks all the ways that a Trojan horse client can compromise security involves considering very subtle ways to transmit information. Security can be compromised by so-called *covert channels* as well as by explicit access to data. The threats described above having to do with transmitting information by creating files are examples of covert channels that are blocked by the access control policy.

How do we know that there are not other covert channels that are not blocked by the access control policy? For example, the FTRM will contain replicated, distributed data. As mentioned in section 7.2.5, we must take certain measures to ensure that this distributed data is managed consistently. Our approach is to use the ISIS network primitives. Suppose, however, that we had instead used some form of a simple locking scheme. In other words, suppose that when a client makes an FTRM call, the user monitor sent out messages throughout the network telling the server monitors to "lock" all the data that the FTRM call needs to access, performed the operations, and then "unlocked" all of the data it had been using. Suppose further that if another FTRM call attempted to lock the same data, it received an error return, and then issued an error (e.g., "Cannot access due to data lock") to the client. This approach could be used to solve the problem of maintaining data consistency, but it would introduce a covert channel because high-level clients could signal low-level clients by issuing calls that lock low-level files. low-level clients could detect these locks by attempting to access the same files and receiving the error message. If we had chosen this design instead of using ISIS, the FTRM would contain a covert channel. This covert channel would be part of the FTRM, but would not be blocked by the access control policy because it uses information transmission mechanisms that are not taken into account by the access control policy (namely the error messages returned by FTRM calls).

Of course, we can never be completely sure that we have not overlooked some way of transmitting information. We can, however, gain increased confidence that we have not overlooked subtle channels by using a *formal security theory*. A formal security theory is a mathematical definition of security in terms of some form of information theory. In order to apply such a theory, we must first describe the system as a mathematical model, that is, we must represent the system as some kind of mathematical object that can be analyzed using mathematical tools and techniques. A formal security theory is a formal definition in terms of such mathematical models that says which models represent secure systems and which do not. Once we have written down a mathematical model of the system whose security we are interested in (in our case, the FTRM), we can analyze it using mathematics to determine whether it meets the formal definition of security.

The reason this sort of analysis gives more confidence that the system in question is secure is that the definition of which models represent secure systems is based on general information theory. As we will see below, our definition of security is based on representing the system as an automaton that interacts with its environment through accepting *inputs* and generating *outputs*. Information theory is used to define when some class of inputs can be used to transmit information through some other class of outputs. This definition is not specific to a system, but applies to any system represented as such an automaton. The definition of security does not assume that information is transferred only through file accesses, but considers any information transfer from any class of inputs to any class of outputs.

For these reasons, we use the formal theory of *restrictiveness*, and the specialization of restrictiveness to buffered server processes, as the basis for formally analyzing the FTRM model's security. A server process is one that waits for input with set values of its state parameters, processes each input possibly producing output, and then returns to wait for the next input with possibly changed values of its state parameters. A buffered process saves its inputs, if necessary until it is ready to receive them, so that it never reveals any information by its ability or inability to accept input. Informally, a buffered server process is restrictive if its outputs in response to any input are at levels that dominate the level of the input, if the part of its state-parameter information needed to give its future behavior visible at a security level does not change in response to inputs not visible at this level, and if two partial state parameter descriptions giving the same future behavior visible at a security level are not distinguished by any output visible at that level. Complete descriptions of restrictiveness can be found in the literature on information security, in particular in [14] and [15].

In the remainder of this section, we discuss how we used restrictiveness to specify the fault tolerance of the FTRM as well as its security. Our initial plans were to formally define the security property that the FTRM is required to obey and to represent the fault tolerance mechanisms of the FTRM in the model, but not to formally define what it meant for the FTRM to be fault tolerant. It later became clear that we could define security for the FTRM in such a way that it would imply fault tolerance.

As will be explained in detail in section 7.4 below, we modeled data corruption faults in the FTRM as if they were inputs from the external environment. In this way, we made sure that data corruption faults could occur in a completely arbitrary pattern that the FTRM could not control. A side effect of this way of representing faults is that we had to assign a security level to the inputs that represent faults. The usual level to assign to such "artificial" inputs is `unclassified`, since they carry no classified information. We need not assign such inputs the level `unclassified` however. Since they are merely artifacts of the modeling process, we are free to assign them any security level that will allow the model to satisfy the restrictiveness property. The only inputs and outputs whose levels we must make sure are correct are those that come from or go to real external entities with real security classifications.

In considering what security level to assign to the "fault inputs", we realized that if we assigned them the security level `systemhigh` (that is, the security level that dominates all other security levels), then in order for the FTRM model to satisfy restrictiveness, it would have to be impossible to infer anything about the occurrence of fault inputs from observing the system behavior at any level lower than `systemhigh`. In fact, the FTRM model does not make any assumptions on what levels the clients have, so as far as the constraints placed on the system by the FTRM model, it is possible that none of the clients have level `systemhigh`. Since this situation is permitted by the model, in order for the model to satisfy restrictiveness, it must be the case that the clients cannot infer anything about the occurrence of fault inputs. In particular, they cannot be able to infer that any faults have occurred. In other words, if we assign the fault inputs level `systemhigh`, then restrictiveness implies that the system must be fault tolerant, because it implies that the clients cannot tell from observing their inputs and outputs whether any faults have occurred.

7.4 Formal Model

This section contains and describes the SML code formally specifying the FTRM model for HOL90.

7.4.1 Front Matter

The specification begins with lines removing any older versions of the FTRM theory, setting the theory path to include the library of Romulus process and security theories, and creating the FTRM theory, a child of the Romulus security theory. These lines also load the HOL library dealing with character strings and the SML code for Romulus convenience functions **romcontype** and **romrecord** creating concrete-recursive and record types.

The Romulus utility **romcontype** is very similar to the standard HOL function **define_type** for defining concrete-recursive types, but has a slightly simpler interface in the case where all type constructors for the type being defined are prefix operators, avoids a bug in the current implementation of HOL90 that makes it impossible to define type constructors that take tuples as arguments, and returns both an SML variable whose value is the type just defined and a theorem giving an abstract characterization of this type. The SML variable giving the type is useful in the case where the type defined is actually a polymorphic type constructor.

The Romulus utility **romrecord** is superficially similar to **romcontype**, and indeed does define a record as a concrete-recursive type. However, it also defines a constructor function for creating records of the type being defined and defines accessor and update functions for each entry in the record. The constructor function defined by **romrecord** for a record type is named by prepending Make_ to the name of the record type. The accessor and update functions defined by **romrecord** for a record entry are named after the name of the entry or by prepending **update_** to the name of the entry, respectively.

```
System.system "rm -f ftrm.holsig ftrm.thms";

new_theory "ftrm";
load_library{lib = get_library_from_disk "romulus",
             theory = "-"};
load_library{lib = Sys_lib.string_lib, theory = "-"};
```

7.4.2 Type Declarations

The specification next declares all the types used in the specification of the FTRM process, culminating in the definition of the type of PSL object appropriate as a model of the FTRM process. The types defined include the state parameters, the input events, and the output events. For maximum generality, the specification uses type variables instead of explicit types wherever possible, so the types it declares are actually polymorphic type constructors. The type variables `'Server`, `'Client`, and `'Level` denote the arbitrary types of server identifiers, client processes, and security levels.

State Parameters

The specification first declares the type of the parameter giving the state of the FTRM process as it waits to receive the next request from a client. This type will later be used in the declarations of functions not defined in the FTRM model, in the definitions of FTRM invocations, and in defining several FTRM-specific functions. It makes several preliminary definitions in working up to the definition of state parameters.

The model assumes that a file contains a list of bytes, that reading a file returns the full list of bytes in this file, and that writing a file replaces the full list of bytes in this file with a new list of bytes.

A *status* is a return value, typically implemented as an integer, telling whether or not a command succeeded. The model needs to know only that "success" and "failure" are both status values and that they are distinct.

```
val (Status_Def, Status) =
 romcontype
  "Status"
  [("success", []), ("failure", [])];
```

Similarly, an *access* is a form of access for a client process to a file.

```
val (Access_Def, Access) =
 romcontype
  "Access"
  [("none", []), ("read", []), ("write", []), ("read_write", [])];
```

The model takes a file name to be a character string. A *file address* is an abstract address of a disk data structure giving a file's name, whether the file is currently open, and its current contents. In the model, it is taken to be an integer. A *file descriptor* is a pair consisting of a file address and a file-access type. In an actual implementation, it could be an index into a system table containing this information, but we do not model this detail because we do not model the possibility that the information in tables indexed by file descriptors can be faulty for a functioning server.

```
val file_name = ty_antiq(==`:string`==);

val file_address = ty_antiq(==`:num`==);

val (file_descriptor_Def, file_descriptor) =
 romrecord
  "file_descriptor"
  [
   ("descriptor_address", ==`:^file_address`==),
   ("descriptor_access",  ==`:^Access`==)
  ];
```

The definition of addressed_data models the possibility that a server's disk might not contain a plausible data structure at a file address, either because the address does not correspond to a disk location or because the data there is not in the expected form. The entry corrupted is true if the data does not really exist or is not of the expected form. In this case, the other entries in the record are defined, but meaningless, and are never used.

```
val (addressed_data_Def, addressed_data) =
 romrecord
  "addressed_data"
  [
   ("corrupted", ==`:bool`==),
   ("hasname",   ==`:^file_name`==),
   ("isopen",    ==`:bool`==),
   ("contents",  ==`:(num)list`==)
  ];
```

The model defines a concrete recursive type corresponding to all the possible FTRM requests and the pieces of information sent with them, information such as which file to open and which access to obtain for it when it is open.

```
val files_data = ty_antiq(==`:^file_address -> ^addressed_data`==);
val (FTRM_request_Def, FTRM_request) =
 romcontype
  "FTRM_request"
  [
   ("request_close",   [==‘:^file_descriptor‘==]),
   ("request_open",    [==‘:^file_name‘==,==‘:^Access‘==]),
   ("request_read",    [==‘:^file_descriptor‘==]),
   ("request_write",   [==‘:^file_descriptor‘==, ==‘:(num)list‘==])
  ];
```

Similarly, the model defines a concrete recursive type for the sorts of information received in response to a FTRM request. This information always includes a status value telling whether the request succeeded, but it can also include additional information such as the file descriptor of the file opened or the contents of the file read.

```
val (FTRM_reply_Def, FTRM_reply) =
 romcontype
  "FTRM_reply"
  [
   ("reply_close",     [==‘:^Status‘==]),
   ("reply_open",      [==‘:^Status‘==,==‘:^file_descriptor‘==]),
   ("reply_read",      [==‘:^Status‘==,==‘:(num)list‘==]),
   ("reply_write",     [==‘:^Status‘==])
  ];
```

The model also defines a concrete recursive type for the possible ISIS requests generated by the FTRM in response to the FTRM requests from the client processes.

```
val (ISIS_request_Def, ISIS_request) =
 romcontype
  "ISIS_request"
  [
   ("isis_check",      [==‘:^file_name‘==]),
   ("isis_close",      [==‘:^file_descriptor‘==]),
   ("isis_open",       [==‘:^file_name‘==,==‘:^Access‘==]),
   ("isis_read",       [==‘:^file_descriptor‘==]),
   ("isis_write",      [==‘:^file_descriptor‘==, ==‘:(num)list‘==])
  ];
```

With the preliminary definitions made, the model then defines the type of the state parameter for the FTRM as a record containing the following information: the number of faulty servers; a function telling whether or not each server is faulty; a function giving the list of pending FTRM requests made by each client process; a function giving the list of pending ISIS requests made by the FTRM process in response to the pending FTRM requests for each client process; a function telling for each client process and file descriptor whether that client process currently has that file open; a function giving the contents of each file on each server; a function giving the security level of each client process as it is recorded on each server; and a function giving the security level of each file as it is recorded on each server. Some of the state-parameter information is abstract and impossible to implement, while other information is implemented but assumed not to be faulty in the model, and still other information is implemented and modeled as possibly being faulty.

```
val c_security_data = ty_antiq(==`:'Client -> 'Level`==);

val f_security_data = ty_antiq(==`:^file_name -> 'Level`==);

val (FTRM_State_Def, FTRM_State) =
 romrecord
  "FTRM_State"
  [
   (* unimplementable system data *)

   ("fault_number",      ==`:num`==),
   ("server_fault",      ==`:'Server -> bool`==),

   (* implemented, assumed fault-proof, system data *)

   ("ftrm_info",         ==`:'Client -> (^FTRM_request)list`==),
   ("isis_request_info", ==`:'Client-> (^ISIS_request)list`==),
   ("open_info",         ==`:'Client -> ^file_descriptor -> bool`==),

   (* implemented, modeled as possibly faulty, system data *)

   ("server_files",      ==`:'Server -> ^files_data`==),
   ("server_c_security", ==`:'Server -> ^c_security_data`==),
   ("server_f_security", ==`:'Server -> ^f_security_data`==)
  ];
```

Input Events

The specification next defines the input events to the FTRM model. These input events are themselves defined in terms of the concrete recursive type `FTRM_request`, which names the various requests to the FTRM from the client processes. An input event is defined as one of three type constructors, corresponding to abstract input ports conveying messages to the model. The constructors and the interpretations of their arguments follow:

- `ClientInput` conveys a request from a client process to the FTRM. Its arguments give the client process and that client's request.
- `Fault` describes a failure in a server. Its arguments identify the server that fails and give the new values of the files data, the client-process security-level data, and the file security-level data for the failed server.
- `Tick` describes a modeling artifact for showing the multiprocessing in the FTRM. Its argument identifies the client process whose pending requests will next be acted on by the FTRM.

```
val (FTRM_InEv_Def, FTRM_InEv) =
 romcontype
  "FTRM_InEv"
  [
   ("ClientInput", [==`:'Client`==,
                    ==`:^FTRM_request`==]),
   ("Fault",       [==`:'Server`==,
                    ==`:^files_data`==,
                    ==`:^c_security_data`==,
                    ==`:^f_security_data`==]),
   ("Tick",        [==`:'Client`==])
  ];
```

Output Events

The specification then defines the output events produced by the FTRM. These output events are themselves defined in terms of the concrete recursive type `FTRM_reply`, which gives the types of the information appropriate to the various client-process requests. An output event is defined as the type constructor `ClientOutput`, corresponding to a single abstract output port conveying information to the client processes. The arguments to `ClientOutput` identify the client to receive the information and the information this client is to receive.

```
val (FTRM_OutEv_Def, FTRM_OutEv) =
 romcontype
  "FTRM_OutEv"
  [("ClientOutput", [==`:'Client`==,==`:^FTRM_reply`==])];
```

Invocations

The specification next defines the invocations to be used later in defining the model of the FTRM process. Invocations are essentially names for calls to PSL-valued functions; they are mapped to the PSL processes resulting from these calls. Invocations provide a means for overcoming the limitation in HOL's `define_type` function that concrete recursive types cannot be defined in terms of functions whose values are of the type being defined.

The invocations name calls to the FTRM model process itself and to the function computing the model process' response to input events. All these calls include the FTRM state parameter as an argument.

```
val (FTRM_Invoc_Def, FTRM_Invoc) =
 romcontype
  "FTRM_Invoc"
  [
   ("FTRM",          [==`:^FTRM_State`==]),
   ("FTRM_Response", [==`:^FTRM_State`==, ==`:^FTRM_InEv`==])
  ];
```

PSL Processes

The invocations complete the definition of the type of PSL processes appropriate as models for the FTRM. The specification defines the SML variable `FTRM_Proc` as an abbreviation for this type.

```
val FTRM_Proc =
 ty_antiq(==`:(^FTRM_OutEv,^FTRM_InEv,^FTRM_Invoc) process`==);
```

7.4.3 Functions Taken as Primitive

The specification then declares HOL constants for the constants or functions taken as primitive in the FTRM model. These functions involve details of the implementation that the model excludes. For convenience in future reference, the specification also defines SML variables giving each polymorphic function a (possibly variable) type.

Function `client_ext_level` assigns each client its true security level, independent of any system faults.

```
new_constant{Name= "client_ext_level", Ty= ==`:'Client -> 'Level`==};

val client_ext_level = --`client_ext_level:'Client -> 'Level`--;
```

Function `dom` is the dominance relation on security levels.

```
new_constant{Name = "dom", Ty = ==`:'Level -> 'Level -> bool`==};

val dom = --`dom:'Level -> 'Level -> bool`--;
```

Function `file_ext_level` assigns each file its true security level, independent of any system faults.

```
new_constant{Name="file_ext_level",Ty= ==`:^file_name -> 'Level`==};

val file_ext_level = --`file_ext_level:^file_name -> 'Level`--;
```

Constant `initial_file_system` gives the initial state of the file system for all servers; they all start out the same.

```
new_constant{Name = "initial_file_system",Ty= ==`:^files_data`==};

val initial_file_system = --`initial_file_system:^files_data`--;
```

Constant `maxfaults` is the maximum number of faults the FTRM is assumed capable of handling in the model; the model ignores any additional faults.

```
new_constant{Name="maxfaults",Ty= ==`:num`==};
```

Function `return_close` returns a status value for closing a file given a file descriptor and files data.

```
new_constant{
 Name = "return_close",
 Ty= ==`:^file_descriptor -> ^files_data -> ^Status`==};

val return_close =
 --`return_close:
    ^file_descriptor -> ^files_data -> ^Status`--;
```

Function `return_open` returns a status value and a file descriptor for opening a file given a file name, access type, and files data.

```
new_constant{
 Name = "return_open",
 Ty= ==`:^file_name ->
         ^Access ->
         ^files_data ->
         (^Status # ^file_descriptor)`==};

val return_open =
 --`return_open:
     ^file_name ->
     ^Access ->
     ^files_data ->
     (^Status # ^file_descriptor)`--;
```

Function `return_read` returns a status value and a list of bytes taken as the current contents of the file given a file descriptor and files data.

```
new_constant{
 Name = "return_read",
 Ty= ==`:^file_descriptor ->
         ^files_data ->
         (^Status # (num)list)`==};

val return_read =
 --`return_read:
     ^file_descriptor ->
     ^files_data ->
     (^Status # (num)list)`--;
```

Function `return_write` returns a status value for writing a file given a file descriptor, list of bytes taken as the new contents of the file, and before-write files data.

```
new_constant{
 Name = "return_write",
 Ty= ==`:^file_descriptor -> (num)list -> ^files_data -> ^Status`==};

val return_write =
 --`return_write:
     ^file_descriptor -> (num)list -> ^files_data -> ^Status`--;
```

Constant `systemhigh` is the highest security level.

```
new_constant{Name = "systemhigh", Ty = ==`:'Level`==};

val systemhigh = --`systemhigh:'Level`--;
```

Constant `systemlow` is the lowest security level.

```
new_constant{Name = "systemlow", Ty = ==`:'Level`==};

val systemlow = --`systemlow:'Level`--;
```

Function `vote_file_descriptor` takes a function mapping servers to file descriptor values and returns the value this function assigns to most of the servers.

```
new_constant{
 Name = "vote_file_descriptor",
 Ty = ==`:('Server -> ^file_descriptor) -> ^file_descriptor`==};

val vote_file_descriptor =
 --`vote_file_descriptor:
     ('Server -> ^file_descriptor) -> ^file_descriptor`--;
```

Function `vote_level` takes a function mapping servers to levels and returns the level this function assigns to most of the servers.

```
new_constant{
 Name = "vote_level",
 Ty = ==`:('Server -> 'Level) -> 'Level`==};

val vote_level = --`vote_level: ('Server -> 'Level) -> 'Level`--;
```

Function `vote_num_list` takes a function mapping servers to lists of bytes and returns the list this function assigns to most of the servers.

```
new_constant{
 Name = "vote_num_list",
 Ty = ==`:('Server -> (num)list) -> (num)list`==};

val vote_num_list =
 --`vote_num_list: ('Server -> (num)list) -> (num)list`--;
```

Function vote_status takes a function mapping servers to Status values and returns the value this function assigns to most of the servers.

```
new_constant{
 Name = "vote_status",
 Ty = ==`:('Server -> ^Status) -> ^Status`==};

val vote_status = --`vote_status:('Server -> ^Status) -> ^Status`--;
```

7.4.4 Assumptions for Primitive Functions

The specification next states assumed properties of the constants and functions taken as primitive in the FTRM model.

The first several assumptions simply assert obvious defining properties of the lowest and highest security levels and the dominance relation on security levels.

```
new_open_axiom(
 "systemlow_low",
 --`!l:'Level. dom l systemlow`--);

new_open_axiom(
 "systemhigh_high",
 --`!l:'Level. dom systemhigh l`--);

new_open_axiom(
 "dom_reflexive",
 --`!l:'Level. dom l l`--);

new_open_axiom(
 "dom_transitive",
 --`!(l1:'Level) l2 l3.
     ((dom l1 l2) /\ (dom l2 l3)) ==> (dom l1 l3)`--);
```

The next assumption asserts of the initial file system that for all file addresses, every address is either not the address of a valid file (modeled by saying that the file at that address is "corrupted") or that the file is not open.

```
new_open_axiom(
 "initially_all_files_closed",
 --`!fa.
     (corrupted (initial_file_system fa)) \/
     ~(isopen (initial_file_system fa))`--);
```

The next assumptions say the `return_close`, `return_open`, `return_read`, and `return_write` functions work as expected:

- If `return_close` returns success for attempting to close a file given by a file descriptor, then the access in that descriptor is not `none`, the file address in the descriptor is a valid file address, and the file addressed was open when `return_close` was called.

- If `return_open` returns success and a file descriptor, the access in the file descriptor is the access requested, the address in the file descriptor is a valid file address, the address is the address of the file requested, and the file requested was not already open when `return_open` was called.

- If `return_read` returns success and a file contents for a file descriptor, the access in that descriptor was for reading or reading and writing, the address in the descriptor is a valid address of an open file, and the file contents returned are the contents of this file.

- If `return_write` returns success for a file descriptor and new file contents, the access in that descriptor was for writing or reading and writing, and the address in the descriptor is a valid address of an open file. (The actual contents of the file are changed later, after `return_write` completes.)

```
new_open_axiom(
 "return_close_valid",
 --`!fd files.
     (^return_close fd files = success) ==>
     (~(descriptor_access fd = none) /\
      ~(corrupted (files (descriptor_address fd))) /\
      (isopen (files (descriptor_address fd))))`--);
```

```
new_open_axiom(
 "return_open_valid",
 --`!fname access files fd.
     (^return_open fname access files = (success,fd)) ==>
     ((descriptor_access fd = access) /\
      ~(corrupted (files (descriptor_address fd))) /\
      ((hasname (files (descriptor_address fd))) = fname) /\
      ~(isopen (files (descriptor_address fd))))`--);

new_open_axiom(
 "return_read_valid",
 --`!fd files nlist.
     (^return_read fd files = (success,nlist)) ==>
     (((descriptor_access fd = read) \/
       (descriptor_access fd = read_write)) /\
      ~(corrupted (files (descriptor_address fd))) /\
      (isopen (files (descriptor_address fd))) /\
      (nlist = contents (files (descriptor_address fd))))`--);

new_open_axiom(
 "return_write_valid",
 --`!fd nlist files.
     (^return_write fd nlist files = success) ==>
     (((descriptor_access fd = write) \/
       (descriptor_access fd = read_write)) /\
      ~(corrupted (files (descriptor_address fd))) /\
      (isopen (files (descriptor_address fd))))`--);
```

The final assumption says that all the voting functions work correctly, and if a status, file descriptor, file contents, or security level are voted on after querying all servers — when the total number of failed servers is less than `maxfaults` — then the voted value will be the value returned by an unfailed server.

```
new_open_axiom(
 "vote_results_valid",
 --`!(state:^FTRM_State) f c fd ac nlist.

     let goodserver = (@gs. ~(server_fault state gs)) in

     (fault_number state <= maxfaults) ==>
(
 (vote_status
   (\server. ~return_close fd (server_files state server)) =
  (~return_close fd (server_files state goodserver))) /\

 (vote_status
   (\server.
      FST (~return_open f ac (server_files state server))) =
  FST (~return_open f ac (server_files state goodserver))) /\

 (vote_status
   (\server.
      FST (~return_read fd (server_files state server))) =
  FST (~return_read fd (server_files state goodserver))) /\

 (vote_status
   (\server.
      ~return_write fd nlist (server_files state server)) =
  (~return_write fd nlist (server_files state goodserver))) /\

 (vote_file_descriptor
   (\server.
      SND (~return_open f ac (server_files state server))) =
  SND (~return_open f ac (server_files state goodserver))) /\

 (vote_num_list
   (\server.
      SND (~return_read fd (server_files state server))) =
  SND (~return_read fd (server_files state goodserver))) /\

 (vote_level (\server. server_c_security state server c) =
  server_c_security state goodserver c) /\

 (vote_level (\server. server_f_security state server f) =
  server_f_security state goodserver f)

)‘--);
```

7.4.5 Security-Level Assignments

The specification then defines the functions assigning security levels to input and output events. Intuitively, the level of an input event from, or an output event to, a client process is the level of the client process sending or receiving the event. A fault event is given level systemhigh, for reasons previously explained. A Tick event advancing the state of a client process' requests is given the level of that client process.

The specification uses new_recursive_definition, which defines functions on concrete recursive types. One of its arguments, rec_axiom, is the theorem giving an abstract characterization of the concrete recursive type over which the function is being defined.

```
new_recursive_definition {
 name = "FTRM_InLevel",
 fixity = Prefix,
 rec_axiom = FTRM_InEv_Def,
 def =
  let
   val FTRM_InLevel =
    --`FTRM_InLevel:^FTRM_State -> ^FTRM_InEv -> 'Level`--;
  in
   --`
   (^FTRM_InLevel state (ClientInput c request) =
      (^client_ext_level c)) /\

   (^FTRM_InLevel state (Fault serv fdata csdata fsdata) =
      ^systemhigh) /\

   (^FTRM_InLevel state (Tick c) =
      (^client_ext_level c))
   `--
  end};

new_recursive_definition {
 name = "FTRM_OutLevel",
 fixity = Prefix,
 rec_axiom = FTRM_OutEv_Def,
 def =
  let
   val FTRM_OutLevel =
    --`FTRM_OutLevel:^FTRM_State -> ^FTRM_OutEv -> 'Level`--;
  in
   --`
    (^FTRM_OutLevel state (ClientOutput c reply) =
      (^client_ext_level c))
   `--
  end};
```

7.4.6 Projection Function

The specification then defines the projection function, a function of a security level and an FTRM state parameter. The projection's value at a level and a state parameter is the state parameter containing all, but only, the information necessary to produce the FTRM's future behavior visible at that level. Intuitively, for every level except systemhigh, the projection removes any evidence of failures and any client process or file whose level is not dominated by that level; for any level except systemhigh, the number of faults is 0, no server is faulty, no FTRM or ISIS requests are pending from any client whose level is not dominated by that level, every existing file has a level dominated by that level and has the same contents on all servers that it has on a non-faulty server, and level information for all existing files and client processes is correct on all servers.

The specification uses several of the record-entry access functions defined by earlier calls to romrecord.

```
new_definition(
 "FTRM_Projection",
 let
  val FTRM_Projection =
   --`FTRM_Projection:'Level -> ^FTRM_State -> ^FTRM_State`--;
  val server = --`server:'Server`--;
 in
  --`
   ^FTRM_Projection level state =

   ((level = systemhigh) =>
    state
   |
    (let proj_fault_number = 0 in

     let proj_server_fault ^server = F in

     let proj_ftrm_info c =
      (dom level (client_ext_level c)) =>
       (ftrm_info state c)
      |
       [] in

     let proj_isis_request_info c =
      (dom level (client_ext_level c)) =>
  (isis_request_info state c)
 |
  [] in

let proj_open_info c fd =
 (dom level (client_ext_level c)) =>
  (open_info state c fd)
 |
  F in

let proj_server_files ^server fa =
 (~(corrupted (server_files state server fa)) /\
  (dom
    level
    (file_ext_level
      (hasname (server_files state server fa))))) =>
   (server_files
     state
     (@goodserver. ~(server_fault state goodserver))
     fa)
  |
   (update_corrupted T (server_files state server fa)) in

let proj_server_c_security ^server c =
 (dom level (^client_ext_level c)) =>
   (client_ext_level c)
  |
   ^systemlow in

let proj_server_f_security ^server f =
 (dom level (file_ext_level f)) =>
   (file_ext_level f)
  |
   ^systemhigh in

Make_FTRM_State
 proj_fault_number
 proj_server_fault
 proj_ftrm_info
 proj_isis_request_info
 proj_open_info
 proj_server_files
 proj_server_c_security
 proj_server_f_security))
```

```
  '--
 end);
```

7.4.7 Invariant

The specification next defines the invariant, a predicate to be shown by induction to be true of every state parameter attained by the FTRM model process. The invariant serves as an optional, useful induction hypothesis about the FTRM state parameter.

Intuitively, the invariant says the following: the total number of faulty servers is no more than `maxfaults`; the file-contents, client-security, and file-security information on all non-faulty servers is the same; and that for all servers, if a client process has a file open and that file has not been corrupted by a fault then 1) if the file is open for read access, the level of the server process dominates the level of the file, 2) if the file is open for write access, the level of the file dominates the level of the server process, and 3) if the file is open for read and write access, the levels of the server process and file are equal. Note that maintaining this invariant requires guaranteeing that faults cannot rename files or shift files so that a file descriptor for one file becomes the file descriptor for a different file. This condition is enforced by the specification later in its handling of fault events.

```
new_definition(
 "FTRM_Invariant",
 let
  val FTRM_Invariant = --'FTRM_Invariant:^FTRM_State -> bool'--;
 in
  --'
   ^FTRM_Invariant state =

     ((fault_number state) <= maxfaults) /\

     (!s1 s2 fa f c.
       (~(server_fault state s1) /\
        ~(server_fault state s2)) ==>
       ((server_files state s1 fa =
         server_files state s2 fa) /\
        (server_c_security state s1 c =
         server_c_security state s2 c) /\
        (server_f_security state s1 f =
            server_f_security state s2 f))) /\

        (!s c f fd.
          ((open_info state c fd) /\
           ~(corrupted
              ((server_files state s) (descriptor_address fd))) /\
           (f = (hasname
                 ((server_files state s) (descriptor_address fd))))) ==>
          (~(descriptor_access fd = none) /\
           ((descriptor_access fd = read) =>
             (~dom (client_ext_level c) (file_ext_level f))
           |
            (descriptor_access fd = write) =>
             (~dom (file_ext_level f) (client_ext_level c))
           |
            ((~client_ext_level c) = (~file_ext_level f)))))
    '--
    end);
```

7.4.8 Initial State Parameter

The specification next defines the initial value of the FTRM state parameter: no faults, no pending FTRM requests, no pending ISIS requests, no files open for any client, files on all servers as in `initial_file_system`, and all security data on all servers correct.

```
  new_definition(
   "FTRM_InitParam",
   let
    val FTRM_InitParam = --`FTRM_InitParam:^FTRM_State`--;
   in
    --`
     ^FTRM_InitParam =

       Make_FTRM_State
        0
        (\s. F)
        (\c. [])
        (\c. [])
        (\c fd. F)
        (\s. initial_file_system)
        (\s c. client_ext_level c)
        (\s f. file_ext_level f)
  '--
 end);
```

7.4.9 Invocation Interpretations

The specification then gives the core of the model, the interpretations of the invocations of the FTRM server process and the function giving the model's response to input events. The function FTRM_InvocVal, defined last, asserts that functions ftrm and ftrmResponse are called from PSL processes via the invocations FTRM and FTRM_Response, respectively. The specification works up to the definition of FTRM_InvocVal, first defining the ftrmResponse subroutines doftrmrequest and doisisrequest, which define the processing of client FTRM requests and pending ISIS requests, respectively. The specification then defines ftrmResponse and ftrm, and finally defines FTRM_InvocVal. The functions doftrmrequest and doisisrequest contain most of the details of the model of the FTRM process.

Function doftrmrequest

The function doftrmrequest gives the FTRM response to each possible request from a client process. It is defined using new_recursive_definition, over the concrete-recursive type FTRM_request.

The initial lines of the doftrmrequest definition name the function, provide the appropriate theorem giving an abstract characterization of the function's domain, and define a local SML variable that gives type information for the polymorphic function being defined.

```
new_recursive_definition {
 name = "doftrmrequest",
 fixity = Prefix,
 rec_axiom = FTRM_request_Def,
 def =
  let
   val doftrmrequest =
    --'doftrmrequest:
        ^FTRM_State -> 'Client -> ^FTRM_request -> ^FTRM_Proc'--;
  in
   --'
```

For an FTRM close request, the FTRM sends an appropriate failure notification and has no further effect if the request attempts to close a file that the requesting client does not have open. Otherwise, the FTRM puts the request on the end of the queue of the client process' pending FTRM requests and puts an ISIS close request on the end of the queue of its pending ISIS requests.

```
(^doftrmrequest state client (request_close fd) =
   (If
     (~(open_info state client fd))
     ((Send (ClientOutput client (reply_close failure))) ;;
      (Call (FTRM state)))
     (Call
       (FTRM
         (let new_ftrm_info c =
           (c = client) =>
              (SNOC (request_close fd) (ftrm_info state c))
           |
            (ftrm_info state c) in

          let new_isis_request_info c =
           (c = client) =>
              (SNOC (isis_close fd) (isis_request_info state c))
           |
            (isis_request_info state c) in

        (update_ftrm_info new_ftrm_info
        (update_isis_request_info new_isis_request_info
          state))))))) /\
```

For an FTRM open request, the FTRM puts the request on the end of the queue of the client process' pending FTRM requests and puts an ISIS check request on the end of the queue of its pending ISIS requests. The ISIS check request is made first, to query all servers about the security levels of the client and the requested file before allowing the open request to proceed.

```
(^doftrmrequest state client (request_open fname ac) =
   (Call
     (FTRM
       (let new_ftrm_info c =
         (c = client) =>
           (SNOC (request_open fname ac) (ftrm_info state c))
```

```
          |
           (ftrm_info state c) in

          let new_isis_request_info c =
           (c = client) =>
              (SNOC
                (isis_check fname)
                (isis_request_info state c))
          |
           (isis_request_info state c) in

        (update_ftrm_info new_ftrm_info
        (update_isis_request_info new_isis_request_info
          state)))))) /\
```

For an FTRM read request, the FTRM sends an appropriate failure notification and has no further effect if the request attempts to read a file that the requesting client does not have open. Otherwise, the FTRM puts the request on the end of the queue of the client process' pending FTRM requests and puts an ISIS read request on the end of the queue of its pending ISIS requests.

```
(~doftrmrequest state client (request_read fd) =
   (If
     (~(open_info state client fd))
     ((Send (ClientOutput client (reply_read failure []))) ;;
      (Call (FTRM state)))
     (Call
       (FTRM
         (let new_ftrm_info c =
           (c = client) =>
              (SNOC (request_read fd) (ftrm_info state c))
           |
            (ftrm_info state c) in

          let new_isis_request_info c =
           (c = client) =>
              (SNOC (isis_read fd) (isis_request_info state c))
           |
            (isis_request_info state c) in

        (update_ftrm_info new_ftrm_info
        (update_isis_request_info new_isis_request_info
          state))))))) /\
```

For an FTRM `write` request, the FTRM sends an appropriate failure notification and has no further effect if the request attempts to write to a file that the requesting client does not have open. Otherwise, the FTRM puts the request on the end of the queue of the client process' pending FTRM requests and puts an ISIS `write` request on the end of the queue of its pending ISIS requests.

```
(~doftrmrequest state client (request_write fd nlist) =
   (If
     (~(open_info state client fd))
     ((Send (ClientOutput client (reply_write failure))) ;;
      (Call (FTRM state)))
     (Call
       (FTRM
         (let new_ftrm_info c =
           (c = client) =>
              (SNOC
                (request_write fd nlist)
                (ftrm_info state c))
           |
            (ftrm_info state c) in

          let new_isis_request_info c =
           (c = client) =>
              (SNOC
                (isis_write fd nlist)
                (isis_request_info state c))
           |
            (isis_request_info state c) in

        (update_ftrm_info new_ftrm_info
        (update_isis_request_info new_isis_request_info
          state)))))))
```

The final lines of the `doftrmrequest` definition simply end the definition.

```
  '--
end};
```

Function doisisrequest

The function `doisisrequest` gives the processing of each ISIS request and how the result of that processing is used to update the queue of pending

FTRM requests, in response to a Tick input event. It is defined using new_recursive_definition,over the concrete-recursive type ISIS_request.

Before getting into the definition of doisisrequest proper, the specification defines two SML abbreviations and a convenience function used in this definition. The SML variables faildescriptor and failcontents have as values the file descriptor or file contents returned to requests for a file descriptor or file contents when these requests fail. The convenience function requestedaccess extracts the access requested by an FTRM open request, and otherwise has request value none.

```
val faildescriptor = --`Make_file_descriptor 0 none`--;

val failcontents = --`[]:(num)list`--;

new_recursive_definition {
 name = "requestedaccess",
 fixity = Prefix,
 rec_axiom = FTRM_request_Def,
 def =
  let
   val requestedaccess =
    --`requestedaccess:^FTRM_request -> ^Access`--;
  in
   --`
    (^requestedaccess (request_close fd) = none) /\
    (^requestedaccess (request_open fname access) = access) /\
    (^requestedaccess (request_read fd) = none) /\
    (^requestedaccess (request_write fd nlist) = none)
   `--
  end};
```

In understanding the definition of doisisrequest, it is necessary to anticipate the definition of ftrmResponse and the Tick processing in it before doisisrequest is called. This processing produces an intermediate value of the FTRM state parameter by removing the ISIS request at the head of the Tick client's queue of pending ISIS requests. After this processing, ftrmResponse calls doisisrequest with this intermediate value of the FTRM state parameter, the Tick client, and the ISIS request removed from the head of this client's ISIS queue.

The initial lines of the doisisrequest definition name the function, provide the appropriate theorem giving an abstract characterization of the function's domain, and define a local SML variable that gives type information for the polymorphic function being defined.

```
new_recursive_definition {
 name = "doisisrequest",
 fixity = Prefix,
 rec_axiom = ISIS_request_Def,
 def =
  let
   val doisisrequest =
    --`doisisrequest:
        ^FTRM_State -> 'Client -> ^ISIS_request -> ^FTRM_Proc`--;
  in
   --`
```

For an ISIS check request, the FTRM sends an appropriate failure notification and has no further effect if the requesting client has no pending FTRM requests or if its top-of-queue request is not an open request asking for some access. Otherwise, the check request is carried out, and information on the security levels of the requesting client and the requested file is returned by all servers. The FTRM then votes on the values returned by the servers, obtaining the majority responses for these levels.

If the majority-value levels indicate that the request is invalid — the client level does not dominate the file level for read access, the file level does not dominate the client level for write access, or the levels are not equal for read and write access — the FTRM removes the top-of-queue open request from the queue and sends an appropriate failure message.

If the request is valid, the FTRM puts a new ISIS request, an open request for the requested access, on the *head* of the requesting client's queue of pending ISIS requests. This placement guarantees that the correspondence between this ISIS open request and the top-of-queue FTRM open request is always maintained; if the ISIS request was placed at the end of the queue, it could be incorrectly associated with new FTRM requests that come in from the requesting client before the next Tick event for this client.

```
(^doisisrequest istate client (isis_check fname) =
  (If
    ((ftrm_info istate client = []) \/
     (requestedaccess (HD (ftrm_info istate client)) = none))
    (Call (FTRM istate))
```

```
(If
 (let ac =
   requestedaccess (HD (ftrm_info istate client)) in
  let clev =
   vote_level (\s. server_c_security istate s client) in
  let flev =
   vote_level (\s. server_f_security istate s fname) in

  ((ac = read) /\ ~(dom clev flev)) \/
  ((ac = write) /\ ~(dom flev clev)) \/
  ((ac = read_write) /\ ~(flev = clev)))

 (* case when open request is invalid *)

 (let new_ftrm_info c =
   (c = client) =>
     (TL (ftrm_info istate c))
   |
     (ftrm_info istate c) in

   (Send
     (ClientOutput
       client
       (reply_open failure ~faildescriptor));;
   (Call (FTRM (update_ftrm_info new_ftrm_info istate))))

 (* case when open request is valid *)

 (let ac =
   requestedaccess (HD (ftrm_info istate client)) in
  let new_isis_request_info c =
   (c = client) =>
     (CONS
       (isis_open fname ac)
       (isis_request_info istate c))
   |
     (isis_request_info istate c) in

   (Call
     (FTRM
       (update_isis_request_info
         new_isis_request_info istate)))))) /\
```

For an ISIS close request, the ISIS request is carried out, and the individual servers' responses as to whether the close succeeded are gathered. (A close attempt might fail on a server, for instance, if the server crashed.) Whether or not the voted status value is success, the FTRM removes the top-of-queue FTRM close request and reports the voted status to the client. If the voted value is success, the FTRM resets the client's open_info value to indicate that it no longer has the closed file open.

```
(~doisisrequest istate client (isis_close fd) =
   let st =
    vote_status
     (\s. return_close fd (server_files istate s)) in
   let new_ftrm_info c =
    (c = client) =>
      (TL (ftrm_info istate c))
    |
      (ftrm_info istate c) in
   let new_open_info c arbfd =
    ((c = client) /\ (arbfd = fd)) =>
      F
    |
      (open_info istate c arbfd) in

   (Send (ClientOutput client (reply_close st))) ;;
   (Call
     (FTRM
       ((st = failure) =>
         (update_ftrm_info new_ftrm_info istate)
       |
         (update_ftrm_info new_ftrm_info
         (update_open_info new_open_info istate)))))) /\
```

For an ISIS open request, the ISIS request is carried out, and the individual servers' responses as to whether the open succeeded and if so what file descriptor was returned are gathered. The FTRM evaluates the majority values of the status and the file descriptor. If the voted status value is failure, the FTRM sends an appropriate failure message with a dummy "failed" file descriptor and removes the top-of-queue FTRM open request. If the voted status value is success, the FTRM sends an appropriate success message with the voted file descriptor and resets the client's open_info value to indicate that it now has the opened file open, resets the server_files value to show that the opened file is open, and removes the top-of-queue FTRM open request.

```
(~doisisrequest istate client (isis_open fname ac) =
   let st =
    vote_status
     (\s.
       FST (return_open fname ac (server_files istate s))) in
   let fd =
    vote_file_descriptor
     (\s.
       SND (return_open fname ac (server_files istate s))) in
   let new_ftrm_info c =
    (c = client) =>
      (TL (ftrm_info istate c))
    |
      (ftrm_info istate c) in
   let new_open_info c arbfd =
    ((c = client) /\ (arbfd = fd)) =>
      T
    |
      (open_info istate c arbfd) in
   let new_server_files s fa =
    (fa = descriptor_address fd) =>
      (update_isopen T (server_files istate s fa))
    |
      (server_files istate s fa) in

   (If
     (st = failure)
     (Send
       (ClientOutput
         client
         (reply_open failure ~faildescriptor)))
     (Send (ClientOutput client (reply_open success fd)))) ;;
   (Call
     (FTRM
       ((st = failure) =>
         (update_ftrm_info new_ftrm_info istate)
       |
         (update_ftrm_info new_ftrm_info
         (update_open_info new_open_info
         (update_server_files new_server_files istate))))))) /\
```

For an ISIS read request, the ISIS request is carried out, and the individual servers' responses as to whether the read succeeded and if so what file contents were returned are gathered. The FTRM evaluates the majority values of the status and the file contents. If the voted status value is `failure`, the FTRM sends an appropriate failure message with a dummy "failed" file contents, but otherwise sends an appropriate success message with the voted file contents. In either case, it then removes the top-of-queue FTRM `read` request.

```
(^doisisrequest istate client (isis_read fd) =
   let st =
    vote_status
     (\s. FST (return_read fd (server_files istate s))) in
   let nlist =
    vote_num_list
     (\s. SND (return_read fd (server_files istate s))) in
   let new_ftrm_info c =
    (c = client) =>
      (TL (ftrm_info istate c))
    |
      (ftrm_info istate c) in

   (If
     (st = failure)
     (Send
       (ClientOutput
         client
         (reply_read failure ^failcontents)))
     (Send (ClientOutput client (reply_read success nlist)))) ;;
   (Call (FTRM (update_ftrm_info new_ftrm_info istate)))) /\
```

For an ISIS `write` request, the ISIS request is carried out, and the individual servers' responses as to whether the write succeeded are gathered. The FTRM evaluates the majority value of the status. If the voted status value is `success`, the FTRM changes the file contents of the file written on all servers, and otherwise leaves it unchanged on all servers. In either case, it returns the voted status value and removes the top-of-queue FTRM `write` request.

Note that this specification ignores the effect of a write request on failed servers, and assumes that files on failed servers are updated when then are updated on non-failed servers. This is unrealistic, but acceptable for the model because it treats the information on failed servers as meaningless.

```
(^doisisrequest istate client (isis_write fd nlist) =
        let st =
         vote_status
          (\s. return_write fd nlist (server_files istate s)) in
        let new_ftrm_info c =
         (c = client) =>
           (TL (ftrm_info istate c))
          |
           (ftrm_info istate c) in
        let new_server_files s fa =
         (fa = descriptor_address fd) =>
           (update_contents nlist (server_files istate s fa))
          |
           (server_files istate s fa) in

        (Send (ClientOutput client (reply_close st))) ;;
        (Call
          (FTRM
           ((st = failure) =>
             (update_ftrm_info new_ftrm_info istate)
            |
             (update_ftrm_info new_ftrm_info
             (update_server_files new_server_files istate)))))
    '--
   end};
```

Function ftrmResponse

The function `ftrmResponse` gives the response of the FTRM model process to an arbitrary input event.

The initial lines of the `ftrmResponse` definition name the function, provide the appropriate theorem giving an abstract characterization of the function's domain, and define a local SML variable that gives type information for the polymorphic function being defined.

```
new_recursive_definition {
 name = "ftrmResponse",
 fixity = Prefix,
 rec_axiom = FTRM_InEv_Def,
 def =
  let
   val ftrmResponse =
    --'ftrmResponse:^FTRM_State -> ^FTRM_InEv -> ^FTRM_Proc'--;
  in
```

For a ClientInput input, ftrmResponse just calls doftrmrequest with the current state parameter, the requesting client, and the request.

```
(^ftrmResponse state (ClientInput client request) =
   (doftrmrequest state client request)) /\
```

For a Fault input, ftrmResponse ignores the input if the current fault number is already maxfaults — the model does not consider the possibility of a greater number of faults. Otherwise, ftrmResponse updates the state parameter to increment the number of faults, notes that the faulted server is now faulty, and updates the values stored on that server to the values given in the fault event. There are the following exceptions, though: ftrmResponse makes no changes to the information on a server for files that would uncorrupt a corrupted file or rename an existing file.

```
(^ftrmResponse state (Fault serv fdata csdata fsdata) =
   (Call
     (FTRM
      ((fault_number state = maxfaults) =>
         state
       |
        (let new_fault_number = (fault_number state) + 1 in

         let new_server_fault s =
          (s = serv) =>
            T
          |
            (server_fault state s) in

         let new_server_files s fa =
          (s = serv) =>
            ((corrupted (fdata fa)) =>
               (fdata fa)
            |
             (corrupted (server_files state s fa))  =>
              (server_files state s fa)
            |
             ((hasname (fdata fa)) =
              (hasname (server_files state s fa))) =>
               (fdata fa)
          |
           (server_files state s fa))
        |
         (server_files state s fa) in

       let new_server_c_security s =
        (s = serv) =>
          csdata
        |
          (server_c_security state s) in

       let new_server_f_security s =
        (s = serv) =>
          fsdata
        |
          (server_f_security state s) in

       let newserverfiles s =
        (s = serv) =>
          fdata
        |
          (server_files state s) in

      Make_FTRM_State
       new_fault_number
       new_server_fault
       (ftrm_info state)
       (isis_request_info state)
       (open_info state)
       new_server_files
       new_server_c_security
       new_server_f_security))))) /\
```

For a Tick input, ftrmResponse ignores the input if the client named by the event has no pending ISIS requests. Otherwise, it computes an intermediate value of the FTRM state parameter in which the top-of-queue ISIS request for this client has been removed and calls doisisrequest with this intermediate state parameter, the Tick client, and the ISIS request removed from the top is its ISIS-request queue.

```
(~ftrmResponse state (Tick client) =
   (If
     (isis_request_info state client = []))
(Call (FTRM state))
(let new_isis_request_info c =
  (c = client) =>
    (TL (isis_request_info state c))
  |
    (isis_request_info state c) in
 let istate =
  update_isis_request_info new_isis_request_info state in

 doisisrequest
  istate
  client
  (HD (isis_request_info state client)))))
```

The final lines of the ftrmResponse definition simply end the definition.

```
  '--
 end};
```

Function ftrm

The function ftrm gives the top-level description of the FTRM model process. It is simple: the process waits for an arbitrary input event, then invokes FTRM_Response to determine its response as a function of its state parameter and this input event. (The Receive PSL command supplies the input event received, implicitly taken off a buffer, as an parameter to the function invoked.)

```
new_definition(
 "ftrm",
 let
  val ftrm = --'ftrm:^FTRM_State -> ^FTRM_Proc'--;
 in
  --'
   ^ftrm state =
     (Receive (\ev:^FTRM_InEv. T) (FTRM_Response state))
  '--
 end);
```

Function FTRM_InvocVal

The function FTRM_InvocVal maps every invocation to the corresponding value of a PSL-valued function, mapping the invocation constructor FTRM to the function `ftrm` and the invocation constructor `FTRM_Response` to the function `ftrmResponse`.

```
new_recursive_definition {
 name = "FTRM_InvocVal",
 fixity = Prefix,
 rec_axiom = FTRM_Invoc_Def,
 def =
  let
   val FTRM_InvocVal =
    --`FTRM_InvocVal:^FTRM_Invoc -> ^FTRM_Proc`--
  in
   --`
    (^FTRM_InvocVal (FTRM state) =
       (ftrm state)) /\

    (^FTRM_InvocVal (FTRM_Response state inev) =
       (ftrmResponse state inev))
   `--
  end};
```

7.4.10 Saving the Theory

The following lines write the theory of the FTRM model just constructed to disk and cause HOL90 to exit.

```
export_theory();
exit();
```

7.5 Proofs

Because of a lack of time, we did not complete any proofs that the FTRM model is restrictive or has security properties that will help to establish that it is restrictive. We do believe, however, that it would be reasonably easy to show that the FTRM model is a server process and that its output events in response to arbitrary input events are at security levels that dominate the levels of these input events.

Chapter 8

The Real-Time Scheduling Model

8.1 Introduction

The purpose of this model is to explore some aspects of service assurance requirements for hard real-time systems. It is an illustrative example of how Romulus can be used to specify state machines with timing information, and how such timed state machines can be used to model real-time systems. In this example, the real-time system is required to schedule two representative tasks similar to those handled by the Operational Flight Program (OFP) of the A-7E Navy aircraft:

- a *periodic* task, which repeatedly performs some task, for example, updating a navigational database with the current position of the aircraft, and
- a *sporadic* task, which is initiated at the request of the pilot to perform some task, for example, firing a missile.

These tasks must be executed (on a single processor, in this example) in such a way that the service assurance requirements of both are met.

A hard real-time system is one designed to meet requirements not only on what actions it performs, but also when it performs them. Such a system must schedule processes to perform tasks that are *time-critical* (i.e., they must be performed in a "timely" fashion).

In the context of a hard real-time system, service assurance properties relate to the timeliness of time-critical tasks and are expressed as constraints on the timing of processes. Timing constraints can of course be arbitrarily complex. However, in this notoriously difficult (and hazardous) field, there have evolved certain useful models of processing and their hard real-time requirements. These models are both general and powerful enough to deal with a wide variety of real-time processing needs, in particular for avionics processing. They are also simple enough to admit full analysis and safe implementation.

Two real-time scheduling algorithms were considered for this example. Liu and Layland [12] examine the *earliest deadline* scheduling algorithm: at every time unit, the process with the earliest deadline is executed. Later work on the Romulus project [16] developed an extension of these results to allow a finer analysis of scheduling problems. We also mention that on a related topic, earlier Romulus work [9] outlined a temporal logic approach to analyzing real-time concurrent systems and scheduling problems, though this did not involve PSL process specifications in the traditional Romulus style. However, the earliest deadline algorithm requires that all processes be periodic, and that their deadlines must be the same as their periods. The first limitation can be overcome by using a transformation technique described in [17], which maps a scheduling problem involving periodic and sporadic processes into another problem involving periodic processes only. Then, if the latter problem has a solution, the original one does, and the solution to the original problem can be derived.

The second limitation is more serious. Because of it, we have decided to use the scheduling algorithm Teixeira presents in [23]. This is the *static priority interrupt* scheduling algorithm, in which each process is assigned a priority, and, at every time unit, the process with the highest priority is run. This algorithm works for both periodic and sporadic processes, and does not have the restriction that process deadlines are the same as their periods. While this flexibility has a price — lower processor utilization — Teixeira observes that this is outweighed, in many applications, by the ease of implementation and efficiency of the algorithm.

For further discussion of the service assurance theory, see Volume II of the Romulus Documentation Set.

8.2 Description of the Model

In the type of model used in this example, there is one processor, and a set of processes M. A task $T \in M$ may be either periodic or sporadic. Processor time is divided into integral time units, and all processes are independent, i.e., there are no inter-process synchronization requirements. Processes are also interruptible; a process need not be run to completion before another is begun.

Each task $T_i = (c_i, d_i, p_i) \in M$ has three parameters: c_i, the computation time, d_i, the deadline by which the task must be completed, and p_i, the period. If the task is periodic, then it is requested at time kp_i for every non-negative integer k. If the task is sporadic, then it can be requested at any time, but two consecutive requests must be at least p_i time units apart. Each task also has an associated priority, and the tasks are numbered so that if $i < j$, the priority of T_i is greater than that of T_j. No two processes have the same priority.

All time parameters are non-negative integers, and for all tasks, $0 < c_i \leq d_i \leq p_i$. The *utilization factor* of a task is c_i/p_i, and we consider only those sets of tasks in which the sum of utilization factors is less than or equal to 1 (i.e., the number of processors). Thus, the period of each task must be at least 2 if there is more than one task in M.

A service assurance requirement for a task $T_i = (c_i, d_i, p_i)$, is, then, that after each request at time t, the task is finished at or before time $t + d_i$.

In this example, there are two processes:

- The *trigger* is a sporadic task $T_1 = (c_1, d_1, p_1)$, which fires a missile when requested.

- The *updater* is a periodic task $T_2 = (c_2, d_2, p_2)$, which updates the navigational database every p_2 time units.

As the task numbers indicate, the trigger task has a higher priority than the updater task.

At each time unit, the process with the higher priority is selected to run. The updater task is requested to run by the scheduler, every p_2 time units. The trigger task is requested to run when a "trigger" input event is received.

With the static priority interrupt algorithm, an additional restriction (the *latency* restriction) is placed on the task parameters to ensure that both processes meet their deadlines. The function CPU is defined as:

$$\mathrm{CPU}\,(t, i) = (t\,\mathtt{DIV}\,p_i) * c_i + \mathrm{MIN}\,(c_i, t\,\mathtt{MOD}\,p_i)$$

This function gives the maximum amount of CPU time that task i will use in the interval $[0 \ldots t]$. The maximum amount of time is used when the task is requested as often as possible; in each period p_i the amount of CPU time used will be c_i, and in the remaining partial period, the amount of CPU time used will be the minimum of c_i and the remainder of $t\,\mathtt{DIV}\,p_i$. For the updater process, the latency restriction is that the amount of latency $(d_2 - c_2)$ is greater than or equal to the maximum amount of CPU time that the trigger process might require before the deadline of the updater process is reached:

$$d_2 \geq c_2 + \mathrm{CPU}\,(d_2, 1)$$

It should be noted that only scheduling is modeled in this example. The updater and trigger processes are treated as "black boxes;" the only information that is known about them is their task parameters, and that they are independent. It is certainly possible to add more details about these processes, for example, specifications fo what the processes actually do, but this information is not relevant to the proof that the scheduling meets all deadlines.

8.3 The HOL Model

This section contains and describes the Standard ML code for a formal description of the scheduler model in PSL. The scheduler is a PSL server process which receives an input event at the beginning of each time unit, updates its state parameter so as to simulate the running of one of the trigger or updater processes during that time unit if at least one is ready to run, and returns to wait for the next input event. The scheduler state is a record with fields for the time left to run for each process, the total run time for each process, the time left until the next valid trigger request, and the time left until the end of the current updater process period.

Note that the word "process" in this document describes two different kinds of entities:

1. PSL processes (the scheduler), and

2. simulated processes "run" by the scheduler (the trigger and updater).

Since only scheduling is modeled in this example, the simulated processes have no state and exist only in the state of the scheduler. The service assurance requirements (see section 8.3.13) may be expressed in terms of the sequence of values that the scheduler state will assume as input events are received. For example, the run time of a process in an interval is the difference between the process' total run time at the end of the interval and at the beginning of the interval.

The Romulus PSL is general enough to serve as a description language for this model. The Romulus type definition utilities are useful for constructing HOL types. Some of the Romulus security theory is also useful in proving properties about the model, even though the focus here is on service assurance rather than nondisclosure.

8.3.1 Prologue

Older versions of the OFP theory are removed and the OFP theory is created. The Romulus library is loaded and then the arithmetic and tautology checker libraries are loaded.

```
System.system "rm -f ofp.holsig ofp.thms";
new_theory "ofp";
load_library{lib = get_library_from_disk "romulus",
             theory = "-"};

load_library{lib=arith_lib, theory="-"};
load_library{lib=taut_lib, theory="-"};
```

The internal state of a process is a record containing various scheduling-related facts. The following functions are used for defining these records.

The function `prove_record_field_thms` proves access theorems for a record type defined by the Romulus `romrecord` function. The induction theorem for the record type must be proved before this function can be used. Given a record type name `rec` and a list of field names [$\mathtt{f}_1 \ldots \mathtt{f}_n$], for each pair of field names $\mathtt{f}_i$ and $\mathtt{f}_j$ one of the following theorems is proved:

- If $i = j$, `!rec. fi (update_fi x rec) = x`.

  If a record field is updated, extracting the field will result in the updated value.

- If $i \neq j$, `!rec. fi (update_fj x rec) = fi rec.`

  If a record field is updated, extracting a different field will result in the field of the original record.

The function `show_concrete_props`, copied from `romutils.sml`, proves that a concrete recursive type's constructors are distinct and one-to-one, and proves theorems allowing proof by induction or by cases for the type.

```
fun prove_record_field_thms recname flds =
 let
  fun field_update_thm afld ufld =
   let
    val thm =
     if afld = ufld then
      "!rec:" ^ recname ^ ".
         (" ^ afld ^ " (update_" ^ afld ^ " x rec)) = x"
     else
      "!rec:" ^ recname ^ ".
         (" ^ afld ^ " (update_" ^ ufld ^ " x rec)) = (" ^
       afld ^ " rec)";
    val uname = "update_" ^ ufld;
    in
     prove (string_to_term thm,
            INDUCT_THEN (theorem "-" (recname^"_Induct")) ASSUME_TAC THEN
            REWRITE_TAC [(definition "-" uname),
                         (definition "-" afld)])
   end;
 in
  flatten (map (fn x => map (fn y => (field_update_thm x y)) flds) flds)
 end;

fun show_concrete_props defth stem =
 let
  fun show_distinct th =
   let
    val dist = prove_constructors_distinct th
   in
    save_thm(
     (stem^"_Distinct"),
     CONJ
      dist
      (LIST_CONJ(map
                 (GEN_ALL o NOT_EQ_SYM o SPEC_ALL)
                 (CONJUNCTS dist))))
   end;
  fun show_one_one th =
   save_thm((stem^"_One_One"),(prove_constructors_one_one th));
  val induct = save_thm((stem^"_Induct"),(prove_induction_thm defth))
 in
  save_thm((stem^"_Cases"),(prove_cases_thm induct));
  if(can show_distinct defth) then true else false;
  if(can show_one_one defth) then true else false;
  defth
 end;
```

8.3.2 Task Parameters

The functions Ctime, Deadline, and Period give the computation time, deadline, and period (or minimum separation) for the updater and trigger processes. The function CPU gives the maximum CPU time for a process. The axiom ok_processes_assum gives the assumed constraints on the process parameters.

```
new_constant {
 Name = "Ctime",
 Ty = ==`:num->num`==};

new_constant {
 Name = "Deadline",
 Ty = ==`:num->num`==};

new_constant {
 Name = "Period",
 Ty = ==`:num->num`==};

val MIN =
new_definition (
 "MIN",
 --`(MIN x y) = ((x <= y) => x | y)`--);

val CPU =
new_definition (
 "CPU",
 --`(CPU (t:num) (p:num)) =
    ((t DIV (Period p)) * (Ctime p)) +
     (MIN (Ctime p) (t MOD (Period p)))`--);
```

```
val ok_processes_assum =
 new_open_axiom (
  "ok_processes_assum",
   --`((Ctime 1) > 0) /\
     ((Deadline 1) >= (Ctime 1)) /\
     ((Period 1) >= (Deadline 1)) /\
     ((Ctime 2) > 0) /\
     ((Deadline 2) >= (Ctime 2)) /\
     ((Period 2) >= (Deadline 2)) /\
     ((Deadline 2) >= ((Ctime 2) + (CPU (Deadline 2) 1))) /\
     ((Period 1) > 1) /\
     ((Period 2) > 1)`--);
```

8.3.3 State

The model's state contains the time left to execute and the total run time for each process. Also in the state are the time left in the updater's period, and the time left until the next valid trigger request. A process is *ready* (can be executed) if its time left to run is nonzero, or if, at the beginning of the time period, it is made ready to run.

Note the use of the Romulus record definition function `romrecord` to define the state as a record. The accessor function `x` and the update function `update_x` for each field `x`, and the constructor function `Make_OFP_State` are automatically defined. The concrete type properties and field access theorems are then proved.

```
val (OFP_State_Def, OFP_State) =
 romrecord
 "OFP_State"
 [
  ("left_t",==`:num`==),           (* time left for trigger *)
  ("left_u",==`:num`==),           (* time left for updater *)
  ("run_t",==`:num`==),            (* total trigger runtime *)
  ("run_u",==`:num`==),            (* total updater runtime *)
  ("next_t",==`:num`==),           (* time to next valid
                                                trigger req *)
  ("period_u",==`:num`==)          (* period for updater *)
 ];

show_concrete_props (theorem "-" "OFP_State_Def") "OFP_State";
```

```
val state_field_defs =
 [(definition "-" "left_t"),
  (definition "-" "left_u"),
  (definition "-" "run_t"),
  (definition "-" "run_u"),
  (definition "-" "next_t"),
  (definition "-" "period_u")
 ];

val state_field_thms =
 prove_record_field_thms "OFP_State"
  ["left_t",
   "left_u",
   "run_t",
   "run_u",
   "next_t",
   "period_u"];
```

8.3.4 Input Events

Input events to the scheduler occur at regular intervals, and an input event may be either Tick or Trigger. The Tick event just indicates that the next time unit is to begin. The Trigger event is received (instead of a Tick event) when the pilot requests that a missile be fired. The concrete type properties are proved.

```
val (OFP_InEv_Def, OFP_InEv) =
 romcontype
  "OFP_InEv"
  [("Tick", []),
   ("Trigger", [])
  ];

show_concrete_props (theorem "-" "OFP_InEv_Def") "OFP_InEv";
```

8.3.5 Output Events

There are no output events in this model, since, as noted above, only scheduling is modeled in this example.

8.3.6 Invocations

The names for calls to process-valued functions are defined, and the concrete type properties are proved.

```
val (OFP_Invoc_Def, OFP_Invoc) =
 romcontype
  "OFP_Invoc"
  [("Ofp", [==`:^OFP_State`==]),
   ("Ofp_Response", [==`:^OFP_State`==, ==`:^OFP_InEv`==])];

show_concrete_props (theorem "-" "OFP_Invoc_Def") "OFP_Invoc";
```

8.3.7 PSL Processes

The SML variable `OFP_Proc` is defined as an abbreviation for the type of the OFP scheduler process. There are no output events, so the HOL type one is used as the output event type.

```
val OFP_Proc =
 ty_antiq(==`:(one,^OFP_InEv,^OFP_Invoc)process`==);
```

8.3.8 Initial State Parameter

Before the scheduler starts, there are no ready processes and no process has been run, so the `left` and `run` fields for both processes are zero. The period of the updater process is about to begin, so `period_u` is set to zero, which will cause an update to be requested. A trigger request is acceptable, so the `next_t` field is zero.

```
val OFP_InitParam =
new_definition (
 "OFP_InitParam",
 --`OFP_InitParam:^OFP_State = (Make_OFP_State 0 0 0 0 0 0)`--);
```

8.3.9 Top-level Function

The function `ofp` is defined, which is the top-level description of the scheduler process. The scheduler waits for an arbitrary input event, and then invokes `Ofp_Response` to determine its response as a function of the state parameter and the received input event.

```
new_definition (
 "ofp",
 let
  val ofp = --`ofp:^OFP_State -> ^OFP_Proc`--
 in
  --`(^ofp ofps) =
   (Receive (\ev:^OFP_InEv. T) (Ofp_Response ofps))`--
 end);
```

8.3.10 Running a Process

The function run_proc simulates the running of a process for one time unit by updating fields in its argument state. If there is only one ready process, it is run. Otherwise, the process with the highest priority is run. The run time of the running process is incremented, and the time to the next trigger request and the updater process period are decremented. Note that when this function is finished, the time unit is over, so the updater period is set to one less than the relevant process parameter if the current value is zero.

```
val run_proc =
new_definition (
 "run_proc",
 --`run_proc ofps =
  (Make_OFP_State

   (* if trigger is ready, run it *)

   (((left_t ofps) > 0) =>
    ((left_t ofps) - 1) | (left_t ofps))

   (* if updater is ready and trigger is not, run it *)

   ((((left_u ofps) > 0) /\ ((left_t ofps) = 0)) =>
    ((left_u ofps) - 1) | (left_u ofps))

   (* increment the trigger or the updater runtimes if required *)

   (((left_t ofps) > 0) =>
    (SUC (run_t ofps)) | (run_t ofps))
   ((((left_u ofps) > 0) /\ ((left_t ofps) = 0)) =>
    (SUC (run_u ofps)) | (run_u ofps))
(* decrement the minimum time to the next trigger if it is nonzero *)

(((next_t ofps) > 0) => (next_t ofps) - 1 | 0)

(* decrement the updater period or reset it if it is zero *)

(((period_u ofps) = 0) => ((Period 2) - 1) | (period_u ofps) - 1))'--);
```

8.3.11 Response Function

The ofp_Response function is defined using two auxiliary functions, do_tick and do_trigger. Function do_tick, called if the current input event is Tick, starts a new updater process if the updater period is zero. Function do_trigger, called if the current input event is Trigger, starts a trigger process if the time to the next valid trigger request is zero. It also updates the state with do_tick, since a Trigger input event, if received, takes the place of a Tick event and actions resulting from the input of a Tick event must also take place if a Trigger event is received.

The ofp_Response function updates the state with do_tick or do_trigger, depending on the input event, and updates the resulting state with run_proc.

```
val do_tick =
new_definition (
 "do_tick",
 --'do_tick ofps =
     (((period_u ofps) = 0) =>
      (update_left_u ((Ctime 2) + (left_u ofps)) ofps) |
      ofps)'--);

val do_trigger =
new_definition (
 "do_trigger",
 --'do_trigger ofps =
     (((next_t ofps) = 0) =>
      (update_left_t (Ctime 1)
       (update_next_t (Period 1) (do_tick ofps))) |
      (do_tick ofps))'--);

val ofp_Response =
new_recursive_definition {
 name = "ofp_Response",
 fixity = Prefix,
 rec_axiom = OFP_InEv_Def,
 def =
  let
   val ofp_Response =
    --`ofp_Response:^OFP_State -> ^OFP_InEv -> ^OFP_Proc`--
  in
   --`((^ofp_Response ofps Tick) =
       (Call (Ofp (run_proc (do_tick ofps))))) /\

      ((^ofp_Response ofps Trigger) =
       (Call (Ofp (run_proc (do_trigger ofps)))))`--
  end};
```

8.3.12 Invocation Values

The function OFP_InvocVal maps an invocation to the value of a process-valued function.

```
new_recursive_definition {
 name = "OFP_InvocVal",
 fixity = Prefix,
 rec_axiom = OFP_Invoc_Def,
 def =
  let
   val OFP_InvocVal =
    --`OFP_InvocVal:^OFP_Invoc -> ^OFP_Proc`--
  in
   --`(^OFP_InvocVal (Ofp ofps) =
       (ofp ofps)) /\
      (^OFP_InvocVal (Ofp_Response ofps inev) =
       (ofp_Response ofps inev))`--
  end};
```

8.3.13 Requirements

The function OFP_Reaction maps times to model states, and the requirements are stated in terms of conditions on these states. PossibleNextParameter is defined in the Romulus security theory, and is used to derive the next state parameter to the scheduler process. The SML variable inevseq is used to abbreviate the HOL term inevseq:num->OFP_InEv.

```
val OFP_Reaction =
new_recursive_definition {
 name = "OFP_Reaction",
 fixity = Prefix,
 rec_axiom = (theorem "prim_rec" "num_Axiom"),
 def =
  let
   val OFP_Reaction =
    --`OFP_Reaction:(num -> OFP_InEv) -> num -> OFP_State`--;
  in
   --`
    (^OFP_Reaction inevseq 0 = OFP_InitParam) /\

    (^OFP_Reaction inevseq (SUC n) =
       @nextstate.
         PossibleNextParameter
          OFP_InvocVal
          Ofp
          nextstate
          (ofp_Response (OFP_Reaction inevseq n) (inevseq n)))
   `--
  end};

val inevseq = --`inevseq:num->OFP_InEv`--;
```

For a sequence of input events `inevseq` and a time `n`, the model state immediately after time `n` is `(OFP_Reaction inevseq n)`. Note also that the first input event is numbered one, not zero.

The requirements are:

1. If a trigger process is requested at time n, and the request is valid (it is not too soon after the previous trigger request, if there was one), the run time of the process between n and $n + \text{Deadline}(1)$ is $\text{Ctime}(1)$.

   ```
   !inevseq n.
   (((inevseq n) = Trigger) /\
    (next_t (OFP_Reaction inevseq n) = 0)) ==>
   ((run_t (OFP_Reaction inevseq (n + Deadline 1))) -
    (run_t (OFP_Reaction inevseq n)) = (Ctime 1))
   ```

2. An updater process runs every $n \cdot \text{Period}(2)$ time units. It is requested at the beginning of each interval, and the run time of the process between $n \cdot \text{Period}(2)$ and $n \cdot \text{Period}(2) + \text{Deadline}(2)$ is $\text{Ctime}(2)$.

```
!inevseq n.
(run_u (OFP_Reaction inevseq (n * Period 2 + Deadline 2))) -
 (run_u (OFP_Reaction inevseq (n * Period 2))) =
Ctime 2
```

8.3.14 Epilogue

Finally, the theory is exported and the HOL session terminated.

```
export_theory();
exit();
```

8.4 Proofs

This chapter contains and describes the SML code which proves the requirements stated above. Only the SML input is given; the proof transcript is about 17000 lines, too large for this document.

For brevity, the following abbreviations are used in the proof descriptions:

left_t (n)	`(left_t (OFP_Reaction inevseq n))`
left_u (n)	`(left_u (OFP_Reaction inevseq n))`
run_t (n)	`(run_t (OFP_Reaction inevseq n))`
run_u (n)	`(run_u (OFP_Reaction inevseq n))`
next_t (n)	`(next_t (OFP_Reaction inevseq n))`
period_u (n)	`(period_u (OFP_Reaction inevseq n))`

8.4.1 Tactics and Elementary Lemmas

This section describes the tactics and elementary lemmas (mostly about arithmetic) that are used in the requirement proofs.

Rewriting and Miscellaneous Tactics

The tactic `ARITH_RES_TAC` proves an assertion of linear natural number arithmetic using `ARITH_CONV` from the HOL `arith` library, resolves the resulting theorem with the assumptions of the goal, and adds any new results to the assumptions. The `EQT_ELIM` step is required because `(ARITH_CONV x)` produces the result `x = T`.

```
fun ARITH_RES_TAC x =
 IMP_RES_TAC (EQT_ELIM (ARITH_CONV x));
```

The tactic ARITH_REWRITE_TAC uses ARITH_CONV to prove a list of assertions and then rewrites the goal with the resulting list of theorems.

```
fun ARITH_REWRITE_TAC x =
 REWRITE_TAC (map (EQT_ELIM o ARITH_CONV) x);
```

The tactic PTAUT_REWRITE_TAC is like ARITH_REWRITE_TAC, except that the tautology checker PTAUT_CONV from the HOL library taut is used instead of ARITH_CONV.

```
fun PTAUT_REWRITE_TAC x =
 REWRITE_TAC (map (EQT_ELIM o Taut.PTAUT_CONV) x);
```

The tactic COND_REWRITE_TAC provides a limited conditional rewrite rule capability. Given a theorem of the form:

$$u_1 \Longrightarrow u_2 \Longrightarrow \ldots \Longrightarrow u_n \Longrightarrow x = y$$

this tactic works like the tactic

```
IMP_RES_THEN (REWRITE_TAC o (fn x => [x]))
```

except that a resolvent may be formed by more than one application of Modus Ponens. That is, an attempt is made to resolve each antecedent u_i of the given theorem with the assumptions of the goal, and, if successful, the antecedent is removed from the theorem by an application of Modus Ponens. If all antecedents are successfully removed, the result is used to rewrite the goal. The theorem may also have antecedents that are conjunctions; the theorem will be converted into the above (canonical) form by RES_CANON.

The tactic ARITH_COND_REWRITE_TAC is similar, except that the argument is an assertion that is first proved with ARITH_CONV.

```
fun is_implication x =
 if is_forall x then
  is_implication (#Body (dest_forall x))
 else is_imp x andalso (not (is_neg x));

fun COND_REWRITE_TAC x =
 IMP_RES_THEN (fn y => if is_implication (concl y) then
                              (COND_REWRITE_TAC y)
                             else
                              (REWRITE_TAC [y])) x;

fun ARITH_COND_REWRITE_TAC x =
       COND_REWRITE_TAC (EQT_ELIM (ARITH_CONV x));
```

The tactic rom_condcases_TAC, copied from romtactics.sml, splits a goal involving conditional expressions into separate cases.

```
fun rom_condcases_TAC (A,gl) =
 ((COND_CASES_TAC THEN
   ASM_REWRITE_TAC [] THEN
   rom_condcases_TAC)
  ORELSE
  ALL_TAC) (A,gl);
```

The tactic UNDISCH_N_TAC "undischarges" an assumption of the goal specified by its position in the assumption list. This eliminates the need to quote an assumption in order to undischarge it.

```
fun UNDISCH_N_TAC n =
 ASSUM_LIST (fn thl =>
            (UNDISCH_TAC (concl (el n thl))));
```

Model-specific tactics

The tactic reaction_cases_TACsplits an expression involving OFP_Reaction into two cases, one for each possible input event. The argument term represents the input event on which case analysis is required. Because of the way that OFP_Reaction is defined, the goal must have terms of the form (OFP_Reaction inevseq (SUC n)), and, in the resulting cases, each occurrence of this term is rewritten to a term of the form

```
@nextstate.
  PossibleNextParameter
   OFP_InvocVal
   Ofp
   nextstate
   (ofp_Response (OFP_Reaction inevseq n) Tick))
```

or

```
  @nextstate.
    PossibleNextParameter
     OFP_InvocVal
     Ofp
     nextstate
     (ofp_Response (OFP_Reaction inevseq n) Trigger))
```

which can be simplified with simplify_response_TAC, described below.

```
fun reaction_cases_TAC inev =
 REWRITE_TAC [OFP_Reaction] THEN
 SPEC_TAC (inev,--`inev:OFP_InEv`--) THEN
 INDUCT_THEN (theorem "-" "OFP_InEv_Induct") ASSUME_TAC;
```

If it is known by hypothesis which input event will occur, ASM_REWRITE_TAC-[OFP_Reaction] can be used instead of this tactic, and no input case analysis is required.

The tactic simplify_response_TAC simplifies a goal involving terms resulting from rewriting applications of OFP_Reaction; it is usually applied after reaction_cases_TAC. The tactic first rewrites the goal with the definition of ofp_Response, which results in a PSL process. For this model, the only possible operator is Call, so the second rewrite will result in the invocation's parameter, using the fact that OFP_Invoc is a one-to-one function and the select_eq theorem to simplify the choice (@) expression. The resulting parameter is in terms of the run_proc, do_trigger, and do_tick functions, so the goal can be further simplified by rewriting with the definitions of these functions and the state field definitions and theorems. All three of these functions have conditional expressions, and it turns out to be best to do a case split on the conditions in do_trigger and do_tick, so there is an application of rom_condcases_TAC after rewriting with these functions but before rewriting with run_proc. The end result is six cases, two for a Tick input and four for a Trigger input.

```
store_thm (
 "select_eq",
 --`(@x:'a . x = y) = y`--,
 CONV_TAC SELECT_CONV THEN
 EXISTS_TAC (--`y:'a`--) THEN
 REFL_TAC);

val PossibleNextParameter_Call_Self =
 GEN_ALL
  (el
   4
   (CONJUNCTS
   (SPEC_ALL
    (theorem "romsecure" "PossibleNextParameter_Rewrites"))));

val simplify_response_TAC =
 REWRITE_TAC [ofp_Response] THEN
 REWRITE_TAC [PossibleNextParameter_Call_Self,
              (theorem "-" "OFP_Invoc_One_One"),
              (theorem "-" "select_eq")] THEN
 REWRITE_TAC [do_trigger, do_tick] THEN
 rom_condcases_TAC THEN
 REWRITE_TAC ([run_proc] @ state_field_defs @ state_field_thms);
```

In many proofs, application of

```
reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC
```

to a goal will result in six cases, and then each case can be proved by rewriting with the assumptions of the case, or by arithmetic simplification with CONV_TAC ARITH_CONV (possibly using ok_processes_assum and the assumptions of the case), or by showing that the assumptions of the case are contradictory.

Arithmetic Theorems

The following theorems state some of the properties of the MIN (minimum) function. They are useful for simplifying expressions involving MIN, and for subgoaling with MATCH_MP_TAC.

```
val MIN_GE =
 store_thm (
  "MIN_GE",
  --`!a b. (a >= b) ==> ((MIN a b) = b)`--,
  REWRITE_TAC [MIN] THEN
  CONV_TAC ARITH_CONV);

val MIN_GT =
 store_thm (
  "MIN_GT",
  --`!a b. (a > b) ==> ((MIN a b) = b)`--,
    REWRITE_TAC [MIN] THEN
    CONV_TAC ARITH_CONV);

  val MIN_LE =
   store_thm (
    "MIN_LE",
    --`!a b. (a <= b) ==> ((MIN a b) = a)`--,
    REWRITE_TAC [MIN] THEN
    CONV_TAC ARITH_CONV);

  val LT_MIN =
   store_thm (
    "LT_MIN",
    --`!a b c. ((a < b) /\ (a = (MIN b c))) ==> (b > c)`--,
    REWRITE_TAC[MIN] THEN
    CONV_TAC ARITH_CONV);

  val GE_MIN =
   store_thm (
    "GE_MIN",
    --`!a b c. ((a >= b) /\ (a = (MIN b c))) ==> (b <= c)`--,
    REWRITE_TAC [MIN] THEN
    CONV_TAC ARITH_CONV);
```

The following theorems describe some of the properties of the DIV and MOD functions. MOD_SUC and DIV_SUC are useful in inductive proofs.

```
  g(`(0 < b) ==>
     (((SUC a) MOD b) = (((a MOD b) = (b - 1)) => 0 | SUC (a MOD b)))`);

  e (STRIP_TAC);
  e (MATCH_MP_TAC (theorem "arithmetic" "MOD_UNIQUE"));
  e (EXISTS_TAC (--`(((a MOD b) = (b - 1)) =>
       (SUC (a DIV b)) | (a DIV b))`--));
  e (IMP_RES_TAC (definition "arithmetic" "DIVISION"));
  e (ASSUM_LIST (fn thl => (MP_TAC (SPEC (--`a:num`--) (el 2 thl)))));
  e (COND_CASES_TAC);

  e (REWRITE_TAC [theorem "arithmetic" "MULT_CLAUSES"]);
  e (UNDISCH_TAC (--`a MOD b = b - 1`--));
  e (UNDISCH_TAC (--`0 < b`--));
  e (CONV_TAC ARITH_CONV);

  e (ASSUM_LIST (fn thl => (MP_TAC (SPEC (--`a:num`--) (el 2 thl)))));
```

```
e (UNDISCH_TAC (--`~(a MOD b = b - 1)`--));
e (CONV_TAC ARITH_CONV);
val MOD_SUC = save_top_thm "MOD_SUC";

g(`(0 < b) ==>
    (((SUC a) DIV b) =
     (((a MOD b) = (b - 1)) => (SUC (a DIV b)) | (a DIV b)))`);

e (STRIP_TAC);
e (MATCH_MP_TAC (theorem "arithmetic" "DIV_UNIQUE"));
e (EXISTS_TAC (--`((a MOD b) = (b - 1)) => 0 | (SUC (a MOD b))`--));
e (IMP_RES_TAC (definition "arithmetic" "DIVISION"));
e (ASSUM_LIST (fn thl => (MP_TAC (SPEC (--`a:num`--) (el 2 thl)))));
e (COND_CASES_TAC);

e (REWRITE_TAC [theorem "arithmetic" "MULT_CLAUSES"]);
e (UNDISCH_TAC (--`a MOD b = b - 1`--));
e (UNDISCH_TAC (--`0 < b`--));
e (CONV_TAC ARITH_CONV);

e (ASSUM_LIST (fn thl => (MP_TAC (SPEC (--`a:num`--) (el 2 thl)))));
e (UNDISCH_TAC (--`~(a MOD b = b - 1)`--));
e (CONV_TAC ARITH_CONV);
val DIV_SUC = save_top_thm "DIV_SUC";

val MOD_NONZERO =
 store_thm (
  "MOD_NONZERO",
  --`((k > 0) /\ (k < p)) ==> ~(((n * p) + k) MOD p = 0)`--,
  STRIP_TAC THEN
  IMP_RES_TAC (theorem "arithmetic" "MOD_MULT") THEN
  ASM_REWRITE_TAC[] THEN
  UNDISCH_N_TAC 3 THEN CONV_TAC ARITH_CONV);
```

Process Parameter Theorems

The following theorems state facts derivable from the assumed process parameter assumptions; `Period_1_gt_0` is required when doing proofs involving natural number arithmetic for the trigger process requirement, and `Period_2_gt_0` is required when doing proofs involving `DIV` and `MOD` for the updater process requirement.

```
val Period_1_gt_0 =
 store_thm (
 "Period_1_gt_0",
 --`0 < Period 1`--,
 MP_TAC (ok_processes_assum) THEN CONV_TAC ARITH_CONV);

val Period_2_gt_0 =
 store_thm (
  "Period_2_gt_0",
  --`0 < Period 2`--,
  MP_TAC (ok_processes_assum) THEN CONV_TAC ARITH_CONV);
```

8.4.2 Trigger Requirement Proof

The proof that the trigger process requirement is met is fairly simple, since the trigger process has the higher priority, and if it is ready to run, it will be run.

`left_t_le_next_t`

We first show that left_t $(n) \leq$ next_t (n). The proof is by induction on n, and case analysis of the possible inputs. This lemma is used in the proof of `left_t_run_t_step`.

```
g(`!^inevseq n.
   (left_t (OFP_Reaction inevseq n)) <=
      (next_t (OFP_Reaction inevseq n))`);

e (GEN_TAC THEN INDUCT_TAC);
e (REWRITE_TAC ([OFP_Reaction, OFP_InitParam] @ state_field_defs));
e (CONV_TAC ARITH_CONV);

e (UNDISCH_N_TAC 1);
e (reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC);

e (CONV_TAC ARITH_CONV);
e (CONV_TAC ARITH_CONV);
e (MP_TAC ok_processes_assum THEN CONV_TAC ARITH_CONV);
e (MP_TAC ok_processes_assum THEN CONV_TAC ARITH_CONV);
e (UNDISCH_N_TAC 2 THEN CONV_TAC ARITH_CONV);
e (UNDISCH_N_TAC 2 THEN CONV_TAC ARITH_CONV);
val left_t_le_next_t = save_top_thm "left_t_le_next_t";
```

left_t_run_t_step

Show that if left_t $(n) > 0$, then left_t $(n+1) =$ left_t $(n) - 1$ and run_t $(n+1) =$ run_t $(n) + 1$. The proof is by case analysis of the possible inputs, and `left_t_le_next_t` is used to show that in some of the cases, the hypotheses are contradictory. This lemma is used in the proof of `left_t_run_t_linear`.

```
g(`!inevseq n.
   ((left_t (OFP_Reaction inevseq n)) > 0) ==>
    (((left_t (OFP_Reaction inevseq (SUC n))) =
     ((left_t (OFP_Reaction inevseq n)) - 1)) /\
    ((run_t (OFP_Reaction inevseq (SUC n))) =
     (SUC (run_t (OFP_Reaction inevseq n)))))`);

e (REPEAT GEN_TAC);
e (reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC);

e (CONV_TAC ARITH_CONV);
e (CONV_TAC ARITH_CONV);
e (UNDISCH_N_TAC 2 THEN
   MP_TAC (SPEC_ALL left_t_le_next_t) THEN
   CONV_TAC ARITH_CONV);
e (UNDISCH_N_TAC 2 THEN
   MP_TAC (SPEC_ALL left_t_le_next_t) THEN
   CONV_TAC ARITH_CONV);
e (CONV_TAC ARITH_CONV);
e (CONV_TAC ARITH_CONV);
val left_t_run_t_step = save_top_thm "left_t_run_t_step";
```

left_t_run_t_linear

Show that if left_t $(n) = k$, left_t $(n+k) = 0$ and run_t $(n+k) =$ run_t $(n) + k$. The proof is by induction on k; the induction is arranged so that the variable n in the induction hypothesis will still be universally quantified. When proving the induction step, the induction hypothesis is instantiated with SUC n and `next_t_step` is used to show that the instantiated hypothesis implies the goal. This lemma is used in the proof of `trigger_meets_deadline`.

```
g(`!inevseq k n.
   ((left_t (OFP_Reaction inevseq n)) = k) ==>
    ((left_t (OFP_Reaction inevseq (n + k)) = 0) /\
    ((run_t (OFP_Reaction inevseq (n + k))) =
     (run_t (OFP_Reaction inevseq n)) + k))`);

 e (GEN_TAC);
 e (INDUCT_TAC);
 e (ARITH_REWRITE_TAC [--`n + 0 = n`--]);

 e (GEN_TAC);
 e (DISCH_TAC);
 e (ARITH_RES_TAC (--`(n = (SUC k)) ==> (n > 0)`--));
 e (IMP_RES_TAC (SPEC_ALL (theorem "-" "left_t_run_t_step")));
 e (ASSUM_LIST (fn thl =>
     ASSUME_TAC (REWRITE_RULE [el 2 thl]
        (SPEC (--`(SUC n)`--) (el 5 thl)))));
 e (ARITH_RES_TAC (--`(a = SUC b) ==> (a - 1 = b)`--));
 e (RES_TAC);
 e (ARITH_REWRITE_TAC [--`n + SUC k = SUC n + k`--]);
 e (ASM_REWRITE_TAC[]);
 val left_t_run_t_linear = save_top_thm "left_t_run_t_linear";
```

`next_t_step`

Show that if next_t $(n) > 0$, next_t $(n+1) =$ next_t $(n) - 1$. The proof is by case analysis on the possible inputs. This lemma is used in the proofs of `next_t_linear` and `run_t_finished`.

```
 g(`!inevseq n.
    ((next_t (OFP_Reaction inevseq n)) > 0) ==>
     ((next_t (OFP_Reaction inevseq (SUC n))) =
      ((next_t (OFP_Reaction inevseq n)) - 1))`);

 e (REPEAT STRIP_TAC);
 e (reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC);

 e (ASM_REWRITE_TAC[]);
 e (ASM_REWRITE_TAC[]);
 e (ARITH_RES_TAC (--`a > 0 ==> ~(a = 0)`--));
 e (ARITH_RES_TAC (--`a > 0 ==> ~(a = 0)`--));
 e (ASM_REWRITE_TAC[]);
 e (ASM_REWRITE_TAC[]);
 val next_t_step = save_top_thm "next_t_step";
```

next_t_linear

Show that for $k \leq \text{next\_t}\,(n)$, $\text{next\_t}\,(n+k) = \text{next\_t}\,(n) - k$. The proof is by induction on k; the induction is arranged so that the variable n in the induction hypothesis will still be universally quantified. When proving the induction step, the induction hypothesis is instantiated with SUC n and next_t_step is used to show that the instantiated hypothesis implies the goal. This lemma is used in the proof of trigger_meets_deadline.

```
g(`!inevseq k n.
  (k <= (next_t (OFP_Reaction inevseq n))) ==>
   ((next_t (OFP_Reaction inevseq (n + k))) =
    (next_t (OFP_Reaction inevseq n)) - k)`);

e (GEN_TAC);
e (INDUCT_TAC);
e (ARITH_REWRITE_TAC [--`n + 0 = n`--,--`n - 0 = n`--]);

e (GEN_TAC);
e (DISCH_TAC);
e (ARITH_RES_TAC (--`(SUC k <= a) ==> (k <= a)`--));
e (ARITH_RES_TAC (--`(SUC k <= a) ==> (a > 0)`--));
e (IMP_RES_TAC (SPEC_ALL (theorem "-" "next_t_step")));
e (ASSUM_LIST (fn thl =>
    ASSUME_TAC (REWRITE_RULE [el 1 thl]
       (SPEC (--`(SUC n)`--) (el 5 thl)))));
e (ARITH_RES_TAC (--`(SUC a <= b) ==> (a <= b - 1)`--));
e (RES_TAC);
e (ARITH_REWRITE_TAC [--`n + SUC k = SUC n + k`--]);
e (ASM_REWRITE_TAC[]);
e (CONV_TAC ARITH_CONV);
val next_t_linear = save_top_thm "next_t_linear";
```

trigger_init

Show that if a valid trigger request is received at time n, then $\text{next\_t}\,(n+1) = \text{Period}\,(1)-1$, $\text{left\_t}\,(n+1) = \text{Ctime}\,(1)-1$, and $\text{run\_t}\,(n+1) = \text{run\_t}\,(n)+1$. The proof is by analysis of the (single) input case. This lemma is used in the proof of trigger_meets_deadline.

```
g(`!^inevseq n.
   (((inevseq n) = Trigger) /\
    ((next_t (OFP_Reaction inevseq n)) = 0)) ==>
    (((next_t (OFP_Reaction inevseq (SUC n))) = (Period 1 - 1)) /\
     ((left_t (OFP_Reaction inevseq (SUC n))) = (Ctime 1 - 1)) /\
     ((run_t (OFP_Reaction inevseq (SUC n))) =
      (SUC (run_t (OFP_Reaction inevseq n)))))`);

e (REPEAT GEN_TAC THEN STRIP_TAC);
e (ASM_REWRITE_TAC [OFP_Reaction] THEN simplify_response_TAC);
e (MP_TAC Period_1_gt_0 THEN
   MP_TAC ok_processes_assum THEN
   CONV_TAC ARITH_CONV);
e (MP_TAC Period_1_gt_0 THEN
   MP_TAC ok_processes_assum THEN
   CONV_TAC ARITH_CONV);
e (RES_TAC);
e (RES_TAC);
val trigger_init = save_top_thm "trigger_init";
```

left_t_run_t_step1

Show that if left_t$(n) = 0$ and next_t$(n) > 0$, then left_t$(n+1) = 0$ and run_t$(n+1) =$ run_t(n). The proof is by case analysis of the possible inputs. This lemma is used in the proof of `run_t_finished`.

```
g(`!^inevseq n.
    (((left_t (OFP_Reaction inevseq n)) = 0) /\
     ((next_t (OFP_Reaction inevseq n)) > 0)) ==>
    (((left_t (OFP_Reaction inevseq (SUC n))) = 0) /\
     ((run_t (OFP_Reaction inevseq (SUC n))) =
      (run_t (OFP_Reaction inevseq n))))`);

e (GEN_TAC THEN GEN_TAC THEN STRIP_TAC);
e (reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC);

e (UNDISCH_N_TAC 3 THEN CONV_TAC ARITH_CONV);
e (UNDISCH_N_TAC 3 THEN CONV_TAC ARITH_CONV);
e (ARITH_RES_TAC (--`(a > 0) ==> ~(a = 0)`--));
e (ARITH_RES_TAC (--`(a > 0) ==> ~(a = 0)`--));
e (ARITH_RES_TAC (--`(a = 0) ==> ~(a > 0)`--) THEN ASM_REWRITE_TAC[]);
e (ARITH_RES_TAC (--`(a = 0) ==> ~(a > 0)`--) THEN ASM_REWRITE_TAC[]);
val left_t_run_t_step1 = save_top_thm "left_t_run_t_step1";
```

`run_t_finished`

Show that if $\text{left\_t}\,(n) = 0$ and $k \leq \text{next\_t}\,(n)$, then $\text{run\_t}\,(n+k) = \text{run\_t}\,(n)$, i.e., the amount of run time does not change after the trigger process is finished and before a new one can be started. The proof is by induction on k; the induction is arranged so that the variable `n` in the induction hypothesis will still be universally quantified. When proving the induction step, the induction hypothesis is instantiated with `SUC n`, and `next_t_step` and `left_t_run_t_step1` are used to show that the instantiated hypothesis implies the goal. The lemma is used in the proof of `trigger_meets_deadline`.

```
g(`!~inevseq k n.
   (((left_t (OFP_Reaction inevseq n)) = 0) /\
    (k <= (next_t (OFP_Reaction inevseq n)))) ==>
   ((run_t (OFP_Reaction inevseq (n + k))) =
    (run_t (OFP_Reaction inevseq n)))`);

e (GEN_TAC THEN INDUCT_TAC);
e (ARITH_REWRITE_TAC [--`n + 0 = n`--]);

e (REPEAT STRIP_TAC);
e (ARITH_RES_TAC (--`(SUC a <= b) ==> (a <= b - 1)`--));
e (ARITH_RES_TAC (--`(SUC a <= b) ==> (b > 0)`--));
e (IMP_RES_TAC next_t_step);
e (IMP_RES_TAC left_t_run_t_step1);
e (ASSUM_LIST (fn thl =>
               IMP_RES_TAC (REWRITE_RULE [el 1 thl, el 2 thl, el 3 thl]
                            (SPEC (--`SUC n`--) (el 8 thl)))));
e (ARITH_REWRITE_TAC [--`n + SUC k = SUC n + k`--]);
e (ASM_REWRITE_TAC[]);
val run_t_finished = save_top_thm "run_t_finished";
```

`trigger_meets_deadline`

We now have the lemmas we need to prove the trigger process requirement. The lemma `trigger_init` is used to establish the initial conditions, the lemma `left_t_run_t_linear` is used to show $\text{left\_t}\,(n + \text{Ctime}\,(1)) = 0$ and $\text{run\_t}\,(n + \text{Ctime}\,(1)) = \text{Ctime}\,(1)$, the lemma `next_t_linear` is used to show that $\text{next\_t}\,(n + \text{Ctime}\,(1)) = \text{Period}\,(1) - \text{Ctime}\,(1)$, The lemma `run_t_finished` is then instantiated with $\text{Deadline}\,(1) - \text{Ctime}\,(1)$ and $n + \text{Ctime}\,(1)$ to show that $\text{run\_t}\,(n + \text{Deadline}\,(1)) = \text{run\_t}\,(n) + \text{Ctime}\,(1)$, which can then be used to show that the goal is true.

```
g(`!^inevseq n.
   (((inevseq n) = Trigger) /\
    (next_t (OFP_Reaction inevseq n) = 0)) ==>
   ((run_t (OFP_Reaction inevseq (n + Deadline 1))) -
    (run_t (OFP_Reaction inevseq n)) = (Ctime 1))`);

e (REPEAT STRIP_TAC);
e (STRIP_ASSUME_TAC ok_processes_assum);
e (IMP_RES_TAC trigger_init);
e (IMP_RES_TAC left_t_run_t_linear);
e (ARITH_RES_TAC (--`((Ctime 1) > 0) ==>
                     ((SUC n + ((Ctime 1) - 1)) = (n + (Ctime 1)))`--));
e (UNDISCH_N_TAC 2 THEN UNDISCH_N_TAC 2 THEN ASM_REWRITE_TAC[] THEN
   REPEAT STRIP_TAC);
e (ARITH_RES_TAC
   (--`(((next_t (OFP_Reaction inevseq (SUC n))) = (Period 1) - 1) /\
        ((Period 1) > 1) /\
        ((Period 1) >= (Deadline 1)) /\
        ((Deadline 1) >= (Ctime 1))) ==>
       ((Ctime 1) - 1 <= (next_t (OFP_Reaction inevseq (SUC n))))`--));
e (IMP_RES_TAC next_t_linear);
e (ARITH_RES_TAC (--`((Period 1 >= Deadline 1) /\
                      (Deadline 1 >= Ctime 1))
                      ==> (Period 1 >= Ctime 1)`--));
e (ASSUME_TAC Period_1_gt_0);
e (ARITH_RES_TAC (--`((0 < (Period 1)) /\
                      ((Ctime 1) > 0) /\
                      ((Period 1) >= (Ctime 1))) ==>
                     ((((Period 1) - 1) - ((Ctime 1) - 1)) =
                      (Period 1) - (Ctime 1))`--));
e (UNDISCH_N_TAC 4 THEN ASM_REWRITE_TAC[] THEN DISCH_TAC);
e (ARITH_RES_TAC (--`(((Deadline 1) >= (Ctime 1)) /\
                      ((Period 1) >= (Ctime 1)) /\
                      ((Period 1) >= (Deadline 1))) ==>
                     (((Deadline 1) - (Ctime 1)) <=
                      ((Period 1) - (Ctime 1)))`--));
e (ARITH_RES_TAC (--`((Deadline 1) >= (Ctime 1)) ==>
                      ((n + (Ctime 1)) + ((Deadline 1) - (Ctime 1)) =
                       n + (Deadline 1))`--));
e (MP_TAC (SPECL [--`^inevseq`--,
                  --`Deadline 1 - Ctime 1`--,--`n + Ctime 1`--]
                  run_t_finished));
e (ASM_REWRITE_TAC[]);
e (DISCH_TAC THEN ASM_REWRITE_TAC[]);
```

```
e (CONV_TAC ARITH_CONV);
val trigger_meets_deadline = save_top_thm "trigger_meets_deadline";
```

8.4.3 Updater Requirement Proof

The proof that the updater process requirement is met is much more complicated than that for the trigger process, for the following reasons:

- The updater process has a lower priority than the trigger process, so it is necessary to show that even if the trigger process uses as much CPU time as it can, the updater process will still meet its deadline.

- The updater process is periodic, and so the requirement must be proved for all periods:

  $$[n \cdot \text{Period}\,(2) \ldots n \cdot \text{Period}\,(2) + k], \quad k \leq \text{Period}\,(2)$$

  It turns out that the proof must be done by double induction on n and k, where k is shown to be Deadline (2).

- The behavior of most of the relevant state variables as a function of time is quite irregular. When the behavior of a state variable or some function of state variables *is* regular, it frequently turns out to be periodic, which means that the DIV and MOD functions must be used to describe it. DIV and MOD are nonlinear functions and are not well-supported in HOL — there are few available theorems about them and the `arith` library can't handle them.

- The only approach we were able to develop is very messy, with a lot of algebraic manipulation.

The proof of the updater requirement is based on the observation that the run time of the updater process in an interval $[a \ldots b]$ is

$$\text{run\_u}\,(b) - \text{run\_u}\,(a) \tag{8.1}$$

In this same interval, the amount of time the trigger process will run is:

$$\text{run\_t}\,(b) - \text{run\_t}\,(a)$$

and the amount of time available for the updater process to run is

$$(b - a) - (\text{run\_t}\,(b) - \text{run\_t}\,(a)) \tag{8.2}$$

In the remainder of this discussion, we will abbreviate Period (2) by P, Deadline (2) by D, and Ctime (2) by C.

For the interval $[nP \ldots nP + k]$, where $k \leq P$, the maximum amount of time the updater process will run is C, as long as the process in the previous interval finished at or before the end of the previous interval. The time available for the updater process to run is:

$$k - (\text{run\_t}\,(nP + k) - \text{run\_t}\,(nP))$$

and the amount of time the updater process will actually run is:

$$\min\,(C, k - (\text{run\_t}\,(nP + k) - \text{run\_t}\,(nP)))$$

Thus, it should be true that:
1.

$$\text{left\_u}\,(nP) = 0 \tag{8.3}$$

and
2.

$$\text{run\_u}\,(nP + k) - \text{run\_u}\,(nP) = \min\,(C, k - (\text{run\_t}\,(nP + k) - \text{run\_t}\,(nP))) \tag{8.4}$$

for all n, k, where $k \leq P$.

When we have shown this, we can then substitute D for k in equation 8.4, and get

$$\text{run\_u}\,(nP + D) - \text{run\_u}\,(nP) = \min\,(C, D - (\text{run\_t}\,(nP + D) - \text{run\_t}\,(nP))) \tag{8.5}$$

In the worst case where all input events are trigger requests, $\text{run\_t}\,(n) = \text{CPU}\,(n, 1)$. Thus,

$$\text{run\_t}\,(nP + k) - \text{run\_t}\,(nP) = \text{CPU}\,(nP + k, 1) - \text{CPU}\,(nP, 1) \tag{8.6}$$

And, from [23],

$$\text{CPU}\,(nP + k, 1) - \text{CPU}\,(nP, 1) \leq \text{CPU}\,(k, 1) \tag{8.7}$$

Thus,

$$\text{run\_t}\,(nP+k) - \text{run\_t}\,(nP) \leq \text{CPU}\,(k,1) \tag{8.8}$$

Now, one of the process parameter restrictions is:

$$D \geq C + \text{CPU}\,(D,1)$$

which can be rearranged to be

$$D - \text{CPU}\,(D,1) \geq C \tag{8.9}$$

If we substitute D for k in inequality 8.8 we get:

$$\text{run\_t}\,(nP+D) - \text{run\_t}\,(nP) \leq \text{CPU}\,(D,1) \tag{8.10}$$

From inequalities 8.9 and 8.10 we get

$$D - (\text{run\_t}\,(nP+D) - \text{run\_t}\,(nP)) \geq C \tag{8.11}$$

and therefore, the right side of equation 8.5 is just C and we get:

$$\text{run\_u}\,(nP+D) - \text{run\_u}\,(nP) = C$$

which is the updater process requirement.

If not all input events are trigger requests, then the trigger process run time in an interval is less than or equal to that in the worst case, so inequality 8.10 still holds.

Case Analysis Predicates

The following three predicates are used for case analysis of runnable processes in some of the proofs that follow. `(trigger_ready inevseq n)` states the conditions under which the trigger process will be run at time n: the trigger process will be run if $\text{left\_t}\,(n) > 0$ (the trigger process is ready to run) or if $\text{next\_t}\,(n) = 0$ (a trigger request is acceptable) and the input event at time n is `Trigger`. `(updater_ready inevseq n)` states the conditions under which the updater process will be run at time n, if `trigger_ready` is false: the updater process will be run if $\text{left\_u}\,(n) > 0$ (the updater process is ready to run) or if $\text{period\_u}\,(n) = 0$ (a new updater process is starting). Thus a case analysis can be done on the following three conditions:

1. (trigger_ready inevseq n)
2. ~(trigger_ready inevseq n) /\bs (updater_ready inevseq n)
3. ~(trigger_ready inevseq n) /\bs ~(updater_ready inevseq n)

The lemma not_Trigger shows that if an input event is not Trigger, it must be Tick (the only other possibility).

```
val trigger_ready =
new_definition (
 "trigger_ready",
 --`trigger_ready inevseq n =
    (((left_t (OFP_Reaction inevseq n)) > 0) \/
     (((next_t (OFP_Reaction inevseq n)) = 0) /\
        ((inevseq n) = Trigger)))`--);

val updater_ready =
new_definition (
 "updater_ready",
 --`updater_ready inevseq n =
     ((left_u (OFP_Reaction inevseq n)) > 0) \/
      ((period_u (OFP_Reaction inevseq n)) = 0)`--);

val not_Trigger =
 store_thm (
  "not_Trigger",
   --`!x:OFP_InEv. ~(x = Trigger) = (x = Tick)`--,
   INDUCT_THEN (theorem "-" "OFP_InEv_Induct") ASSUME_TAC THEN
   REWRITE_TAC [theorem "-" "OFP_InEv_Distinct"]);
```

trigger_ready_result

Show that if the trigger process is ready to run at time n, then the run time of the trigger process at time $n+1$ is incremented and the run time of the updater process at time $n+1$ is unchanged. The proof is by case analysis of the antecedent and the possible inputs. This lemma is used in runnable process case analysis in the lemmas run_u_mono, run_t_mono, run_t_diff, n_ge_run_t, and updater_meets_deadline_0.

```
g(`!^inevseq n.
   (trigger_ready inevseq n) ==>
   (((run_t (OFP_Reaction inevseq (SUC n))) =
      (SUC (run_t (OFP_Reaction inevseq n)))) /\
     ((run_u (OFP_Reaction inevseq (SUC n))) =
      (run_u (OFP_Reaction inevseq n))))`);

  e (REWRITE_TAC [trigger_ready]);
  e (REPEAT GEN_TAC THEN STRIP_TAC);
  e (reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC);

  e (MP_TAC ok_processes_assum THEN
        UNDISCH_N_TAC 2 THEN CONV_TAC ARITH_CONV);
  e (UNDISCH_N_TAC 2 THEN CONV_TAC ARITH_CONV);
  e (MP_TAC ok_processes_assum THEN CONV_TAC ARITH_CONV);
  e (MP_TAC ok_processes_assum THEN CONV_TAC ARITH_CONV);
  e (MP_TAC ok_processes_assum THEN
        UNDISCH_N_TAC 3 THEN CONV_TAC ARITH_CONV);
  e (UNDISCH_N_TAC 3 THEN CONV_TAC ARITH_CONV);

  e (ASM_REWRITE_TAC [OFP_Reaction] THEN simplify_response_TAC);
  e (MP_TAC ok_processes_assum THEN CONV_TAC ARITH_CONV);
  e (MP_TAC ok_processes_assum THEN CONV_TAC ARITH_CONV);
  e (RES_TAC);
  e (RES_TAC);
  val trigger_ready_result = save_top_thm "trigger_ready_result";
```

`trigger_not_ready_result`

Show that if the trigger process is not ready to run at time n, then the run time of the trigger process at time $n + 1$ is unchanged. The proof is by case analysis of the antecedent and the possible inputs. This lemma is used in runnable process case analysis in the lemmas `run_u_mono`, `run_t_mono`, `run_t_diff`, `n_ge_run_t`, and `updater_meets_deadline_0`.

```
  g(`!^inevseq n.
    ~(trigger_ready inevseq n) ==>
      ((run_t (OFP_Reaction inevseq (SUC n))) =
       (run_t (OFP_Reaction inevseq n)))`);

  e (REWRITE_TAC [trigger_ready]);
  e (PTAUT_REWRITE_TAC [--`~(a \/ (b /\ c)) = (~a /\ (~b \/ ~c))`--]);
  e (REWRITE_TAC [not_Trigger]);
  e (REPEAT STRIP_TAC);

  e (reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC);
```

```
e (ASM_REWRITE_TAC[]);
e (ASM_REWRITE_TAC[]);
e (RES_TAC);
e (RES_TAC);
e (ASM_REWRITE_TAC[]);
e (ASM_REWRITE_TAC[]);

e (ASM_REWRITE_TAC [OFP_Reaction] THEN simplify_response_TAC THEN
   ASM_REWRITE_TAC[]);
val trigger_not_ready_result = save_top_thm "trigger_not_ready_result";
```

updater_ready_result

Show that if the trigger process is not ready to run at time n and the updater process is, then the run time of the updater process at time $n+1$ is incremented. The proof is by case analysis of the antecedent and the possible inputs. This lemma is used in runnable process case analysis in the lemmas run_u_mono and updater_meets_deadline_0.

```
g(`!^inevseq n.
((updater_ready inevseq n) /\ ~(trigger_ready inevseq n)) ==>
 ((run_u (OFP_Reaction inevseq (SUC n))) =
  (SUC (run_u (OFP_Reaction inevseq n))))`);

e (REWRITE_TAC [updater_ready, trigger_ready]);
e (PTAUT_REWRITE_TAC [--`~(a \/ (b /\ c)) = (~a /\ (~b \/ ~c))`--]);
e (REWRITE_TAC [not_Trigger]);
e (REPEAT STRIP_TAC);

e (reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC);
e (MP_TAC ok_processes_assum THEN
      UNDISCH_N_TAC 3 THEN CONV_TAC ARITH_CONV);
e (UNDISCH_N_TAC 4 THEN UNDISCH_N_TAC 3 THEN CONV_TAC ARITH_CONV);
e (RES_TAC);
e (RES_TAC);
e (MP_TAC ok_processes_assum THEN UNDISCH_N_TAC 4 THEN UNDISCH_N_TAC 4
   THEN CONV_TAC ARITH_CONV);
e (MP_TAC ok_processes_assum THEN UNDISCH_N_TAC 4 THEN UNDISCH_N_TAC 4
   THEN CONV_TAC ARITH_CONV);

e (ASM_REWRITE_TAC [OFP_Reaction] THEN simplify_response_TAC);
e (MP_TAC ok_processes_assum THEN UNDISCH_N_TAC 3 THEN UNDISCH_N_TAC 3
   THEN CONV_TAC ARITH_CONV);
e (MP_TAC ok_processes_assum THEN UNDISCH_N_TAC 3 THEN UNDISCH_N_TAC 3
   THEN CONV_TAC ARITH_CONV);

e (reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC);
e (MP_TAC ok_processes_assum THEN UNDISCH_N_TAC 3 THEN
     CONV_TAC ARITH_CONV);
e (RES_TAC);
e (RES_TAC);
e (MP_TAC ok_processes_assum THEN UNDISCH_N_TAC 3 THEN
     CONV_TAC ARITH_CONV);

e (ASM_REWRITE_TAC [OFP_Reaction] THEN simplify_response_TAC);
e (MP_TAC ok_processes_assum THEN UNDISCH_N_TAC 3 THEN
     CONV_TAC ARITH_CONV);
e (RES_TAC);
val updater_ready_result = save_top_thm "updater_ready_result";
```

updater_not_ready_result

Show that if the updater process is not ready to run at time n, then the run time of the updater process at time $n+1$ is unchanged. The proof is by case analysis of the antecedent and the possible inputs. This lemma is used in runnable process case analysis in the lemmas run_u_mono and updater_meets_deadline_0.

```
g(`!^inevseq n.
   ~(updater_ready inevseq n) ==>
     ((run_u (OFP_Reaction inevseq (SUC n))) =
      (run_u (OFP_Reaction inevseq n)))`);

e (REWRITE_TAC [updater_ready]);
e (PTAUT_REWRITE_TAC [--`~(a \/ b) = (~a /\ ~b)`--]);
e (REPEAT STRIP_TAC);
e (reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC);
e (RES_TAC);
e (ASM_REWRITE_TAC[]);
e (ASM_REWRITE_TAC[]);
e (ASM_REWRITE_TAC[]);
val updater_not_ready_result = save_top_thm "updater_not_ready_result";
```

period_u_desc

Describe the behavior of the period_u state variable. Show that period_u (0) = 0, and that if period_u $(n) = 0$ then period_u $(n+1) =$ Period $(2) - 1$, and period_u $(n) - 1$ otherwise. The first case follows from the value of the initial state, and the proof of the second case is by case analysis of the possible inputs. This lemma is used in the proof of period_u_formula.

```
g(`((period_u (OFP_Reaction inevseq 0)) = 0) /\
   ((period_u (OFP_Reaction inevseq (SUC n))) =
    ((period_u (OFP_Reaction inevseq n)) = 0 =>
     (Period 2) - 1 |
     (period_u (OFP_Reaction inevseq n)) - 1))`);

e (REPEAT STRIP_TAC);
e (REWRITE_TAC ([OFP_Reaction, OFP_InitParam] @ state_field_defs));
e (reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC THEN
   ASM_REWRITE_TAC[]);
val period_u_desc = save_top_thm "period_u_desc";
```

period_u_formula

Show that the value of the period_u state variable as a function of n is 0 if n is a multiple of Period (2), and Period $(2) - n$ MOD Period (2) otherwise. The proof is by induction on n, using lemma period_u_desc to characterize the value of the state variable. The arith library cannot handle MOD, so there is much low-level work to do. This lemma is used in the proof of period_u_desc.

```
g(`!n.
   (period_u (OFP_Reaction inevseq n)) =
    (((n MOD (Period 2)) = 0) => 0 |
      ((Period 2) - (n MOD (Period 2))))`);

e (ASSUME_TAC Period_2_gt_0);
e (INDUCT_TAC);
e (COND_REWRITE_TAC (theorem "arithmetic" "ZERO_MOD"));
e (REWRITE_TAC ([OFP_Reaction, OFP_InitParam] @ state_field_defs));

e (ASM_REWRITE_TAC [period_u_desc]);
e (COND_REWRITE_TAC MOD_SUC);
e (IMP_RES_TAC (definition "arithmetic" "DIVISION"));
e (ASSUM_LIST (fn thl => (MP_TAC (SPEC (--`n:num`--) (el 1 thl)))));
```

```
e (ASM_CASES_TAC (--`n MOD (Period 2) = 0`--));

e (POP_ASSUM SUBST1_TAC THEN REWRITE_TAC[]);
e (STRIP_ASSUME_TAC ok_processes_assum);
e (ARITH_COND_REWRITE_TAC (--`Period 2 > 1 ==> ~(0 = Period 2 - 1)`--));
e (CONV_TAC ARITH_CONV);

e (POP_ASSUM (SUBST1_TAC o EQF_INTRO) THEN REWRITE_TAC[]);
e (STRIP_TAC);
e (ARITH_COND_REWRITE_TAC (--`(a MOD b < b) ==> ~(b - a MOD b = 0)`--));

e (ASM_CASES_TAC (--`n MOD (Period 2) = (Period 2) - 1`--));

e (POP_ASSUM SUBST1_TAC THEN REWRITE_TAC[]);
e (CONV_TAC ARITH_CONV);

e (POP_ASSUM (SUBST1_TAC o EQF_INTRO) THEN REWRITE_TAC[]);
e (REWRITE_TAC [(theorem "num" "NOT_SUC")]);
e (CONV_TAC ARITH_CONV);
val period_u_formula = save_top_thm "period_u_formula";
```

period_u_0

Show that period_u $(n) = 0$ if and only if n is a multiple of Period (2). The proof uses period_u_formula and properties of MOD. This lemma is used in the proofs of new_run_u_left_u_sum, run_u_left_u_sum, left_u_finished, updater_ready_result1, and updater_not_ready_result1.

```
g(`!^inevseq n.
   ((n MOD (Period 2)) = 0) =
      ((period_u (OFP_Reaction inevseq n)) = 0)`);

e (REPEAT GEN_TAC);
e (ASSUME_TAC Period_2_gt_0);
e (SUBST1_TAC (SPEC_ALL (theorem "-" "period_u_formula")));
e (ASM_CASES_TAC (--`n MOD (Period 2) = 0`--));
e (ASM_REWRITE_TAC[]);

e (ASM_REWRITE_TAC[]);
e (IMP_RES_TAC (SPEC (--`(Period 2)`--)
                  (definition "arithmetic" "DIVISION")));
e (POP_ASSUM (MP_TAC o (SPEC (--`n:num`--))));
```

```
e (CONV_TAC ARITH_CONV);
val period_u_0 = save_top_thm "period_u_0";
```

new_run_u_left_u_sum

Show that when n is a multiple of Period (2), the sum run_u $(n+1)$ + left_u $(n+1)$ is Ctime (2) more than the sum in the previous period. Lemma period_u_0 is used to show that a new period starts at time n, and the proof then follows from case analysis of the possible inputs. This lemma is used in the proof of run_u_left_u_sum.

```
g(`!^inevseq n. ((n MOD (Period 2)) = 0) ==>
   (((run_u (OFP_Reaction inevseq (SUC n))) +
     (left_u (OFP_Reaction inevseq (SUC n)))) =
    ((run_u (OFP_Reaction inevseq n)) +
     (left_u (OFP_Reaction inevseq n)) + (Ctime 2)))`);

e (REPEAT STRIP_TAC);
e (ASSUME_TAC (SPEC_ALL period_u_0));
e (RES_TAC);

e (reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC);
e (MP_TAC ok_processes_assum THEN CONV_TAC ARITH_CONV);
e (RES_TAC);
e (MP_TAC ok_processes_assum THEN CONV_TAC ARITH_CONV);
e (MP_TAC ok_processes_assum THEN CONV_TAC ARITH_CONV);
val new_run_u_left_u_sum = save_top_thm "new_run_u_left_u_sum";
```

run_u_left_u_sum

Show that the sum run_u $(n+1)$+left_u $(n+1)$ is $(n \text{DIV} \text{Period}(2)) \cdot \text{Ctime}(2)$ at the beginning of a period, and $((n \text{DIV} \text{Period}(2))+1) \cdot \text{Ctime}(2)$ elsewhere. The proof is by induction on n. The proof of the induction step is done by analysis of the following three possibilities:

1. Time n is at the end of a period: $n \texttt{ MOD } \text{Period}(2) = \text{Period}(2) - 1$.

2. Time n is at the beginning of a period: $n \texttt{ MOD } \text{Period}(2) = 0$.

3. Time n is in the middle of a period (neither of the above conditions is true).

The proof of each case is by case analysis of the possible inputs. Lemma period_u_0 is used to show contradictory hypotheses where appropriate, and lemma
new_run_u_left_u_sum is used in the second case. This result is used in the proof of run_u_left_u_sum1.

```
g(`!^inevseq n.
    ((run_u (OFP_Reaction inevseq n)) +
        (left_u (OFP_Reaction inevseq n))) =
    ((n MOD (Period 2) = 0) =>
        (n DIV (Period 2)) * (Ctime 2) |
        (SUC (n DIV (Period 2))) * (Ctime 2))`);

e (GEN_TAC);
e (INDUCT_TAC);
e (REWRITE_TAC ([OFP_Reaction, OFP_InitParam] @ state_field_defs));
e (ASSUME_TAC Period_2_gt_0);
e (COND_REWRITE_TAC (theorem "arithmetic" "ZERO_DIV"));
e (COND_REWRITE_TAC (theorem "arithmetic" "ZERO_MOD"));
e (CONV_TAC ARITH_CONV);

e (ASM_CASES_TAC (--`(n MOD Period 2) = ((Period 2) - 1)`--));

e (UNDISCH_N_TAC 2);
e (STRIP_ASSUME_TAC ok_processes_assum);
e (ARITH_RES_TAC (--`(a = b - 1) ==> (b > 1) ==> ~(a = 0)`--));
e (ASSUME_TAC Period_2_gt_0);
e (COND_REWRITE_TAC DIV_SUC);
e (COND_REWRITE_TAC MOD_SUC);
e (ASM_REWRITE_TAC[]);

e (reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC);
e (IMP_RES_TAC period_u_0 THEN RES_TAC);
e (CONV_TAC ARITH_CONV);
e (IMP_RES_TAC period_u_0 THEN RES_TAC);
e (CONV_TAC ARITH_CONV);
e (IMP_RES_TAC period_u_0 THEN RES_TAC);
e (CONV_TAC ARITH_CONV);

e (ASM_CASES_TAC (--`n MOD Period 2 = 0`--));
e (UNDISCH_N_TAC 3);
e (ASSUME_TAC Period_2_gt_0);
e (COND_REWRITE_TAC DIV_SUC);
e (COND_REWRITE_TAC MOD_SUC);
```

```
e (ASM_REWRITE_TAC[]);
e (ARITH_REWRITE_TAC [--`((SUC 0) = 0) = F`--]);
e (COND_REWRITE_TAC new_run_u_left_u_sum);
e (REWRITE_TAC [definition "arithmetic" "MULT"]);
e (CONV_TAC ARITH_CONV);

e (UNDISCH_N_TAC 3);
e (ASSUME_TAC Period_2_gt_0);
e (COND_REWRITE_TAC DIV_SUC);
e (COND_REWRITE_TAC MOD_SUC);
e (ASM_REWRITE_TAC[]);
e (ARITH_REWRITE_TAC [--`((SUC a) = 0) = F`--]);
e (reaction_cases_TAC (--`^inevseq n`--) THEN simplify_response_TAC);

e (IMP_RES_TAC period_u_0 THEN RES_TAC);
e (CONV_TAC ARITH_CONV);
e (IMP_RES_TAC period_u_0 THEN RES_TAC);
e (CONV_TAC ARITH_CONV);
e (IMP_RES_TAC period_u_0 THEN RES_TAC);
e (CONV_TAC ARITH_CONV);
val run_u_left_u_sum = save_top_thm "run_u_left_u_sum";
```

run_u_left_u_sum1

Show that at the middle or end of the nth period, the sum of run_u and left_u is $(n+1) \cdot \mathrm{Ctime}(2)$. The proof is by analysis of the two cases of whether $k = \mathrm{Period}(2)$ (the end of the period) or not (the middle of the period), and follows from lemma run_u_left_u_sum. This result is used in the proofs of run_u_finished, updater_ready_result1, updater_not_ready_-result1, and suc_period_u_ctime_u.

```
g(`!^inevseq n k.
    ((k > 0) /\ (k <= Period 2)) ==>
     ((run_u (OFP_Reaction inevseq (n * Period 2 + k))) +
      (left_u (OFP_Reaction inevseq (n * Period 2 + k))) =
     (SUC n) * (Ctime 2))`);

e (REPEAT STRIP_TAC);
e (MP_TAC (SPECL [--`^inevseq`--,--`(n * Period 2 + k)`--]
               run_u_left_u_sum));
e (ASM_CASES_TAC (--`k = Period 2`--));

e (ASM_REWRITE_TAC[]);
```

```
e (REWRITE_TAC [(SYM (SPECL [--`n:num`--,--`Period 2`--]
                    (CONJUNCT2 (definition "arithmetic" "MULT"))))]);
e (ASSUME_TAC Period_2_gt_0);
e (COND_REWRITE_TAC (theorem "arithmetic" "MOD_EQ_0"));
e (IMP_RES_TAC (theorem "arithmetic" "DIV_MULT"));
e (POP_ASSUM (MP_TAC o (SPEC (--`SUC n`--))));
e (ARITH_REWRITE_TAC [--`a + 0 = a`--] THEN DISCH_TAC THEN
     ASM_REWRITE_TAC[]);

e (ARITH_RES_TAC (--`((a <= b) /\ ~(a = b)) ==> (a < b)`--));
e (COND_REWRITE_TAC MOD_NONZERO);
e (COND_REWRITE_TAC (theorem "arithmetic" "DIV_MULT"));
val run_u_left_u_sum1 = save_top_thm "run_u_left_u_sum1";
```

left_u_finished

Show that if `left_u` is 0 in a period, it will remain 0 until the end of the period. The proof is by induction on the remaining time in the period and case analysis of the possible inputs, and uses `period_u_0` to show that the hypotheses of some of the cases are contradictory. This result is used in the proof of `run_u_finished`.

```
g(`!^inevseq n k k'.
   (((left_u (OFP_Reaction inevseq ((n * (Period 2)) + k))) = 0) /\
    (k > 0) /\
    (k + k' <= (Period 2))) ==>
   ((left_u (OFP_Reaction inevseq ((n * (Period 2)) + k + k'))) = 0)`);

e (GEN_TAC THEN GEN_TAC THEN GEN_TAC THEN INDUCT_TAC THEN STRIP_TAC);

e (ARITH_REWRITE_TAC [--`k + 0 = k`--] THEN ASM_REWRITE_TAC[]);

e (ARITH_REWRITE_TAC [--`(n * Period 2 + k + SUC k') =
                         (SUC (n * Period 2 + k + k'))`--]);
e (ARITH_RES_TAC (--`(k + SUC k' <= Period 2) ==>
                        (k + k' <= Period 2)`--));
e (ARITH_RES_TAC (--`(k + SUC k' <= Period 2) ==>
                        (k + k' < Period 2)`--));
e (ARITH_RES_TAC (--`k > 0 ==> k + k' > 0`--));
e (POP_ASSUM (ASSUME_TAC o SPEC_ALL));
e (RES_TAC);
e (reaction_cases_TAC (--`^inevseq (n * Period 2 + k + k')`--) THEN
   simplify_response_TAC);
```

```
e (IMP_RES_TAC MOD_NONZERO THEN IMP_RES_TAC period_u_0 THEN RES_TAC);
e (UNDISCH_N_TAC 2 THEN CONV_TAC ARITH_CONV);
e (IMP_RES_TAC MOD_NONZERO THEN IMP_RES_TAC period_u_0 THEN RES_TAC);
e (UNDISCH_N_TAC 3 THEN CONV_TAC ARITH_CONV);
e (IMP_RES_TAC MOD_NONZERO THEN IMP_RES_TAC period_u_0 THEN RES_TAC);
e (UNDISCH_N_TAC 3 THEN CONV_TAC ARITH_CONV);
val left_u_finished = save_top_thm "left_u_finished";
```

run_u_finished

Show that if left_u is 0 in a period, run_u will be unchanged until the end of the period. The proof uses left_u_finished to show that the value of left_u is zero at the end of the period, and run_u_left_u_sum1 to show that the value of run_u is therefore the same at the end of the period. This result is used in the proof of suc_period_u_ctime_u.

```
g(`!^inevseq n k k'.
  (((left_u (OFP_Reaction inevseq ((n * (Period 2)) + k))) = 0) /\
   (k > 0) /\
   (k + k' <= (Period 2))) ==>
  ((run_u (OFP_Reaction inevseq ((n * (Period 2)) + k + k'))) =
   (run_u (OFP_Reaction inevseq ((n * (Period 2)) + k))))`);

e (REPEAT STRIP_TAC);
e (ARITH_RES_TAC (--`(k + k' <= Period 2) ==> (k <= Period 2)`--));
e (ARITH_RES_TAC (--`(k > 0) ==> (k + k' > 0)`--));
e (POP_ASSUM (ASSUME_TAC o SPEC_ALL));
e (IMP_RES_TAC run_u_left_u_sum1);
e (POP_ASSUM (MP_TAC o SPEC_ALL));
e (POP_ASSUM (MP_TAC o SPEC_ALL));
e (IMP_RES_TAC left_u_finished);
e (ASM_REWRITE_TAC[]);
e (CONV_TAC ARITH_CONV);
val run_u_finished = save_top_thm "run_u_finished";
```

run_t_le_CPU

Show that the amount of time the trigger process runs in an interval $n \ldots n+k$ is less than or equal to CPU $(k, 1)$. There was not enough time to prove this lemma, so it is presented as an axiom. An informal argument that this assertion is true is the following.

There are three possible cases:

1. $k \leq \text{Ctime}(1)$.

   Then, $\text{CPU}(k, 1) = k$ and $\text{run\_t}(k) \leq k$, since a process cannot run longer than the available time.

2. $\text{Ctime}(1) < k \leq \text{Period}(1)$.

   Then, $\text{CPU}(k, 1) = \text{Ctime}(1)$. In this interval, the run time cannot be greater than Ctime (1), because in the worst case, where all input events are `Trigger`, the runs of the trigger process are still spaced Period (1) time units apart. If not all input events are `Trigger`, the spacing between runs may be even wider, resulting in even less run time in the interval.

3. $k > \text{Period}(1)$.

   Then, k will contain $k\,\texttt{DIV}\,\text{Period}(1)$ periods, and in each period, the run time will be less than or equal to Ctime (1). Also,

   $$\text{CPU}((k\,\texttt{DIV}\,\text{Period}(1)) \cdot \text{Period}(1), 1) = (k\,\texttt{DIV}\,\text{Period}(1)) \cdot \text{Ctime}(1)$$

   In the remaining partial period, one of the above conditions will be true, so the run time in that partial period will be less than or equal to the value of CPU, and thus the run time in the entire interval will be less than or equal to $\text{CPU}(k, 1)$.

This result is used in the proof of `min_deadline`.

```
val run_t_le_CPU =
 new_open_axiom (
  "run_t_le_CPU",
  --`!n k. ((run_t (OFP_Reaction inevseq (n + k))) -
      (run_t (OFP_Reaction inevseq n))) <= (CPU k 1)`--);
```

`min_deadline`

Show that the amount of time available for the updater process to run in the interval $[n \ldots n + \text{Deadline}(2)]$ is greater than or equal to Ctime (2), and so the minimum of the two values is Ctime (2). The proof follows from lemma `run_t_le_CPU`, by subgoaling on `MIN_LE`. This result is used in the proofs of `suc_period_u_ctime_u` and `updater_meets_deadline`.

```
g(`!n.
   (MIN (Ctime 2)
    ((Deadline 2) -
     ((run_t (OFP_Reaction inevseq (n + (Deadline 2)))) -
      (run_t (OFP_Reaction inevseq n))))) =
   (Ctime 2)`);

e (GEN_TAC);
e (MATCH_MP_TAC MIN_LE);
e (MP_TAC (SPECL [--`n:num`--,--`Deadline 2`--] run_t_le_CPU));
e (MP_TAC ok_processes_assum);
e (CONV_TAC ARITH_CONV);
val min_deadline = save_top_thm "min_deadline";
```

run_u_mono

Show that run_u$(n+k) \geq$ run_u(n). The proof is by induction on k, and the induction step is proved by runnable process case analysis. This result is used by updater_ready_result1, suc_period_u_ctime_u, and updater_meets_deadline_0.

```
g(`!^inevseq n k.
   (run_u (OFP_Reaction inevseq (n + k))) >=
    (run_u (OFP_Reaction inevseq n))`);

e (GEN_TAC THEN GEN_TAC THEN INDUCT_TAC);

e (ARITH_REWRITE_TAC [--`n + 0 = n`--] THEN CONV_TAC ARITH_CONV);

e (ARITH_REWRITE_TAC [--`n + SUC k = SUC (n + k)`--]);
e (ASM_CASES_TAC (--`trigger_ready inevseq (n + k)`--) THENL
   [ALL_TAC,
    ASM_CASES_TAC (--`updater_ready inevseq (n + k)`--)]);

e (COND_REWRITE_TAC trigger_ready_result THEN ASM_REWRITE_TAC[]);

e (COND_REWRITE_TAC trigger_not_ready_result THEN
   COND_REWRITE_TAC updater_ready_result THEN
   ASM_REWRITE_TAC[]);
e (UNDISCH_N_TAC 3 THEN CONV_TAC ARITH_CONV);

e (COND_REWRITE_TAC trigger_not_ready_result THEN
   COND_REWRITE_TAC updater_not_ready_result THEN
```

```
    ASM_REWRITE_TAC[]);
  val run_u_mono = save_top_thm "run_u_mono";
```

run_t_mono

Show that run_t $(n+k) \geq$ run_t (n). The proof is by induction on k, and the induction step is proved by runnable process case analysis. This result is used by updater_meets_deadline_0.

```
  g(`!^inevseq n k.
     (run_t (OFP_Reaction inevseq (n + k))) >=
      (run_t (OFP_Reaction inevseq n))`);

  e (GEN_TAC THEN GEN_TAC THEN INDUCT_TAC);

  e (ARITH_REWRITE_TAC [--`n + 0 = n`--] THEN CONV_TAC ARITH_CONV);

  e (ARITH_REWRITE_TAC [--`n + SUC k = SUC (n + k)`--]);
  e (ASM_CASES_TAC (--`trigger_ready inevseq (n + k)`--));

  e (COND_REWRITE_TAC trigger_ready_result THEN ASM_REWRITE_TAC[]);
  e (UNDISCH_N_TAC 2 THEN CONV_TAC ARITH_CONV);

  e (COND_REWRITE_TAC trigger_not_ready_result THEN ASM_REWRITE_TAC[]);
  val run_t_mono = save_top_thm "run_t_mono";
```

run_t_diff

Show that run_t $(n+k) -$ run_t $(n) \leq k$. The proof is by induction on k, and the induction step is proved by runnable process case analysis. This result is used by updater_meets_deadline_0.

```
  g(`!^inevseq n k.
     (run_t (OFP_Reaction inevseq (n + k))) -
      (run_t (OFP_Reaction inevseq n)) <=
     k`);

  e (GEN_TAC THEN GEN_TAC THEN INDUCT_TAC);

  e (ARITH_REWRITE_TAC [--`n + 0 = n`--] THEN CONV_TAC ARITH_CONV);

  e (ARITH_REWRITE_TAC [--`n + SUC k = SUC (n + k)`--] THEN
      UNDISCH_N_TAC 1);
```

```
e (ASM_CASES_TAC (--`trigger_ready inevseq (n + k)`--));

e (COND_REWRITE_TAC trigger_ready_result THEN CONV_TAC ARITH_CONV);
e (COND_REWRITE_TAC trigger_not_ready_result THEN CONV_TAC ARITH_CONV);
val run_t_diff = save_top_thm "run_t_diff";
```

updater_ready_result1

This lemma is used in two of the cases in the proof of updater_meets_deadline_0; the antecedent is the induction hypothesis in the proof of that lemma, with the additional assumption that the updater process is ready to run. With these assumptions, it is shown that the amount of time available in the interval for the updater process to run is less than Ctime (2).

There are two main cases in this proof, which follow from the definition of updater_ready: $\text{left\_u}\,(n * \text{Period}\,(2) + k) > 0$, and $\text{period\_u}\,(n * \text{Period}\,(2) + k) = 0$. In the first case, subgoaling on LT_MIN produces a new goal in which it must be shown that the updater run time in the interval is less than Ctime (2), and this follows from run_u_left_u_sum1 and run_u_mono. In the second case, period_u_0 is used to show that k must be zero, and so the goal is trivially true.

```
g(`!^inevseq n k.
  ((((run_u (OFP_Reaction inevseq ((n * (Period 2)) + k))) -
     (run_u (OFP_Reaction inevseq (n * (Period 2))))) =
    (MIN (Ctime 2)
     (k - ((run_t (OFP_Reaction inevseq ((n * (Period 2)) + k))) -
           (run_t (OFP_Reaction inevseq (n * (Period 2)))))))) /\
   (updater_ready inevseq ((n * (Period 2)) + k)) /\
   (k < (Period 2)) /\
   ((run_u (OFP_Reaction inevseq (n * (Period 2)))) = (n * (Ctime 2))))
   ==>
   ((Ctime 2) >
      (k - ((run_t (OFP_Reaction inevseq ((n * (Period 2)) + k))) -
      (run_t (OFP_Reaction inevseq (n * (Period 2)))))))`);

e (REWRITE_TAC [updater_ready]);
e (REPEAT STRIP_TAC);
e (MATCH_MP_TAC
   (SPEC (--`(run_u (OFP_Reaction inevseq (n * Period 2 + k))) -
             (run_u (OFP_Reaction inevseq (n * Period 2)))`--) LT_MIN));
e (CONJ_TAC);
```

```
e (ASM_CASES_TAC (--`k = 0`--));

e (POP_ASSUM SUBST1_TAC THEN ARITH_REWRITE_TAC [--`a + 0 = a`--]);
e (MP_TAC ok_processes_assum THEN CONV_TAC ARITH_CONV);

e (ARITH_RES_TAC (--`~(k = 0) ==> (k > 0)`--));
e (ARITH_RES_TAC (--`(k < Period 2) ==> (k <= Period 2)`--));
e (MP_TAC (SPEC_ALL run_u_left_u_sum1));
e (ASSUM_LIST (fn thl => REWRITE_TAC [el 1 thl, el 2 thl, el 4 thl])
   THEN REWRITE_TAC [definition "arithmetic" "MULT"]);
e (UNDISCH_N_TAC 4);
e (UNDISCH_N_TAC 5);
e (MP_TAC (SPECL [--`^inevseq`--,--`n * (Period 2)`--,--`k:num`--]
                 run_u_mono));
e (CONV_TAC ARITH_CONV);

e (ASM_REWRITE_TAC[]);

e (ASM_CASES_TAC (--`k = 0`--));

e (POP_ASSUM SUBST1_TAC);
e (ARITH_REWRITE_TAC [--`a + 0 = a`--] THEN
   MP_TAC ok_processes_assum THEN
   CONV_TAC ARITH_CONV);

e (ARITH_RES_TAC (--`~(k = 0) ==> (k > 0)`--));
e (IMP_RES_TAC MOD_NONZERO);
e (IMP_RES_TAC period_u_0);
e (RES_TAC);
val updater_ready_result1 = save_top_thm "updater_ready_result1";
```

updater_not_ready_result1

This lemma is used in two of the cases in the proof of updater_meets_deadline_0; the antecedent is the induction hypothesis in the proof of that lemma, with the additional assumption that the updater process is not ready to run. With these assumptions, it is shown that the amount of time available in the interval for the updater process to run is greater than or equal to Ctime (2).

The proof of this lemma begins by massaging the ~(updater_ready ...) hypothesis into a more convenient form (left_u $(n * \text{Period}\,(2) + k) = 0$ and period_u $(n * \text{Period}\,(2) + k) > 0$). Subgoaling on GE_MIN produces a new goal in which it must be shown that the updater run time in the interval is greater than or equal to Ctime (2), and, after showing that k must be greater than 0 using period_u_0, run_u_left_u_sum1 and the assumption that left_u $(n * \text{Period}\,(2) + k) = 0$ is used to show that the new goal is true.

```
g(`!^inevseq n k.
   ((((run_u (OFP_Reaction inevseq ((n * (Period 2)) + k))) -
      (run_u (OFP_Reaction inevseq (n * (Period 2))))) =
     (MIN (Ctime 2)
      (k - ((run_t (OFP_Reaction inevseq ((n * (Period 2)) + k))) -
            (run_t (OFP_Reaction inevseq (n * (Period 2)))))))) /\
    ~(updater_ready inevseq ((n * (Period 2)) + k)) /\
    (k < (Period 2)) /\
    ((run_u (OFP_Reaction inevseq (n * (Period 2)))) = (n * (Ctime 2))))
    ==> ((Ctime 2) <=
    (k - ((run_t (OFP_Reaction inevseq ((n * (Period 2)) + k))) -
          (run_t (OFP_Reaction inevseq (n * (Period 2)))))))`);

e (REWRITE_TAC [updater_ready]);
e (PTAUT_REWRITE_TAC [--`~(a \/ b) = (~a /\ ~b)`--]);
e (ARITH_REWRITE_TAC [--`~(a > 0) = (a = 0)`--]);
e (REPEAT STRIP_TAC);
e (MATCH_MP_TAC
   (SPEC (--`(run_u (OFP_Reaction inevseq (n * Period 2 + k))) -
             (run_u (OFP_Reaction inevseq (n * Period 2)))`--) GE_MIN));
e (CONJ_TAC);

e (ASM_CASES_TAC (--`k = 0`--));

e (UNDISCH_N_TAC 4);
e (REWRITE_TAC
   [(SYM (SPECL [--`^inevseq`--,--`(n * Period 2 + k)`--] period_u_0))]);
e (POP_ASSUM SUBST1_TAC);
e (REWRITE_TAC
    [(MP (SPECL [--`Period 2`--,--`0`--]
           (theorem "arithmetic" "MOD_MULT"))
         Period_2_gt_0)]);

e (ARITH_RES_TAC (--`~(k = 0) ==> (k > 0)`--));
e (ARITH_RES_TAC (--`(k < Period 2) ==> (k <= Period 2)`--));
e (IMP_RES_TAC run_u_left_u_sum1);
e (POP_ASSUM (MP_TAC o
              REWRITE_RULE [definition "arithmetic" "MULT"] o
              SPEC_ALL));
e (UNDISCH_N_TAC 6);
```

```
e (UNDISCH_N_TAC 8);
e (CONV_TAC ARITH_CONV);

e (ASM_REWRITE_TAC[]);
val updater_not_ready_result1 =
   save_top_thm "updater_not_ready_result1";
```

n_ge_run_t

Show that $n \geq$ run_t(n). The proof is by induction on n and runnable process case analysis. This result is used by `updater_meets_deadline_0`.

```
g(`!^inevseq n. n >= run_t (OFP_Reaction inevseq n)`);
e (GEN_TAC);
e (INDUCT_TAC);
e (REWRITE_TAC ([OFP_Reaction, OFP_InitParam] @ state_field_defs));
e (CONV_TAC ARITH_CONV);

e (UNDISCH_N_TAC 1 THEN ASM_CASES_TAC (--`trigger_ready inevseq n`--));
e (COND_REWRITE_TAC trigger_ready_result THEN CONV_TAC ARITH_CONV);
e (COND_REWRITE_TAC trigger_not_ready_result THEN CONV_TAC ARITH_CONV);
val n_ge_run_t = save_top_thm "n_ge_run_t";
```

suc_period_u_ctime_u

This lemma is used in one of the cases in the proof of `updater_meets_deadline_0`; the antecedent is the induction hypothesis in the proof of that lemma, where k is instantiated with Deadline(2). With these assumptions, it is shown that the updater run time at the end of period $n+1$ is $(n+1)\cdot$Ctime(2).

```
g(`((((run_u (OFP_Reaction inevseq (n * Period 2 + Deadline 2))) -
      (run_u (OFP_Reaction inevseq (n * Period 2)))) =
     (MIN (Ctime 2)
      (Deadline 2 -
       ((run_t (OFP_Reaction inevseq (n * Period 2 + Deadline 2))) -
        (run_t (OFP_Reaction inevseq (n * Period 2))))))) /\
    ((run_u (OFP_Reaction inevseq (n * Period 2))) = n * Ctime 2)) ==>
   (run_u (OFP_Reaction inevseq (SUC n * Period 2)) = SUC n * Ctime 2)`);

e (REWRITE_TAC [min_deadline, definition "arithmetic" "MULT"]);

e (REPEAT STRIP_TAC);
e (ASSUME_TAC
    (SPECL [--`^inevseq`--,--`n * Period 2`--,--`Deadline 2`--]
       run_u_mono));
e (ARITH_RES_TAC (--`((a >= b) /\ (a - b = c)) ==> (a = b + c)`--));
e (ASSUM_LIST (fn thl => (ASSUME_TAC (REWRITE_RULE [el 3 thl]
      (el 1 thl)))));
e (STRIP_ASSUME_TAC ok_processes_assum);
e (ARITH_RES_TAC (--`((Deadline 2 >= Ctime 2) /\ (Ctime 2 > 0)) ==>
                     (Deadline 2 > 0)`--));
e (ARITH_RES_TAC (--`(Period 2 >= Deadline 2) ==>
                     (Deadline 2 <= Period 2)`--));
e (IMP_RES_TAC
    (SPECL [--`^inevseq`--,--`n:num`--,--`Deadline 2`--]
        run_u_left_u_sum1));
e (POP_ASSUM (ASSUME_TAC o
              REWRITE_RULE [definition "arithmetic" "MULT"] o
              SPEC_ALL));
e (ARITH_RES_TAC (--`((a = b) /\ (a + c = b)) ==> (c = 0)`--));
e (MP_TAC
    (SPECL [--`^inevseq`--,--`n:num`--,--`Deadline 2`--,
            --`Period 2 - Deadline 2`--] run_u_finished));
e (ARITH_RES_TAC (--`(Period 2 >= Deadline 2) ==>
                     (Deadline 2 + (Period 2 - Deadline 2) =
                      Period 2)`--));
e (ASSUM_LIST (fn thl => REWRITE_TAC [el 1 thl, el 5 thl, el 9 thl]));
e (ARITH_REWRITE_TAC [--`a <= a`--]);
e (ASM_REWRITE_TAC[]);
val suc_period_u_ctime_u = save_top_thm "suc_period_u_ctime_u";
```

updater_meets_deadline_0

Show that equation 8.4 is true for all n and k. Instead of also showing equation 8.3, we show that

$$\text{run\_u}\,(n \cdot \text{Period}\,(2)) = n \cdot \text{Ctime}\,(2) \tag{8.12}$$

It turns out that equation 8.12 is more useful to have in the induction hypothesis, and, since lemma `run_u_left_u_sum1` shows that the equations are equivalent, this one is used.

The proof is by induction on n and k. In the base case for n, when $n = 0$, we must show that for all $k \leq \text{Period}\,(2)$:

$$\text{run\_u}\,(k) = \text{MIN}\,(\text{Ctime}\,(2), k - \text{run\_t}\,(k))$$

(Equation 8.12 follows from the definition of OFP_InitParam.) Inducting on k, the base case for k follows from the definition of OFP_InitParam. In the induction step, the induction hypothesis is:

$$\text{run\_u}\,(k) = \text{MIN}\,(\text{Ctime}\,(2), k - \text{run\_t}\,(k))$$

and the goal is:

$$\text{run\_u}\,(k+1) = \text{MIN}\,(\text{Ctime}\,(2), k + 1 - \text{run\_t}\,(k+1))$$

The induction step is proved by using runnable process case analysis. If the trigger process is ready to run, then the goal reduces to:

$$\text{run\_u}\,(k) = \text{MIN}\,(\text{Ctime}\,(2), k + 1 - (\text{run\_t}\,(k) + 1))$$

which is the same as the induction hypothesis. If the trigger process is not ready but the updater process is, then the goal reduces to:

$$\text{run\_u}\,(k) + 1 = \text{MIN}\,(\text{Ctime}\,(2), k + 1 - \text{run\_t}\,(k))$$

But, by lemma updater_ready_result1, $\text{Ctime}\,(2) > k - \text{run\_t}\,(k)$, so $\text{Ctime}\,(2) \geq k + 1 - \text{run\_t}\,(k)$. Thus, the goal can be further reduced to:

$$\text{run\_u}\,(k) + 1 = k + 1 - \text{run\_t}\,(k)$$

Also, the hypothesis can be reduced to:

$$\text{run\_u}\,(k) = k - \text{run\_t}\,(k)$$

The proof of the third case, where neither process is ready, is similar. The goal reduces to:

$$\text{run\_u}\,(k) = \text{MIN}\,(\text{Ctime}\,(2), k + 1 - \text{run\_t}\,(k))$$

and by lemma updater_not_ready_result1, $\text{Ctime}\,(2) \leq k - \text{run\_t}\,(k)$, so $\text{Ctime}\,(2) \leq k + 1 - \text{run\_t}\,(k)$. Thus, the goal can be further reduced to:

$$\text{run\_u}\,(k) = \text{Ctime}\,(2)$$

and so can the hypothesis.

The induction step for n is similar, except that equation 8.12 in the goal is no longer trivial and is shown to be true using lemma suc_period_u_ctime_u. Also, there is much more algebraic manipulation required. This result is used in the proof of the updater process requirement, updater_meets_deadline.

```
g(`!^inevseq n k.
   ((k <= (Period 2)) ==>
   (((run_u (OFP_Reaction inevseq ((n * (Period 2)) + k))) -
     (run_u (OFP_Reaction inevseq (n * (Period 2))))) =
    (MIN (Ctime 2)
     (k - ((run_t (OFP_Reaction inevseq ((n * (Period 2)) + k))) -
           (run_t (OFP_Reaction inevseq (n * (Period 2))))))))) /\
   ((run_u (OFP_Reaction inevseq (n * (Period 2)))) = (n * (Ctime 2)))`);

e (GEN_TAC);
e (INDUCT_TAC);

e (REWRITE_TAC [definition "arithmetic" "MULT"]);
e (ARITH_REWRITE_TAC [--`0 + a = a`--]);
e (REWRITE_TAC ([OFP_Reaction, OFP_InitParam] @ state_field_defs));
e (ARITH_REWRITE_TAC [--`a - 0 = a`--]);

e (INDUCT_TAC);
e (REWRITE_TAC ([OFP_Reaction, OFP_InitParam, MIN] @ state_field_defs));
e (MP_TAC ok_processes_assum THEN CONV_TAC ARITH_CONV);

e (DISCH_TAC);
e (ARITH_RES_TAC (--`(SUC a <= b) ==> (a <= b)`--));
e (ARITH_RES_TAC (--`(SUC a <= b) ==> (a < b)`--));
e (RES_TAC);
e (ASM_CASES_TAC (--`trigger_ready inevseq k`--) THENL
   [ALL_TAC,
    ASM_CASES_TAC (--`updater_ready inevseq k`--)]);

e (COND_REWRITE_TAC trigger_ready_result);
e (ASM_REWRITE_TAC[MIN]);
e (CONV_TAC ARITH_CONV);

e (COND_REWRITE_TAC trigger_not_ready_result THEN
   COND_REWRITE_TAC updater_ready_result);
e (MP_TAC
    (REWRITE_RULE [definition "arithmetic" "MULT",
                   definition "arithmetic" "ADD"]
         (SPECL [--`^inevseq`--,--`0`--,--`k:num`--]
            updater_ready_result1)));
e (REWRITE_TAC ([OFP_Reaction, OFP_InitParam] @ state_field_defs));
e (ARITH_REWRITE_TAC [--`a - 0 = a`--]);
e (DISCH_TAC);
e (RES_TAC);
```

```
e (ARITH_RES_TAC (--`(a > b - c) ==> (a >= SUC b - c)`--));
e (UNDISCH_N_TAC 6);
e (COND_REWRITE_TAC MIN_GT);
e (COND_REWRITE_TAC MIN_GE);
e (MP_TAC (SPECL [--`^inevseq`--,--`k:num`--] n_ge_run_t));
e (CONV_TAC ARITH_CONV);

e (COND_REWRITE_TAC trigger_not_ready_result THEN
   COND_REWRITE_TAC updater_not_ready_result);
e (MP_TAC
    (REWRITE_RULE [definition "arithmetic" "MULT",
                   definition "arithmetic" "ADD"]
     (SPECL [--`^inevseq`--,--`0`--,--`k:num`--]
         updater_not_ready_result1)));
e (REWRITE_TAC ([OFP_Reaction, OFP_InitParam] @ state_field_defs));
e (ARITH_REWRITE_TAC [--`a - 0 = a`--]);
e (DISCH_TAC);
e (RES_TAC);
e (ARITH_RES_TAC (--`(a <= b - c) ==> (a <= SUC b - c)`--));
e (UNDISCH_N_TAC 6);
e (COND_REWRITE_TAC MIN_LE);

e (INDUCT_TAC);

e (CONJ_TAC);
e (ARITH_REWRITE_TAC [--`a + 0 = a`--, --`a - a = 0`--]);
e (REWRITE_TAC [MIN]);
e (CONV_TAC ARITH_CONV);

e (POP_ASSUM (MP_TAC o (SPEC (--`Deadline 2`--))) THEN STRIP_TAC);
e (STRIP_ASSUME_TAC ok_processes_assum);
e (ARITH_RES_TAC (--`(Period 2 >= Deadline 2) ==>
                     (Deadline 2 <= Period 2)`--));
e (RES_TAC);
e (IMP_RES_TAC suc_period_u_ctime_u);

e (REPEAT STRIP_TAC);
e (ARITH_RES_TAC (--`(SUC a <= b) ==> (a <= b)`--));
e (ARITH_RES_TAC (--`(SUC a <= b) ==> (a < b)`--));
e (UNDISCH_N_TAC 4 THEN STRIP_TAC);
e (RES_TAC);
e (ARITH_REWRITE_TAC [--`a + SUC b = SUC (a + b)`--]);

e (ASM_CASES_TAC (--`trigger_ready inevseq (SUC n * Period 2 + k)`--)
   THENL [ALL_TAC,
    ASM_CASES_TAC (--`updater_ready inevseq (SUC n * Period 2 + k)`--)]);

e (COND_REWRITE_TAC trigger_ready_result);
e (ASSUME_TAC (SPECL [--`^inevseq`--,--`SUC n * Period 2`--,--`k:num`--]
                     run_t_mono));
e (ARITH_RES_TAC (--`(b >= c) ==> ((SUC a - (SUC b - c)) =
      (a - (b - c)))`--));
e (POP_ASSUM (SUBST1_TAC o (SPEC (--`k:num`--))));
e (ASM_REWRITE_TAC[]);

e (COND_REWRITE_TAC trigger_not_ready_result THEN
   COND_REWRITE_TAC updater_ready_result);
e (IMP_RES_TAC updater_ready_result1);
e (ARITH_RES_TAC (--`(a > b - (c - d)) ==> (a >= SUC b - (c - d))`--));
e (UNDISCH_N_TAC 15);
e (COND_REWRITE_TAC MIN_GT THEN COND_REWRITE_TAC MIN_GE);
e (REWRITE_TAC [
   (MATCH_MP
    (EQT_ELIM (ARITH_CONV (--`(k <= c) ==> ((SUC c) - k =
       SUC (c - k))`--)))
    (SPECL [--`^inevseq`--,--`SUC n * Period 2`--,--`k:num`--]
       run_t_diff))]);
e (REWRITE_TAC [
   (MATCH_MP
    (EQT_ELIM (ARITH_CONV (--`(a >= b) ==> (SUC a - b = SUC (a - b))`--)))
    (SPECL [--`^inevseq`--, --`SUC n * Period 2`--, --`k:num`--]
           run_u_mono))]);
e (ARITH_REWRITE_TAC [--`(a = b) ==> (SUC a = SUC b)`--]);

e (COND_REWRITE_TAC trigger_not_ready_result THEN
   COND_REWRITE_TAC updater_not_ready_result);
e (IMP_RES_TAC
   (SPECL [--`^inevseq`--,--`SUC n`--,--`k:num`--]
      updater_not_ready_result1));
e (ARITH_RES_TAC (--`(a <= (b - (c - d))) ==> (a <= SUC b - (c - d))`--));
e (UNDISCH_N_TAC 7);
e (COND_REWRITE_TAC MIN_LE);

e (ASM_REWRITE_TAC[]);
val updater_meets_deadline_0 = save_top_thm "updater_meets_deadline_0";
```

updater_meets_deadline

The proof of the updater process requirement follows trivially from lemmas min_deadline and updater_meets_deadline_0, when the latter is instantiated with Deadline (2).

```
g(`!^inevseq n.
   (run_u (OFP_Reaction inevseq (n * Period 2 + Deadline 2))) -
    (run_u (OFP_Reaction inevseq (n * Period 2))) =
   Ctime 2`);

e (REPEAT STRIP_TAC);
e (STRIP_ASSUME_TAC ok_processes_assum);
e (ARITH_RES_TAC (--`Period 2 >= Deadline 2 ==>
      Deadline 2 <= Period 2`--));
e (ASSUME_TAC (SPECL [--`^inevseq`--,--`n:num`--,--`Deadline 2`--]
                updater_meets_deadline_0));
e (RES_TAC);
e (UNDISCH_N_TAC 1 THEN REWRITE_TAC [min_deadline]);
val updater_meets_deadline = save_top_thm "updater_meets_deadline";
```

8.5 Remaining and Future Work

Unfortunately, there was not enough time to work out the proof of lemma run_t_le_CPU. It is not difficult to see that it is true, but difficult to state and prove formally. With the proof of this lemma, the proof that the updater process always meets its deadline will be complete.

8.5.1 Arbitrary Number of Processes

The proof presented in this paper is for a particular scheduler with one periodic and one sporadic process. It would be desirable to generalize the proof to an arbitrary number of periodic and sporadic processes, as described in [23]. This, however, would add more detail and complexity to an already detailed and complex proof. For example, the deadline for process i must allow for the maximum CPU use of all higher-priority processes, and the latency restriction becomes:

$$\text{Deadline}\,(i) \geq \text{Ctime}\,(i) + \sum_{j=1}^{i-1} \text{CPU}\,(\text{Deadline}\,(i), j)$$

The scheduler state would have to be a collection of functions which map process numbers to time left for processes to run, total run times, etc., many simple terms in the presented proof would become summations, and many properties about summation would have to be stated and proved.

8.5.2 More Abstract Schedulers

The proof presented in this paper deals with a specific instance of a static priority interrupt scheduler. It would be desirable to prove the process requirements for a more abstract scheduler that is described in terms of properties expresses as HOL predicates rather than an actual PSL specification. (The lemmas `updater_ready_result` and `trigger_ready_result`, among others, are a step in this direction.) Then, to prove a particular priority-based scheduler meets its requirements, it would only be necessary to prove that the scheduler has the properties assumed in the proof. However, from some very preliminary work, it is not clear that proving that a scheduler has these properties is much easier than proving that the scheduler meets the requirements. More work is needed to decide the question.

8.5.3 More Complex Models

Teixeira [23] describes a number of ways in which his model can be generalized.

Accounting for Scheduling Overhead

In a more refined model of the scheduler, the actions of the scheduler itself (deciding which task to run, switching between tasks) also require time. This overhead could be allocated to the computation time of the tasks in various ways.

Non-preemptive Scheduling

It is assumed that a process can always be interrupted so that a higher-priority process may be run. If the currently running process must finish before a new one can be started, the latency requirement for process i becomes:

$$\text{Deadline}\,(i) \geq \text{Ctime}\,(i) + \sum_{j=1}^{i-1} \text{CPU}\,(\text{Deadline}\,(i), j) + \max_{j=i+1}^{n} \text{Ctime}\,(j)$$

to allow for the possibility that a process must wait for a lower-priority process to complete.

Non-distinct Priorities

If more than one process may have the same priority, the latency restriction becomes:

$$\text{Deadline}\,(i) \geq \text{Ctime}\,(i) + \sum_{P_j \geq P_i} \text{CPU}\,(\text{Deadline}\,(i), j)$$

where P_i is the priority of process i. The net result is that processes in this model must have greater latencies than processes in a model with distinct priorities, to allow for other processes with the same or greater priority.

Bibliography

[1] Air Force Studies Board. Multilevel data management security. Technical report, National Research Council, Washington, DC, 1983.

[2] R.K. Bauer, T.A. Berson, and R.J. Feiertag. A key distribution protocol using event markers. *ACM Transactions on Computer Systems*, 1(3):249–255, August 1983.

[3] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report MTR-2997, Revision 2, MITRE Corp., Bedford MA, March 1976.

[4] D. Elliott Bell and Leonard J. La Padula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The Mitre Corporation, May 1973.

[5] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[6] W.E. Boebert. Constructing an infosec system using LOCK technology. In *Proceedings of the 11th National Computer Security Conference*, pages 89–95. National Nureau of Standards, October 1988.

[7] D.E.Denning and G.M. Sacco. Timestamps in key distribution protocols. *CACM*, 24(8):533–536, August 1981.

[8] Li Gong. Handling infeasible specifications of cryptographic protocols. In *Proceedings of The Computer Security Foundations Workshop IV*, pages 99–102, Franconia, NH, June 1991.

[9] Geoffrey R. Hird, Douglas Hoover, and Mark Howard. Basic technology for SDI computer security. Extensions to Romulus to include temporal constructs. Technical Report RADC-TR-90-435, Vol IV, Rome Air Development Center, Grifiss Air Force Base, New York, December 1990. Distribution Restricted. *

[10] S. Jajodia and R. Sandhu. Polyinstantiation integrity in multilevel relations revisited. In *Proceedings of the 4th IFIP WG11.3 Workshop on Database Security*, September 1990.

[11] Boris Kogan and Sushil Jajodia. Data replication and multilevel-secure transaction processing, June 1993. Manuscript.

[12] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[13] T.F. Lunt. The true meaning of polyinstantiation: Proposal for an operational semantics for a multi-level relational database system. In *Proceedings of the 3rd RADC Database Security Workshop*, June 1990.

[14] Daryl McCullough. Covert channels and degrees of insecurity. In *Proceedings of Computer Security Foundations Workshop*, pages 1–33, Franconia, NH, June 1988. The MITRE Corporation, M88-37.

[15] Daryl McCullough. Security analysis of a token ring using Ulysses. In *Proceedings of the Fourth Annual Conference on Computer Assurance (COMPASS '89)*, pages 113–118, Gaithersburg, MD, June 1989.

[16] Abha Moitra and Edward A. Schneider. Basic technology for SDI computer security. SDI real time trusted computer based requirements. Technical Report RADC-TR-90-435, Vol III, Rome Air Development Center, Grifiss Air Force Base, New York, December 1990. Distribution Restricted.

[17] Aloysius K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real Time Environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

*Although this report references the limited document noted above, no limited information has been extracted. USGO Agencies and their contractors; critical technology; Dec 90.

[18] ORA. Romulus: A computer security properties modeling environment, final report - volume 3: Specification languages. Technical report, ORA, June 1990.

[19] ORA. A secure network device driver, final report. Technical report, ORA, November 1990.

[20] Ian Sutherland. Fault-tolerant reference monitor. Technical report, ORA, November 1991. Final Report, U.S Army Strategic Defense SBIR Contract DASG60-91-C-0079.

[21] Ian Sutherland. Shared-state restrictiveness. ORA Internal Report, July 1992.

[22] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[23] Thomas J. Teixeira. Static priority interrupt scheduling. In *Proceedings of the 7th Texas Conference on Computing Systems.* University of Houston, 1978.

☆U.S. GOVERNMENT PRINTING OFFICE: 1996-710-126-20218