Noninterference, Transitivity, and Channel-Control Security Policies[1]

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

`Rushby@csl.sri.com`
Phone: +1 (415) 859-5456 Fax: +1 (415) 859-2844

Technical Report CSL-92-02
December 1992

---

[1] This research was supported in part by the Office of Cryptological Research (OCREAE) under contract MDA904–85–K–0001, by the Naval Research Laboratory and NASA under contract NAS1 18969 (Task 4), and by the US Government under contract MDA904-90-C-7016.

Abstract

We consider noninterference formulations of security policies [10] in which the "interferes" relation is intransitive. Such policies provide a formal basis for several real security concerns, such as channel control [22, 23], and assured pipelines [4]. We show that the appropriate formulation of noninterference for the intransitive case is that developed by Haigh and Young for "multidomain security" (MDS) [12, 13]. We construct an "unwinding theorem" [11] for intransitive polices and show that it differs significantly from that of Haigh and Young. We argue that their theorem is incorrect. An appendix presents a mechanically-checked formal specification and verification of our unwinding theorem.

We also consider the relationship between transitive and intransitive formulations of security. We show that the standard formulations of noninterference and unwinding [10, 11] correspond exactly to our intransitive formulations, specialized to the transitive case. We show that transitive polices are precisely the "multilevel security" (MLS) polices, and that any MLS secure system satisfies the conditions of the unwinding theorem.

In addition, we consider the relationship between noninterference formulations of security and access control formulations, and we identify the "reference monitor assumptions" that play a crucial role in establishing the soundness of access control implementations.

Contents

Chapter 1

Introduction

The concept of noninterference was introduced by Goguen and Meseguer [10] in order to provide a formal foundation for the specification and analysis of security policies and the mechanisms that enforce them. Apart from the work of Feiertag, Levitt, and Robinson [9]—which can be seen as a precursor to that of Goguen and Meseguer—previous efforts, among which those of Bell and La Padula [3] were the most influential, formulated security in terms of access control. Access control formulations suffer from a number of difficulties. First, because they are described in terms of a mechanism for enforcing security, they provide no guidance in circumstances where those mechanisms prove inadequate. Second, it is easy to construct perverse interpretations of access control policies that satisfy the letter, but not the intent of the policy, to the point of being obviously unsecure [17,18]. The proponents of access control formulations counter that interpretations or implementations must be "faithful representations" of the model, but they provide no formal definition of that term.

In contrast, noninterference formulations are pure statements of policy, with no commitment to a specific mechanism for enforcing them—although techniques have been developed for demonstrating that specific mechanisms enforce given noninterference policies. Secondly, noninterference policies have the form of a logical theory; *any* implementation that is a model for the theory (i.e., validates its axioms) will be secure.

The idea of noninterference is really rather simple: a security domain u is noninterfering with domain v if no action performed by u can influence subsequent outputs seen by v. Noninterference has been quite successful in providing formal underpinnings for military multilevel security policies and for the methods of verifying their implementations [11,25].

There are, however, a number of practical security problems that seem beyond the scope of noninterference formulations. One of these is "channel-control," first formulated by Rushby [22, 23]. Channel control security policies can be represented by directed graphs, where nodes represent security domains and edges indicate the direct information flows that are allowed. The paradigmatic example of a channel-control problem is a controller for end-to-end encryption, as portrayed in Figure 1.1 [1, 22].

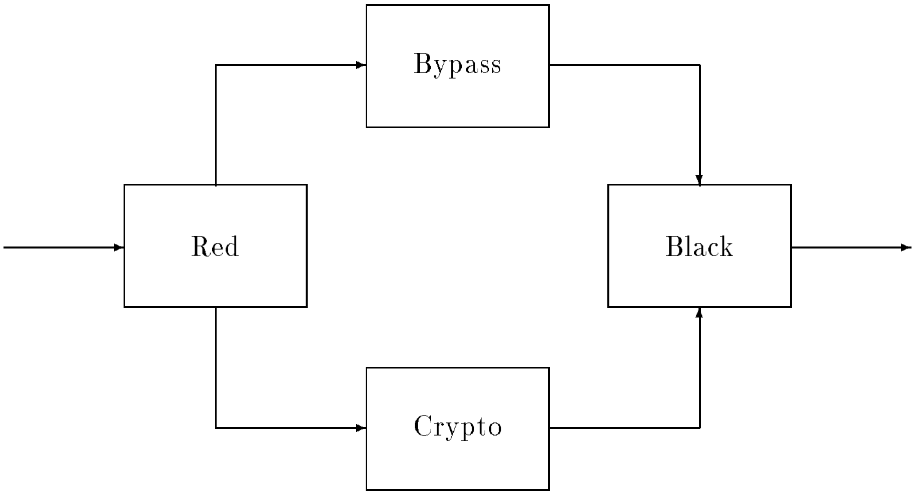

Figure 1.1: End-to-end encryption controller

Plaintext messages arrive at the Red side of the controller; their bodies are sent through the encryption device (Crypto); their headers, which must remain in plaintext so that network switches can interpret them, are sent through the Bypass. Headers and encrypted bodies are reassembled in the Black side and sent out onto the network. The security policy we would like to specify here is the requirement that the *only* channels for information flow from Red to Black must be those through the Crypto and the Bypass.[1] Thus, an important characteristic of many channel control policies is that the edges indicating allowed information flows are not transitive: information is allowed to flow from Red to Black via the Crypto and Bypass, but cannot do so directly.

Another example is shown in Figure 1.2, where transitive and intransitive elements are combined. The edges to the left represent the conventional transitive flow relations between the classification levels used in the USA. On the right are edges to and from a special Downgrader domain that are intransitive. The flows represented by these edges are intransitive because, although information can flow, for example, from the Top Secret to the Confidential domain via the Downgrader, it cannot flow directly from Top Secret to Confidential. Thus, information can flow "upward" in

---

[1] It is a separate problem to specify what those components must do.

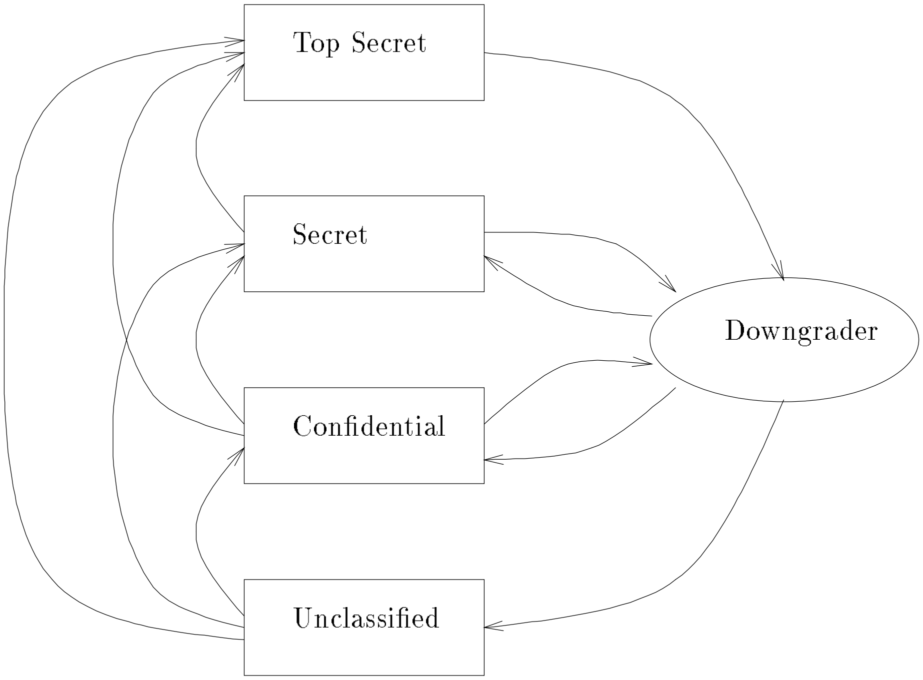

Figure 1.2: Controlled downgrading

security level without restriction, but only flow “downward” through the mediation of the presumably trusted Downgrader domain.

Channel control policies such as those just described seem able to specify a number of security concerns that are beyond the reach of standard security modeling techniques. Boebert and Kain have argued persuasively [4] that a variation on channel-control called “type enforcement” can be used to solve many vexing security problems. A worthwhile challenge, then, is to find an adequate formal foundation for channel-control policies and their ilk.

An early attempt to provide a formal method for verifying, though not specifying, channel-control policies was based on a technique for verifying complete separation [22, 24]. The idea was to remove the mechanisms that provided the intended channels, and then prove that the components of the resulting system were isolated. This approach has recently been shown to be subtly flawed [14], although the method for establishing complete separation has survived fairly intensive scrutiny [15, 28] with only minor emendations.

The success of noninterference formulations in explicating multilevel security policies naturally invites consideration of a noninterference foundation for channel-control. This presents quite a challenge, however. For example, it is clear the Red side of the encryption controller of Figure 1.1 necessarily interferes with the Black; we need to find a way of saying that this interference must only occur through the mediation of the Crypto or the Bypass. Goguen and Meseguer proposed a way of doing this in their original paper on noninterference [10], but the method was incorrect. Goguen and Meseguer recognized this in their second paper on the subject [11] and they introduced several extensions to the basic formulation of noninterference. However, the first really satisfactory treatment of intransitive noninterference policies was given by Haigh and Young [12], with a more polished version the following year [13]. They showed that it was necessary to consider the complete sequence of actions performed subsequent to a given action in order to determine whether that action is allowed to interfere with another domain. For example, an action by the Red domain is allowed to interfere with the Black domain only if there is some intervening action from either the Crypto or the Bypass.

The main purpose of this report is to show that channel-control security policies can be modeled by noninterference policies in which the "interferes" relation is intransitive and in which the definition used for "interference" is that of Haigh and Young. We also show that conventional multilevel policies are a special case of channel-control policies, corresponding to those whose "interferes" relation is transitive. We show that our results collapse to the familiar ones in this special case, thereby providing some additional evidence for their veracity.

An important component of noninterference formulations of security are the "unwinding" theorems [11, 13] that establish conditions on the behavior of individual actions sufficient to ensure security of the system. These unwinding theorems provide the basis of practical methods for formally verifying that an implementation satisfies a noninterference security policy. The main result of this report is the derivation of an unwinding theorem for the channel-control case. We show that this theorem differs significantly from that of Haigh and Young and we argue that their result is incorrect.

The development of noninterference and unwinding for the channel-control case is surprisingly intricate, and in view of the previous history of failed attempts, we present our development rather formally and describe the proofs in detail. An appendix describes the formal verification of our main theorem using the EHDM formal specification and verification system [27].

This report is organized as follows. In the next chapter we present a development of the standard noninterference formulation of security, and then consider the relationship between noninterference security policies and access control policies. This development is structured to provide a model and a basis for comparison with the generalization given later. Chapter 3 examines the case of intransitive noninterference policies and argues that these have no useful interpretation within the standard formulation of noninterference. The second part of Chapter 3 examines the special properties of transitive policies and shows that they are identical to classical multilevel security. Chapter 4 presents a modified formulation of noninterference that does provide a meaningful interpretation to intransitive policies and derives an unwinding theorem for that interpretation. Chapter 5 compares the transitive and intransitive noninterference formulations, and compares our unwinding theorem with that of Haigh and Young. Chapter 6 presents our conclusions. The appendix presents a formal specification and verification of our Intransitive Unwinding Theorem that has been mechanically checked using the EHDM Verification System [27].

Chapter 2

Basic Noninterference

In this chapter we present the core of Goguen and Meseguer's formulation of security in terms of noninterference assertions [10], and the unwinding theorem [11] that underlies the associated verification techniques. Our notation differs considerably from that of Goguen and Meseguer and is more similar to that of later authors, such as Haigh and Young [13].

We model a computer system by a conventional finite-state automaton.

Definition 1 A *system* (or *machine*) M is composed of

- a set S of *states*, with an *initial state* $s_0 \in S$,
- a set A of *actions*, and
- a set O of *outputs*,

together with the functions *step* and *output*:

- $\mathit{step}: S \times A \to S$,
- $\mathit{output}: S \times A \to O$.

We generally use the letters $\ldots s, t, \ldots$ to denote states, letters $a, b, \ldots$ from the front of the alphabet to denote actions, and Greek letters $\alpha, \beta, \ldots$ to denote sequences of actions.

Actions can be thought of as "inputs," or "commands," or "instructions" to be performed by the machine; $\mathit{step}(s, a)$ denotes the next state of the system when action a is applied in state s, while $\mathit{output}(s, a)$ denotes the result returned by the action.

We derive a function *run*

- $run\colon S \times A^* \to S$,

the natural extension of *step* to sequences of actions, by the equations

$$\begin{aligned} run(s, \Lambda) &= s, \text{ and} \\ run(s, a \circ \alpha) &= run(step(s, a), \alpha), \end{aligned}$$

where Λ denotes the empty sequence and $\circ$ denotes concatenation.[1]

In order to discuss security, we must assume some set of security "domains" and a policy that restricts the allowable flow of information among those domains. The agents or subjects of a particular security domain interact with the system by presenting it with actions, and observing the results obtained. Thus we assume

- a set D of security domains, and
- a function $dom\colon A \to D$ that associates a security domain with each action.

We use letters $\ldots u, v, w, \ldots$ to denote domains.

A *security policy* is specified by a reflexive relation $\rightsquigarrow$ on D. We use $\not\rightsquigarrow$ to denote the complement relation, that is

$$\not\rightsquigarrow = (D \times D) \setminus \rightsquigarrow$$

where $\setminus$ denotes set difference. We speak of $\rightsquigarrow$ and $\not\rightsquigarrow$ as the *interference* and *noninterference* relations, respectively. A policy is said to be *transitive* if its interference relation has that property. $\square$

We wish to define security in terms of information flow, so the next step is to capture the idea of the "flow of information" formally. The key observation is that information can be said to flow from a domain u to a domain v exactly when actions submitted by domain u cause the behavior of the system perceived by domain v to be different from that perceived when those actions are not present. We therefore define a function that removes, or "purges," from an action sequence all those actions submitted by domains that are required to be noninterfering with a given domain. The machine is secure if a given domain v is unable to distinguish between the state of the machine after it has processed a given action sequence, and the state after processing the same sequence purged of actions required to be noninterfering with v.

---

[1] Observe that we define *run* using right recursion: that is, we specify $run(s, a \circ \alpha) = run(step(s, a), \alpha)$, rather than the more common left recursive form $run(s, \alpha \circ a) = step(run(s, \alpha), a)$. The choice of right recursion slightly complicates the proof of the basic unwinding theorem (Theorem 1); we employ it here for consistency with the later, more complex development in which its use is essential.

Definition 2 For $v \in D$ and α an action sequence in A^*, we define $purge(\alpha, v)$ to be the subsequence of α formed by deleting all actions associated with domains u such that $u \not\rightsquigarrow v$, that is:

$$\begin{aligned} purge(\Lambda, v) &= \Lambda \\ purge(a \circ \alpha, v) &= \begin{cases} a \circ purge(\alpha, v) & \text{if } dom(a) \rightsquigarrow v \\ purge(\alpha, v) & \text{otherwise.} \end{cases} \end{aligned}$$

We identify security with the requirement that

$$output(run(s_0, \alpha), a) = output(run(s_0, purge(\alpha, dom(a))), a).$$

Because we frequently use expressions of the form $output(run(s_0, \alpha), a)$, it is convenient to first introduce the functions *do* and *test* to abbreviate these forms:

- $do\colon A^* \rightarrow S$
- $test\colon A^* \times A \rightarrow O$

where

$$\begin{aligned} do(\alpha) &= run(s_0, \alpha), \text{ and} \\ test(\alpha, a) &= output(do(\alpha), a). \end{aligned}$$

Then we say a system is *secure* for the policy $\rightsquigarrow$ if

$$test(\alpha, a) = test(purge(\alpha, dom(a)), a).^2$$

□

The intuition here is that the machine starts off in the initial state s_0 and is presented with a sequence $\alpha \in A^*$ of actions. This causes the machine to produce a series of outputs and to progress through a series of states, eventually reaching the state $do(\alpha)$. At that point the action a is performed, and the corresponding output $test(\alpha, a)$ is observed. We can think of presentation of the action a and observation of its output as an experiment performed by $dom(a)$ in order to learn something about the action sequence α. If $dom(a)$ can distinguish between the action sequences α and $purge(\alpha, dom(a))$ by such experiments, then an action by some domain $u \not\rightsquigarrow dom(a)$ has "interfered" with $dom(a)$ and the system is not secure with respect to policies that specify $u \not\rightsquigarrow dom(a)$.

There are several plausible variations on this notion of security. For example, rather than restricting $dom(a)$ to observe only the individual outputs $test(\alpha, a)$, and

---

[2] Formulas such as these are to be read as universally quantified over their free variables (here a and α).

$test(purge(\alpha, dom(a)), a)$ in its attempt to distinguish α from $purge(\alpha, dom(a))$, we could allow the whole sequence of outputs produced by actions b in α satisfying $dom(b) \leadsto dom(a)$ (i.e., the outputs of the actions in α which $dom(a)$ can legitimately observe) to be considered. It is fairly straightforward to prove that such variations are equivalent to the definition used here.

The noninterference definition of security is expressed in terms of sequences of actions and state transitions; in order to obtain straightforward techniques for verifying the security of systems, we would like to derive conditions on individual state transitions. The first step in this development is to partition the states of the system into equivalence classes that all "appear identical" to a given domain. The verification technique will then be to prove that each domain's view of the system is unaffected by the actions of domains that are required to be noninterfering with it.

Definition 3 A system M is *view-partitioned* if, for each domain $u \in D$, there is an equivalence relation $\overset{u}{\sim}$ on S. These equivalence relations are said to be *output consistent* if

$$s \overset{dom(a)}{\sim} t \supset output(s, a) = output(t, a).^3$$

□

Output consistency is required in order to ensure that two states s and t that appear identical to domain u really are indistinguishable in terms of the outputs they produce in response to actions from u.

The definition of security requires that the outputs seen by one domain are unaffected by the actions of other domains that are required to be noninterfering with the first. The next result shows that, for an output consistent system, security is achieved if "views" are similarly unaffected.

Lemma 1 *Let $\leadsto$ be a policy and M a view-partitioned, output consistent system such that,*

$$do(\alpha) \overset{u}{\sim} do(purge(\alpha, u)).$$

Then M is secure for $\leadsto$.

Proof: Setting $u = dom(a)$ in the statement of the lemma gives

$$do(\alpha) \overset{dom(a)}{\sim} do(purge(\alpha, dom(a))),$$

and output consistency then provides

$$output(do(\alpha), a) = output(do(purge(\alpha, dom(a))), a).$$

---

[3] We use $\supset$ to denote implication.

But this is simply

$$test(\alpha, a) = test(purge(\alpha, dom(a)), a),$$

which is the definition of security for $\rightsquigarrow$ given by Definition 2. □

Next, we define constraints on individual state transitions.

Definition 4 Let M be a view-partitioned system and $\rightsquigarrow$ a policy. We say that M *locally respects* $\rightsquigarrow$ if

$$dom(a) \not\rightsquigarrow u \supset s \overset{u}{\sim} step(s, a)$$

and that M is *step consistent* if

$$s \overset{u}{\sim} t \supset step(s, a) \overset{u}{\sim} step(t, a).$$

□

We now have the local conditions on individual state transitions that are sufficient to guarantee security. This result is a version of the unwinding theorem of Goguen and Meseguer [11].

Theorem 1 (Unwinding Theorem) *Let* $\rightsquigarrow$ *be a policy and* M *a view-partitioned system that is*

1. *output consistent,*

2. *step consistent, and*

3. *locally respects* $\rightsquigarrow$.

Then M *is secure for* $\rightsquigarrow$.

Proof: We use proof by induction on the length of α to establish

$$s \overset{u}{\sim} t \supset run(s, \alpha) \overset{u}{\sim} run(t, purge(\alpha, u)). \tag{2.1}$$

The basis is the case $\alpha = \Lambda$ and is elementary. For the inductive step, we assume the inductive hypothesis for α of length n and consider $a \circ \alpha$. By definition,

$$run(s, a \circ \alpha) = run(step(s, a), \alpha). \tag{2.2}$$

For $run(t, purge(a \circ \alpha, u))$, there are two cases to consider.

Case 1: $dom(a) \rightsquigarrow u$. In this case, the definition of *purge* provides

$$run(t, purge(a \circ \alpha, u)) = run(t, a \circ purge(\alpha, u)),$$

and the right hand side expands to give

$$run(t, purge(a \circ \alpha, u)) = run(step(t, a), purge(\alpha, u)). \tag{2.3}$$

Since $s \overset{u}{\sim} t$ and the system is step consistent, it follows that

$$step(s, a) \overset{u}{\sim} step(t, a)$$

and the inductive hypothesis then gives

$$run(step(s, a), \alpha) \overset{u}{\sim} run(step(t, a), purge(\alpha, u))$$

which, by virtue of (2.2) and (2.3), completes the inductive step in this case.

Case 2: $dom(a) \not\rightsquigarrow u$. In this case, the definition of *purge* provides

$$run(t, purge(a \circ \alpha, u)) = run(t, purge(\alpha, u)) \tag{2.4}$$

and the facts that $dom(a) \not\rightsquigarrow u$ and that M locally respects $\rightsquigarrow$ ensure

$$s \overset{u}{\sim} step(s, a).$$

Since $s \overset{u}{\sim} t$ and $\overset{u}{\sim}$ is an equivalence relation, the latter provides

$$step(s, a) \overset{u}{\sim} t$$

and the inductive hypothesis then gives

$$run(step(s, a), \alpha) \overset{v}{\sim} run(t, purge(\alpha, u)),$$

which, by virtue of (2.2) and (2.4), completes the inductive step.

In order to complete the proof, we take $s = t = s_0$ in 2.1 to obtain

$$do(\alpha) \overset{u}{\sim} do(purge(\alpha, u))$$

and then, since M is output consistent, invoke Lemma 1 to complete the proof. □

The unwinding theorem is important because it provides a basis for practical methods for verifying systems that enforce noninterference policies, and also serves to relate noninterference policies to access control mechanisms. We illustrate the latter point by using the unwinding theorem to establish the security of a simple access control mechanism.

2.1 Access Control Interpretations

In order to consider access control mechanisms formally, we need a more elaborate system model. First of all, we need to impose some internal structure on the system state, supposing it to be composed of individual storage locations, or "objects," each of which has a name and a value. The name of each location is fixed, but its value may change from one state to another. Access control functions determine whether a given security domain may "observe" or "alter" the values in given storage locations. We collect these ideas together and introduce convenient notation in the following definition.

Definition 5 A machine has a *structured state* if there exist

- a set N of *names*,
- a set V of *values*, and a function
- $\mathit{contents}\colon S \times N \to V$

with the interpretation that $\mathit{contents}(s, n)$ is the value of the object named n in state s. In addition, we require functions

- $\mathit{observe}\colon D \to \mathcal{P}(N)$, where $\mathcal{P}$ denotes powerset, and
- $\mathit{alter}\colon D \to \mathcal{P}(N)$

with the interpretation that $\mathit{observe}(u)$ is the set of locations whose values can be observed by domain u, while $\mathit{alter}(u)$ is the set of locations whose values can be changed by u. These functions encode the "access control matrix" that represents the access control policy of the system. An access control policy is enforced when the behavior of the system matches the intended interpretation of the *observe* and *alter* functions. This requires the following three conditions to be satisfied:

Reference Monitor Assumptions

1. First, for $u \in D$ define the relation $\overset{u}{\sim}$ on states by

$$s \overset{u}{\sim} t \text{ iff } (\forall n \in \mathit{observe}(u)\colon \mathit{contents}(s, n) = \mathit{contents}(t, n)).$$

Then, in order for the output of an action a to depend only on the values of objects to which $\mathit{dom}(a)$ has *observe* access, we require:

$$s \overset{\mathit{dom}(a)}{\sim} t \supset \mathit{output}(s, a) = \mathit{output}(t, a).$$

2. Next, when an action a transforms the system from one state to another, the new values of all changed objects must depend only on the values of objects to which $dom(a)$ has observe access. That is:

$$\begin{aligned} s \overset{dom(a)}{\sim} t \wedge (&contents(step(s,a),n) \neq contents(s,n) \\ &\vee contents(step(t,a),n) \neq contents(t,n)) \\ \supset \quad &contents(step(s,a),n) = contents(step(t,a),n). \end{aligned} \tag{2.5}$$

   This condition is rather difficult; we discuss it following the complete definition.

3. Finally, if an action a changes the value of object n, then $dom(a)$ must have *alter* access to n:

$$contents(step(s,a),n) \neq contents(s,n) \supset n \in alter(dom(a)).$$

These three conditions are called the "Reference Monitor Assumptions" since they capture the assumptions on the "reference monitor" that performs the access control function in any concrete instantiation of the theory. □

The second of the Reference Monitor Assumptions is somewhat tricky, so we will now explain it in more detail. The goal is to specify that if action a changes the value of location n, then the only information that may be used in creating the new value should be that provided in variables to which $dom(a)$ has observe access. Thus, if two states s and t have the same values in all the locations to which $dom(a)$ has observe access (i.e., if $s \overset{dom(a)}{\sim} t$), then it seems we should specify

$$contents(step(s,a),n) = contents(step(t,a),n) \tag{2.6}$$

for all locations n. The flaw in this specification is that if $dom(a)$ does *not* have observe access to n, then $s \overset{dom(a)}{\sim} t$ does not prevent $contents(s,n) \neq contents(t,n)$. If a does not change the value of location n we will then legitimately have

$$contents(step(s,a),n) \neq contents(step(t,a),n).$$

The repair to the definition is to require (2.6) to hold only if a does change the value of location n. This is accomplished in (2.5), the second of the Reference Monitor Assumptions specified in Definition 5 above.

This problem of specifying what it means for an operation to "reference" a location has been studied before; Popek and Farber [21], for example, construct the dual notion "*NoRef*" as follows. First, for $n \in N$, define the equivalence relation $\overset{n}{\cong}$ by

$$s \overset{n}{\cong} t \overset{\text{def}}{=} (\forall m \in N : contents(s,m) = contents(t,m) \vee m = n).$$

That is, $s \overset{n}{\cong} t$ if the values of all locations, except possibly that of n, are the same in both of states s and t. Then the predicate $\mathit{NoRef}(a, n)$, which is to be true when action a does not reference location n, is defined by

$$\mathit{NoRef}(a, n) \overset{\text{def}}{=} s \overset{n}{\cong} t \supset \mathit{step}(s, a) \overset{n}{\cong} \mathit{step}(t, a).$$

The motivation for this definition is the idea that if a does not reference the value of location n, then changing the value of that location should have no effect on the values assigned to other locations by action a. It is easy to prove that our notion of reference, as embodied in (2.5), implies the notion embodied in Popek and Farber's definition. The converse is not true. This is due to a weakness in Popek and Farber's definition which they discuss in [21, page 742 (footnote 5)]; they suggest a stronger definition whose motivation is identical to that given in our discussion of the formulation of (2.5). Unfortunately, the formal statement of Popek and Farber's stronger definition contains serious typographical errors and it is impossible to tell what was intended. Nonetheless, we consider the relationship between the description of their definition and ours to be sufficiently close that they provide additional confidence in the correctness of our formulation of the second Reference Monitor Assumption.

Given these definitions, we can now state a theorem that relates noninterference to access control mechanisms.

Theorem 2 *Let M be a system with structured state that satisfies the Reference Monitor Assumptions and the following two conditions.*

1. $u \rightsquigarrow v \supset \mathit{observe}(u) \subseteq \mathit{observe}(v)$, *and*
2. $n \in \mathit{alter}(u) \wedge n \in \mathit{observe}(v) \supset u \rightsquigarrow v$.

Then M is secure for $\rightsquigarrow$.

Proof: We show that the conditions of the theorem satisfy those of the unwinding theorem. We identify the view-partitioning relations $\overset{u}{\sim}$ of the Unwinding Theorem with the corresponding relations defined in the statement of the Reference Monitor Assumptions. Output consistency is then satisfied immediately by the first of the Reference Monitor Assumptions.

To establish step consistency, we must prove

$$s \overset{u}{\sim} t \supset \mathit{step}(s, a) \overset{u}{\sim} \mathit{step}(t, a).$$

This can be rewritten as

$$s \overset{u}{\sim} t \supset \mathit{contents}(\mathit{step}(s, a), n) = \mathit{contents}(\mathit{step}(t, a), n)$$

where $n \in \mathit{observe}(u)$. There are three cases to consider

Case 1: $contents(step(s,a),n) \neq contents(s,n)$**.** The third of the Reference Monitor Assumptions gives $n \in alter(dom(a))$; since $n \in observe(u)$, the second of the conditions in the statement of the theorem then gives $dom(a) \rightsquigarrow u$. The first of the conditions in the statement of the theorem then gives

$$observe(dom(a)) \subseteq observe(u),$$

and $s \overset{u}{\sim} t$ then implies $s \overset{dom(a)}{\sim} t$. The second of the Reference Monitor Assumptions then provides the conclusion we require.

Case 2: $contents(step(t,a),n) \neq contents(t,n)$**.** This case is symmetrical with the first.

Case 3: $contents(step(t,a),n) = contents(t,n) \wedge contents(step(t,a),n) = contents(t,n)$**.** Since $s \overset{u}{\sim} t$ and $n \in observe(u)$, we have $contents(s,n) = contents(t,n)$ and the conclusion follows immediately.

It remains to show that the construction locally respects $\rightsquigarrow$. That is, we need to show

$$dom(a) \not\rightsquigarrow u \supset s \overset{u}{\sim} step(s,a).$$

Taking the contrapositive and expanding the definition of $\overset{u}{\sim}$, this becomes

$$(\exists n \in observe(u)\colon contents(s,n) \neq contents(step(s,a),n)) \supset dom(a) \rightsquigarrow u.$$

Now if $contents(s,n) \neq contents(step(s,a),n)$, the third condition of the Reference Monitor Assumptions gives $n \in alter(dom(a))$. Hence, we have

$$n \in alter(dom(a)) \wedge n \in observe(u)$$

and so the second condition to the theorem requires $dom(a) \rightsquigarrow u$ and the proof is complete. □

In the following chapter, we will show that transitive noninterference policies satisfy the conditions of Theorem 2 and thereby relate noninterference to the familiar Bell and La Padula [3] formulation of security.

Chapter 3

Noninterference and Transitivity

The only restriction we placed on the relation $\rightsquigarrow$ defining a security policy was that it should be reflexive. However, we will show that, within the formulation presented so far, only relations that are also transitive have a useful interpretation.

In their original paper on the subject, Goguen and Meseguer [10] suggested that intransitive policies could be used to specify channel control policies. For example, the policy of the encryption controller shown in Figure 1.1 could be specified by the four assertions

$$\begin{array}{rcl} \text{Red} & \rightsquigarrow & \text{Bypass} \\ \text{Red} & \rightsquigarrow & \text{Crypto} \\ \text{Bypass} & \rightsquigarrow & \text{Black} \\ \text{Crypto} & \rightsquigarrow & \text{Black} \end{array}$$

with the understanding that all other combinations, except the reflexive ones, should be noninterfering. In particular, Red $\not\rightsquigarrow$ Black, even though Red $\rightsquigarrow$ Bypass and Bypass $\rightsquigarrow$ Black, so that the policy $\rightsquigarrow$ is intransitive. This is certainly an intuitively attractive specification of the desired policy; unfortunately, it does not accurately capture the desired properties. The problem is that noninterference is a very strong property: the assertion Red $\not\rightsquigarrow$ Black means that there must be *no* way for Black to observe activity by Red. This is not what is required here; Black must certainly be able to observe activity by Red (after all, it is the source of all incoming data), but we want all such observations to be mediated by the Bypass or the Crypto.

If the requirement Red $\not\rightsquigarrow$ Black is too strong, it is obvious that the complementary requirement Red $\rightsquigarrow$ Black is too weak: it would allow unrestricted communication from Red to Black.

We conclude that noninterference, as formulated so far, cannot specify channel-control policies exemplified by Figure 1.1. The question, then, is what interpretation is to be placed on intransitive policies within the current formulation? In its simplest form, we ask how we are to interpret assertions such as

$$\begin{array}{rcl} A & \not\rightsquigarrow & C \\ A & \rightsquigarrow & B \\ B & \rightsquigarrow & C. \end{array}$$

The hope is that this policy describes the "assured pipeline" [4] suggested by Figure 3.1. But as we have already seen, this hope is not fulfilled: the requirement $A \not\rightsquigarrow C$ precludes *all* interference by domain A on domain C, including that which would use domain B as an intermediary. The only satisfactory interpretation seems to be one in which the intermediate domain B is internally composed of two isolated parts, $B1$ and $B2$ as suggested in Figure 3.2. A can interfere with the $B1$ part of B (hence $A \rightsquigarrow B$) and the $B2$ part of B can interfere with C (hence $B \rightsquigarrow C$), but the internal dichotomy of B allows $A \not\rightsquigarrow C$. Under this interpretation, however, it is surely more natural to recognize B as two domains and to formulate the policy accordingly:

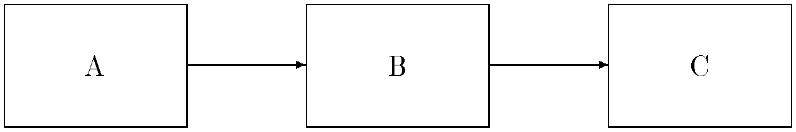

Figure 3.1: Desired interpretation of an intransitive policy

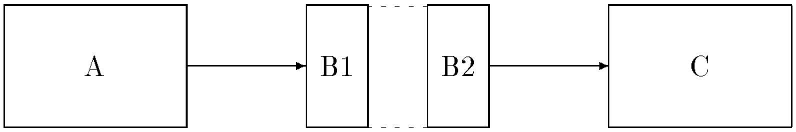

Figure 3.2: Plausible interpretation of an intransitive policy

$$\begin{array}{rcl} A & \not\rightsquigarrow & C \\ A & \rightsquigarrow & B1 \\ B1 & \not\rightsquigarrow & B2 \\ B2 & \rightsquigarrow & C \end{array}$$

But this is (trivially) a transitive policy. We conclude that intransitive policies seem to have no useful interpretation under the present formulation of noninterference.

In the following section, we will develop a formulation of noninterference that does provide a useful interpretation to intransitive policies, and in fact it is an interpretation satisfying the original goal of using noninterference to provide a formal foundation for the specification and verification of channel-control policies. Before we proceed to an examination of intransitive policies, however, we pause to examine the properties of transitive policies.

3.1 Properties of Transitive Policies

To begin, we define the class of *multilevel* security policies that model the systems of clearances and classifications used in the pen-and-paper world.

Definition 6 Let L be a set of *security labels* (comprising "levels," possibly augmented by "compartments") with a partial ordering $\preceq$ (usually read as "is dominated by"). The interpretation of $l_1 \preceq l_2$ is that l_2 is more highly classified (in the case of data), or more highly trusted (in the case of individuals), and that information is permitted to flow from l_1 to l_2, but not vice-versa (unless $l_1 = l_2$).

Let $clearance : D \rightarrow L$ be a function that assigns a fixed security label to each domain in D. Then the *multilevel security (MLS) policy* is:

$$u \rightsquigarrow v \text{ iff } clearance(u) \preceq clearance(v). \tag{3.1}$$

That is, u may interfere with v if the clearance of v dominates that of u.

An arbitrary security policy given by a relation $\rightsquigarrow$ on D is said to be an *MLS-type policy* if a label set L with a partial ordering $\preceq$ and a function $clearance : D \rightarrow L$ can be found such that (3.1) holds. □

Clearly we have:

Theorem 3 *All MLS-type policies are transitive.*

Proof: This follows directly from the transitivity of the partial order $\preceq$. □

The converse is also true. An essentially similar result (using a slightly different construction) was discovered by Dorothy Denning in 1976 [8].

Theorem 4 *All transitive policies are MLS-type policies.*

Proof: Let $\rightsquigarrow$ be a transitive security policy. Define a further relation $\leftrightarrow$ on D by:

$$u \leftrightarrow v \overset{\text{def}}{=} u \rightsquigarrow v \wedge v \rightsquigarrow u.$$

The construction ensures that $\leftrightarrow$ is symmetric. Reflexivity and transitivity of $\leftrightarrow$ follow from that of $\rightsquigarrow$ (recall that all policies are reflexive). Thus $\leftrightarrow$ is an equivalence relation. We identify a label set L with the equivalence classes of $\leftrightarrow$ and use $[u]$ to denote the equivalence class of domain u under $\leftrightarrow$. We define a relation $\preceq$ on L as follows:

$$[u] \preceq [v] \overset{\mathrm{def}}{=} \exists \text{ domains } x \in [u] \text{ and } y \in [v] \text{ such that } x \rightsquigarrow y.$$

It is easy to see that $\preceq$ is a partial order on L (i.e., it is reflexive, transitive, and antisymmetric). Finally, we define the function $\mathit{clearance} : U \rightarrow L$ by

$$\mathit{clearance}(u) \overset{\mathrm{def}}{=} [u].$$

It is then easy to verify that

$$u \rightsquigarrow v \text{ iff } \mathit{clearance}(u) \preceq \mathit{clearance}(v),$$

and so it follows that $\rightsquigarrow$ is an MLS-type policy. □

The access control conditions of Theorem 2 reveal a familiar appearance when they are recast into the notation natural for MLS-type policies. To show this, we must first assign a *classification* label to each storage object by means of a function

- $\mathit{classification}\colon N \rightarrow D$.

Then we have:

Corollary 1 (Bell and La Padula Interpretation) *Let $\rightsquigarrow$ be an MLS-type policy, and M a system with structured state that satisfies the Reference Monitor Assumptions and the following two properties.*

ss-property: $n \in \mathit{observe}(u) \supset \mathit{classification}(n) \preceq \mathit{clearance}(u)$,

$*$-property: $n \in \mathit{alter}(u) \supset \mathit{clearance}(u) \preceq \mathit{classification}(n)$.

Then M is secure for $\rightsquigarrow$.

Proof: Using Theorem 2, we need to prove

$$u \rightsquigarrow v \supset \mathit{observe}(u) \subseteq \mathit{observe}(v),$$

and

$$n \in \mathit{alter}(u) \wedge n \in \mathit{observe}(v) \supset u \rightsquigarrow v.$$

The first of these can be restated as

$$u \rightsquigarrow v \wedge n \in \mathit{observe}(u) \supset n \in \mathit{observe}(v).$$

Using the notation of MLS-type policies and the ss-property, this becomes

$$\begin{aligned} &\mathit{clearance}(u) \preceq \mathit{clearance}(v) \wedge \mathit{classification}(n) \preceq \mathit{clearance}(u) \\ &\quad \supset \quad \mathit{classification}(n) \preceq \mathit{clearance}(v) \end{aligned}$$

and is satisfied immediately by the transitivity of the partial order $\preceq$.

Using the notation of MLS-type policies, the second of the conditions in Theorem 2 becomes

$$n \in \mathit{alter}(u) \wedge n \in \mathit{observe}(v) \supset \mathit{clearance}(u) \preceq \mathit{clearance}(v).$$

Using the ss- and $*$-properties, the antecedent to this implication becomes

$$\mathit{clearance}(u) \preceq \mathit{classification}(n) \wedge \mathit{classification}(n) \preceq \mathit{clearance}(v)$$

and the conclusion then follows from the transitivity of the partial order $\preceq$. □

The ss- and $*$-properties named in this result correspond to the "simple-security" and "star" properties of the Bell and La Padula security model [2, 3]. The simple-security condition asserts that a subject must only be able to observe objects whose classification is dominated by its own clearance, while the star-property asserts the dual condition that it must only be able to alter objects whose classification dominates its own clearance. Since the corollary establishes that these conditions are adequate to ensure the security of a system that enforces an MLS-type policy, it may seem puzzling that the Bell and La Padula formulation is known to have severe weaknesses [17, 18]. In fact, there are two sources for these weaknesses and it may be useful to briefly indicate what they are, and why Corollary 1 is not vulnerable to them.

- One source of weaknesses derives from the lack of a *semantic* characterization of what is meant by "observe" and "alter" in the Bell and La Padula model. It is possible to subvert the model by inverting the intended interpretations of these terms. (So that the simple-security property says the subjects may *alter* only objects of lower classification.) Corollary 1 does not share this weakness because the Reference Monitor Assumptions provide an adequate semantic characterization of the intended interpretation of observe and alter access.

- The other source of weakness concerns the behavior of actions that modify the access control functions. Our notion of a system with structured state is very limited; more realistic models include more implementation detail and also extend the set of access control functions and provide actions for manipulating them. Such actions are called "rules" by Bell and La Padula, who gave a representative set in their Multics interpretation [3]. Two of these rules are known to permit unsecure information flow [19,30]. The reason for this is that the access control "table" and other implementation-level state data of the reference monitor are not treated as objects in the Bell and La Padula model; although the model prevents unsecure information flow through the objects

> that it explicitly recognizes, it places no constraints on the flow of information through the mechanisms of its own realization.
>
> Corollary 1 does not share this weakness because its system model is very limited and does permit the access control tables to change; thus, it admits no "rules." In more complex models, that do permit modification to the access control and other internal tables, the "rules" should be individually verified by direct reference to the appropriate unwinding theorem.

The verification of individual "rules" using the unwinding theorem requires identification of the "views" of the machine state held by different security domains. The next result provides some guidance in the identification of such views, by showing that, for a transitive $\rightsquigarrow$ relation, they are "nested" within each other. This is obvious in the Bell and La Padula model (i.e., everything observable by a subject at level l_1 is also observable to a subject of level l_2 where $l_1 \preceq l_2$). What is interesting here is that Theorem 5 shows that this nesting property is inherent, not accidental.

Definition 7 A view-partitioned machine is said to have the *nesting* property if

$$u \rightsquigarrow v \wedge s \overset{v}{\sim} t \supset s \overset{u}{\sim} t.$$

That is, if states s and t appear identical to domain v, then they also appear identical to those domains u that may interfere with v. □

Theorem 5 *Let $\rightsquigarrow$ be a transitive policy and M a view-partitioned machine which satisfies the conditions of the unwinding theorem. Then there is a nested view-partitioning of M that also satisfies the conditions of the unwinding theorem.*

Proof: Define a new view-partitioning relation $\overset{u}{\simeq}$ on D by

$$s \overset{u}{\simeq} t \overset{\text{def}}{=} (\forall v\colon v \rightsquigarrow u \supset s \overset{v}{\sim} t).$$

That $\overset{u}{\simeq}$ is an equivalence relation follows straightforwardly from the fact that $\overset{u}{\sim}$ is. Output consistency and step consistency of $\overset{u}{\simeq}$ likewise follow from those properties of the $\overset{u}{\sim}$ relation. For $\overset{u}{\simeq}$ to locally respect $\rightsquigarrow$, we require

$$\forall v \rightsquigarrow u : dom(a) \not\rightsquigarrow u \supset s \overset{v}{\sim} step(s, a). \tag{3.2}$$

The transitivity of $\rightsquigarrow$ ensures $dom(a) \not\rightsquigarrow v$ (since otherwise we could combine $dom(a) \rightsquigarrow v$ with $v \rightsquigarrow u$ and contradict $dom(a) \not\rightsquigarrow u$), and (3.2) then follows from the fact that $\overset{v}{\sim}$ locally respects $\rightsquigarrow$.

For the nesting property, we need to demonstrate $x \rightsquigarrow u \supset s \overset{x}{\sim} t$ given $u \rightsquigarrow v$ and $s \overset{v}{\simeq} t$. Transitivity provides $x \rightsquigarrow v$, and the result then follows from the definition of $s \overset{v}{\simeq} t$. □

Finally, we prove that unwinding is, in a certain sense, complete: for *any* secure system, we can find a view-partitioning that satisfies the conditions of the unwinding theorem. Note that this result does not depend on the transitivity of $\rightsquigarrow$, but it does depend on the present interpretation of noninterference which, as we have seen, makes sense only for transitive policies.

Theorem 6 *If M is a secure system, then for each domain $u \in D$ an equivalence relation $\overset{u}{\sim}$ on the set of states can be found that satisfies the conditions of the unwinding theorem.*

Proof: We use the following construction: for $u \in D$ and reachable states s and t, define

$$\begin{aligned} s \overset{u}{\sim} t \quad \overset{\text{def}}{=} \quad & (\forall \alpha \in A^*, b \in A : \mathit{dom}(b) = u \\ & \supset \mathit{output}(\mathit{run}(s, \alpha), b) = \mathit{output}(\mathit{run}(t, \alpha), b)). \end{aligned} \tag{3.3}$$

Clearly, $\overset{u}{\sim}$ is an equivalence relation. Output consistency follows by taking $\alpha = \Lambda$ in (3.3). For step consistency, we need

$$s \overset{u}{\sim} t \supset \mathit{step}(s, a) \overset{u}{\sim} \mathit{step}(t, a).$$

The conclusion to this implication expands to

$$\mathit{output}(\mathit{run}(\mathit{step}(s, a), \alpha), b) = \mathit{output}(\mathit{run}(\mathit{step}(t, a), \alpha), b)$$

and this is equivalent to

$$\mathit{output}(\mathit{run}(s, a \circ \alpha), b) = \mathit{output}(\mathit{run}(t, a \circ \alpha), b),$$

which follows directly from the definition of $s \overset{u}{\sim} t$.

To show that the construction locally respects $\rightsquigarrow$, we need to demonstrate

$$\mathit{dom}(a) \not\rightsquigarrow u \supset s \overset{u}{\sim} \mathit{step}(s, a).$$

The conclusion expands to

$$\mathit{output}(\mathit{run}(s, \alpha), b) = \mathit{output}(\mathit{run}(\mathit{step}(s, a), \alpha), b) \tag{3.4}$$

where $\mathit{dom}(b) = u$. If s is a reachable state, there exists γ such that $s = \mathit{do}(\gamma)$ and so (3.4) can be written as

$$\mathit{test}(\gamma \circ \alpha, b) = \mathit{test}(\gamma \circ a \circ \alpha, b).$$

Since the machine is secure, the definition of noninterference gives

$$\mathit{test}(\gamma \circ \alpha, b) = \mathit{test}(\mathit{purge}(\gamma \circ \alpha, \mathit{dom}(b)), b)$$

and

$$test(\gamma \circ a \circ \alpha, b) = test(purge(\gamma \circ a \circ \alpha, dom(b)), b).$$

But, clearly, since $dom(a) \not\rightsquigarrow u$ and $u = dom(b)$,

$$purge(\gamma \circ \alpha, dom(b)) = purge(\gamma \circ a \circ \alpha, dom(b))$$

and the result follows. □

Chapter 4

Intransitive Noninterference

Goguen and Meseguer recognized the inability of standard noninterference to model channel-control policies and they introduced several extensions to the basic formulation in their second paper on the subject [11]. However, the first really satisfactory treatment of intransitive noninterference policies was given by Haigh and Young [13], with an earlier version the previous year [12].

Both Goguen and Meseguer, and Haigh and Young, recognized that the standard definition of noninterference is too draconian. If $u \not\rightsquigarrow v$, the requirement is that deleting all actions performed by u should produce no change in the behavior of the system as perceived by v. This is too strong if we also have the assertions $u \rightsquigarrow w$ and $w \rightsquigarrow v$. Surely we should only delete those actions of u that are not followed by actions of w: this is the essence of Haigh and Young's reformulation of noninterference. In order to give a formal definition, we need to identify those actions in an action sequence that should not be deleted. This is the purpose of the function *sources*.

Definition 8 We define the function

- $\mathit{sources}\colon A^* \times D \to \mathcal{P}(D)$

by the equations

$$
\begin{array}{rcl}
\mathit{sources}(\Lambda, u) & = & \{u\} \\
\mathit{sources}(a \circ \alpha, u)^1 & = & \left\{ \begin{array}{ll} \mathit{sources}(\alpha, u) \cup \{\mathit{dom}(a)\} & \text{if } \exists v : v \in \mathit{sources}(\alpha, u) \\ & \qquad\qquad \wedge\ \mathit{dom}(a) \rightsquigarrow v \\ \mathit{sources}(\alpha, u) & \text{otherwise.} \end{array} \right.
\end{array}
$$

Our function *sources* corresponds to the function *purgeable* of Haigh and Young [13], although Haigh and Young gave only an informal characterization of

---

[1] This is the definition in which right-recursion is essential.

their function. In essence $v \in \mathit{sources}(\alpha, u)$ means either that $v = u$ or that there is a subsequence of α consisting of actions performed by domains $w_1, w_2, \ldots, w_n$ such that $w_1 \rightsquigarrow w_2 \rightsquigarrow \cdots \rightsquigarrow w_n$, $v = w_1$, and $u = w_n$. In considering whether an action a performed prior to the action sequence α should be allowed to influence u, we ask whether there is any $v \in \mathit{sources}(\alpha, u)$ such that $\mathit{dom}(a) \rightsquigarrow v$. Notice that always

$$\mathit{sources}(\alpha, u) \subseteq \mathit{sources}(a \circ \alpha, u), \text{ and}$$

$$u \in \mathit{sources}(\alpha, u).$$

We can now define the function *ipurge* (for *intransitive*-purge):

- $\mathit{ipurge}\colon A^* \times D \rightarrow A^*$

by the equations

$$\begin{array}{rcl} \mathit{ipurge}(\Lambda, u) & = & \Lambda \\ \mathit{ipurge}(a \circ \alpha, u) & = & \left\{ \begin{array}{ll} a \circ \mathit{ipurge}(\alpha, u) & \text{if } \mathit{dom}(a) \in \mathit{sources}(a \circ \alpha, u) \\ \mathit{ipurge}(\alpha, u) & \text{otherwise.} \end{array} \right. \end{array}$$

Informally, $\mathit{ipurge}(\alpha, u)$ consists of the subsequence of α with all those actions that should not be able to interfere with u removed. Thus, security is now defined in terms of the *ipurge* function:

A machine is *secure* for the policy $\rightsquigarrow$ if

$$\mathit{test}(\alpha, a) = \mathit{test}(\mathit{ipurge}(\alpha, \mathit{dom}(a)), a).$$

$\square$

From this point on, our treatment diverges from that of Haigh and Young. We will argue later that their treatment is incorrect. The first step is to establish the revised form of Lemma 1.

Lemma 2 *Let* $\rightsquigarrow$ *be a policy and* M *a view-partitioned, output consistent system such that,*

$$\mathit{do}(\alpha) \overset{u}{\sim} \mathit{do}(\mathit{ipurge}(\alpha, u)).$$

Then M *is secure for* $\rightsquigarrow$*.*

Proof: The proof is essentially identical to that of Lemma 1.

Setting $u = \mathit{dom}(a)$ in the statement of the lemma gives

$$\mathit{do}(\alpha) \overset{\mathit{dom}(a)}{\sim} \mathit{do}(\mathit{ipurge}(\alpha, \mathit{dom}(a))),$$

and output consistency then provides

$$output(do(\alpha), a) = output(do(ipurge(\alpha, dom(a))), a).$$

But this is simply

$$test(\alpha, a) = test(ipurge(\alpha, dom(a)), a),$$

which is the definition of security for $\rightsquigarrow$ given by Definition 8. □

Next, we present a series of definitions and lemmas that culminate in the revised form of the unwinding theorem.

Definition 9 Let M be a view-partitioned system and $C \subseteq D$ a set of domains. We define the equivalence relation $\overset{C}{\approx}$ on the states of M as follows:

$$s \overset{C}{\approx} t \overset{\text{def}}{=} (\forall u \in C : s \overset{u}{\sim} t).$$

Thus $s \overset{C}{\approx} t$ exactly when the states s and t appear identical to all the members of C. □

Definition 10 Let M be a view-partitioned system and $\rightsquigarrow$ a policy. We say that M is *weakly step consistent* if

$$s \overset{u}{\sim} t \wedge s \overset{dom(a)}{\sim} t \supset step(s, a) \overset{u}{\sim} step(t, a).$$

□

Lemma 3 *Let $\rightsquigarrow$ be a policy and M a view-partitioned system which is weakly step consistent, and locally respects $\rightsquigarrow$. Then*

$$s \overset{sources(a \circ \alpha, u)}{\approx} t \supset step(s, a) \overset{sources(\alpha, u)}{\approx} step(t, a).$$

Proof: Suppose $v \in sources(\alpha, u)$. We need to show that

$$step(s, a) \overset{v}{\sim} step(t, a). \tag{4.1}$$

Note that $v \in sources(\alpha, u)$ implies $v \in sources(a \circ \alpha, u)$, and so the hypothesis to the lemma provides

$$s \overset{v}{\sim} t. \tag{4.2}$$

We now consider two cases.

Case 1: $dom(a) \leadsto v$**.** Then, by the definition of the *sources* function, we have $dom(a) \in sources(a \circ \alpha, u)$ and the hypothesis to the lemma provides

$$s \overset{dom(a)}{\sim} t. \tag{4.3}$$

(4.1) then follows from (4.2) and (4.3) by weak step consistency.

Case 2: $dom(a) \not\leadsto v$**.** Then by local respect for $\not\leadsto$,

$$\begin{aligned} step(s,a) &\overset{v}{\sim} s, \\ step(t,a) &\overset{v}{\sim} t \end{aligned}$$

and (4.1) follows from (4.2).

□

Lemma 4 *Let* $\leadsto$ *be a policy and* M *a view-partitioned system that locally respects* $\leadsto$*. Then*

$$dom(a) \notin sources(a \circ \alpha, u) \supset s \overset{sources(\alpha,u)}{\approx} step(s,a).$$

Proof: We assume the hypothesis and let $v \in sources(\alpha, u)$. It must be that $dom(a) \not\leadsto v$, since otherwise $dom(a) \in sources(a \circ \alpha, u)$. Hence, by local respect for $\not\leadsto$,

$$s \overset{v}{\sim} step(s,a)$$

and the conclusion follows. □

Lemma 5 *Let* $\leadsto$ *be a policy and* M *a view-partitioned system which is weakly step consistent, and locally respects* $\leadsto$*. Then*

$$s \overset{sources(\alpha,u)}{\approx} t \supset run(s,\alpha) \overset{u}{\sim} run(t, ipurge(\alpha,u)).$$

Proof: The proof proceeds by induction on the length of α. The basis is the case $\alpha = \Lambda$ and follows straightforwardly by application of definitions. For the inductive step, we assume the result for α of length n, and consider $a \circ \alpha$. We then need to show

$$s \overset{sources(a\circ\alpha,u)}{\approx} t \supset run(s, a \circ \alpha) \overset{u}{\sim} run(t, ipurge(a \circ \alpha, u)).$$

We now consider two cases.

Case 1: $dom(a) \in sources(a \circ \alpha, u)$. Then $ipurge(a \circ \alpha, u) = a \circ ipurge(\alpha, u)$ and we need to show

$$s \overset{sources(a\circ\alpha,u)}{\approx} t \supset run(step(s,a),\alpha) \overset{u}{\sim} run(step(t,a), ipurge(\alpha,u)).$$

Lemma 3 gives

$$s \overset{sources(a\circ\alpha,u)}{\approx} t \supset step(s,a) \overset{sources(\alpha,u)}{\approx} step(t,a)$$

and the result then follows from the inductive hypothesis.

Case 2: $dom(a) \notin sources(a \circ \alpha, u)$. Then $ipurge(a \circ \alpha, u) = ipurge(\alpha, u)$ and we need to show

$$s \overset{sources(a\circ\alpha,u)}{\approx} t \supset run(step(s,a),\alpha) \overset{u}{\sim} run(t, ipurge(\alpha,u)).$$

Now Lemma 4 gives

$$dom(a) \notin sources(a \circ \alpha, u) \supset s \overset{sources(\alpha,u)}{\approx} step(s,a)$$

and, since $sources(\alpha, u) \subseteq sources(a \circ \alpha, u)$, $s \overset{sources(a\circ\alpha,u)}{\approx} t$ implies

$$s \overset{sources(\alpha,u)}{\approx} t.$$

Because $\overset{sources(\alpha,u)}{\approx}$ is an equivalence relation, it follows that

$$step(s,a) \overset{sources(\alpha,u)}{\approx} t$$

and the result then follows from the inductive hypothesis.

□

Finally, we can present the unwinding theorem for intransitive noninterference policies.

Theorem 7 (Unwinding Theorem for Intransitive Policies) *Let* $\rightsquigarrow$ *be a policy and* M *a view-partitioned system that is*

1. *is output consistent,*
2. *weakly step consistent, and*
3. *locally respects* $\rightsquigarrow$*.*

Then M is secure for $\rightsquigarrow$.

Proof: Taking $s = t = s_0$ in Lemma 5 gives

$$run(s_0, \alpha) \overset{u}{\sim} run(s_0, ipurge(\alpha, u)),$$

which can be rewritten in the form

$$do(\alpha) \overset{u}{\sim} do(ipurge(\alpha, u)),$$

so that the conclusion follows from Lemma 2. □
A formal verification of this theorem has been performed using the EHDM specification and verification system and is described in in the Appendix to this report. The mechanically checked proof follows the argument of Lemmas 3 to 5 very closely.

In the following chapter, we consider the differences and similarities between this unwinding theorem and both the ordinary unwinding theorem and that of Haigh and Young. Before doing so, however, we note that the access control mechanism described in Definition 5 on page 12 of Chapter 2 works for intransitive noninterference policies as well as for transitive ones.

Theorem 8 *Let M be a system with structured state that satisfies the Reference Monitor Assumptions and the condition*

$$n \in alter(u) \wedge n \in observe(v) \supset u \rightsquigarrow v.$$

Then M is secure for $\rightsquigarrow$.

Proof: The proof is similar to that of Theorem 2. We show that the conditions of the theorem satisfy those of the intransitive unwinding theorem. We identify the view-partitioning relations $\overset{u}{\sim}$ of the Intransitive Unwinding Theorem with the corresponding relations defined in the statement of the Reference Monitor Assumptions. Output consistency is then satisfied immediately by the first of the Reference Monitor Assumptions.

To establish weak step consistency, we must prove

$$s \overset{u}{\sim} t \wedge s \overset{dom(a)}{\sim} t \supset step(s, a) \overset{u}{\sim} step(t, a).$$

This can be rewritten as

$$s \overset{u}{\sim} t \wedge s \overset{dom(a)}{\sim} t \supset contents(step(s, a), n) = contents(step(t, a), n)$$

where $n \in observe(u)$. There are three cases to consider

Case 1: $contents(step(s, a), n) \neq contents(s, n)$. The second of the Reference Monitor Assumptions provides the desired conclusion directly (from the hypothesis $s \overset{dom(a)}{\sim} t$).

Case 2: $contents(step(t, a), n) \neq contents(t, n)$. This case is symmetrical with the first.

Case 3: $contents(step(t, a), n) = contents(t, n) \wedge contents(step(t, a), n) = contents(t, n)$. Since $s \overset{u}{\sim} t$, we have $contents(s, n) = contents(t, n)$ and the conclusion is immediate.

It remains to show that the construction locally respects $\rightsquigarrow$. This follows by exactly the same argument as that used in the proof of Theorem 2. □

It is illuminating to examine the similarity between this access control theorem and the ordinary one (Theorem 2). The only difference between the two theorems is that the ordinary one requires the additional condition

$$u \rightsquigarrow v \supset observe(u) \subseteq observe(v).$$

Theorem 8 is able to dispense with this condition because the intransitive unwinding theorem, from which it is derived, requires only *weak* step consistency.

To see how this apparently small difference in formulation allows Theorem 8, but not Theorem 2, to provide an access control interpretation for an intransitive policy, consider the system sketched in Figure 3.1. Theorem 8, allows domain A to have alter access to locations to which domain B has observe access. Similarly, it permits domain B to have alter access to locations to which domain C has observe access. In this way, information can flow from A to B and from B to C. However, A may not have alter access to any locations to which C has observe access; in this way, direct flow of information from A to C is prevented.

The conditions of Theorem 2 also allow domain A to have alter access to locations to which domain B has observe access, but they also require that B have observe access to every location to which A has observe access. Similarly, considering domains B and C, the conditions of Theorem 2 require that C have observe access to every location to which B has observe access. Transitively, therefore, C has observe access to every location to which A has observe access and so A can have "no secrets" from C. Thus, the additional condition of Theorem 2 forces the transitive completion of the policy, and so allows the direct flow of information from A to C.

Chapter 5

Comparisons among the Formulations

In this chapter we compare our treatment of intransitive noninterference policies with the standard treatment of noninterference and with that of Haigh and Young.

5.1 Intransitive vs. Standard Noninterference

We first compare our treatment of intransitive noninterference policies (Chapter 4) with the standard treatment of noninterference policies (Chapter 2) and the special properties of transitive policies (Chapter 3). We will show that, when restricted to transitive policies, our formulation of noninterference corresponds exactly with the standard treatment. This provides some assurance that our treatment is a natural extension of the standard one. To begin, we establish that the definitions of security coincide in the case of transitive polices.

Lemma 6 *If $\rightsquigarrow$ is transitive, then*

$$v \in \mathit{sources}(\alpha, u) \supset v \rightsquigarrow u.$$

Proof: The proof is by induction on the length of α. The basis is the case $\alpha = \Lambda$, and reference to Definition 8 shows that

$$\mathit{sources}(\Lambda, u) = \{u\}$$

and the lemma is satisfied in this case by the reflexivity of $\rightsquigarrow$.

For the inductive step, Definition 8 gives $v \in \mathit{sources}(a \circ \alpha, u)$ if either $v \in \mathit{sources}(\alpha, u)$ or

$$v = \mathit{dom}(a) \wedge (\exists w \in \mathit{sources}(\alpha, u) \wedge \mathit{dom}(a) \rightsquigarrow w).$$

In the first case, the inductive hypothesis provides $v \leadsto u$ directly; in the second, the inductive hypothesis provides $w \leadsto u$, we also have $v = dom(a)$ and $dom(a) \leadsto w$, and so transitivity provides $v \leadsto u$ as required. □

Lemma 7 *If* $\leadsto$ *is transitive, then* $ipurge(\alpha, u) = purge(\alpha, u)$.

Proof: Comparison of Definitions 2 and 8 reveals that we only need to demonstrate

$$dom(a) \leadsto u \text{ iff } dom(a) \in sources(a \circ \alpha, u).$$

The "if" direction was established by the previous lemma. For the "only if" direction, note that $u \in sources(\alpha, u)$, so that $dom(a) \in sources(a \circ \alpha, u)$ follows immediately from Definition 8 and $dom(a) \leadsto u$. □

Theorem 9 *Definitions 2 and 8 of security agree when the relation* $\leadsto$ *is transitive.*

Proof: Since the two definitions differ only in their "purge" functions, this result is an immediate consequence of the previous lemma. □

We now know that the two definitions of security coincide in the case of transitive policies; next, we show that the unwinding theorems do so as well.

Theorem 10 *The Unwinding Theorems 1 and 7 agree when the relation* $\leadsto$ *is transitive.*

Proof: The unwinding theorems differ only in that the intransitive version uses weak step consistency where the regular one uses (ordinary) step consistency. Weak step consistency is the condition

$$s \overset{u}{\sim} t \wedge s \overset{dom(a)}{\sim} t \supset step(s, a) \overset{u}{\sim} step(t, a),$$

while ordinary step consistency is the condition

$$s \overset{u}{\sim} t \supset step(s, a) \overset{u}{\sim} step(t, a).$$

Ordinary step consistency obviously implies weak step consistency; thus, we only need to show that weak step consistency implies ordinary step consistency when $\leadsto$ is transitive. However, it is not necessarily the case that a given view partitioning that satisfies weak step consistency also satisfies ordinary step consistency; thus we must prove that a view partitioning satisfying the intransitive unwinding theorem implies the existence of (another) view partitioning satisfying the ordinary unwinding theorem.

The construction we use is the same as that for the nesting theorem (Theorem 5): we define a new view-partitioning relation $\overset{u}{\simeq}$ on D by

$$s \overset{u}{\simeq} t \overset{\text{def}}{=} (\forall v\colon v \rightsquigarrow u \supset s \overset{v}{\sim} t).$$

The output consistency and local respect for $\rightsquigarrow$ of $\overset{u}{\simeq}$ follow by the same arguments used in Theorem 5, as does the fact that $\overset{u}{\simeq}$ is an equivalence relation. For (ordinary) step consistency, we must show that

$$s \overset{u}{\simeq} t \supset \mathit{step}(s, a) \overset{u}{\simeq} \mathit{step}(t, a),$$

or, equivalently,

$$s \overset{u}{\simeq} t \wedge v \rightsquigarrow u \supset \mathit{step}(s, a) \overset{v}{\sim} \mathit{step}(t, a).$$

Note that $s \overset{u}{\simeq} t \wedge v \rightsquigarrow u \supset s \overset{v}{\sim} t$. There are now two cases to consider.

Case 1: $\mathit{dom}(a) \rightsquigarrow u$. In this case, $s \overset{u}{\simeq} t$ implies $s \overset{\mathit{dom}(a)}{\sim} t$, and since we already have $s \overset{v}{\sim} t$, weak step consistency then supplies $\mathit{step}(s, a) \overset{v}{\sim} \mathit{step}(t, a)$ as required.

Case 2: $\mathit{dom}(a) \not\rightsquigarrow u$. In this case, since we have $v \rightsquigarrow u$, transitivity of $\rightsquigarrow$ requires $\mathit{dom}(a) \not\rightsquigarrow v$. But then, local respect of $\rightsquigarrow$ by $\overset{v}{\sim}$ requires $\mathit{step}(s, a) \overset{v}{\sim} s$ and $\mathit{step}(t, a) \overset{v}{\sim} t$, and so $\mathit{step}(s, a) \overset{v}{\sim} \mathit{step}(t, a)$ follows directly from $s \overset{v}{\sim} t$.

□

5.2 Comparison with Haigh and Young's Formulation

The system model used by Haigh and Young [13] differs slightly from that used here. Their *output* function has signature

- $\mathit{output}\colon S \times D \to O$

whereas we use

- $\mathit{output}\colon S \times A \to O.$

Thus, their *output* function allows a domain u to inspect the system state s directly as $\mathit{output}(s, u)$, whereas ours requires the mediation of an action a with $\mathit{dom}(a) = u$ to form $\mathit{output}(s, a)$. Converting our formulation to theirs requires a corresponding change in the definition of the function *test* to signature

- $\mathit{test}\colon A^* \times D \to O$

with definition

$$test(\alpha, u) = output(do(\alpha), u).$$

The definition of security becomes

$$test(\alpha, u) = test(ipurge(\alpha, u), u),$$

and that of output consistency changes to

$$s \overset{u}{\sim} t \supset output(s, u) = output(t, u).$$

Some small changes are then needed in the proof of Lemma 2 in order to take account of the modified function signatures. No other changes are needed in the development. We have checked this by modifying the formal verification of the Appendix in the manner described above and then re-running all the proofs. The ability to readily check the effect of changed assumptions in this way is one of the great benefits of formal verification: assumptions are recorded with great precision and the "ripple" effect of perturbations can be evaluated mechanically.

Since the slight differences between the system model used here and that used by Haigh and Young have only a trivial impact on the definition of intransitive noninterference, and none at all on our intransitive unwinding theorem, it is reasonable to compare our definitions and theorems with those of Haigh and Young.

Under the proviso that our function *sources* is the same as their informally defined function *purgeable*, our definition for intransitive noninterference is the same as that given by Haigh and Young for "MDS Security." However, the corresponding unwinding theorems differ and in this section we compare our unwinding theorem for intransitive policies with the "SAT MDS Unwinding Theorem" of Haigh and Young.

In our terminology and notation, the SAT MDS Unwinding Theorem of Haigh and Young is the following.

Proposition 1 (SAT MDS Unwinding Theorem) *Let* $\rightsquigarrow$ *be a policy and* M *a view-partitioned system that is*

1. *is output consistent,*
2. *step consistent, and*
3. *MDS-respects* $\rightsquigarrow$*.*

Then M *is secure for* $\rightsquigarrow$*.*

That is, Haigh and Young require step consistency where we require *weak* step consistency, and they require a condition we call "MDS-respect" for $\rightsquigarrow$ where we require local respect. The condition MDS-respect is defined as follows by Haigh and Young [13, p. 147, formula (10)].

Definition 11 Let M be a view-partitioned system and $\rightsquigarrow$ a policy. We say that M *MDS-respects* $\rightsquigarrow$ if, for any choice of action a and state s, if a is purgeable with respect to domain u, then

$$s \overset{u}{\sim} step(s, a).$$

□

This definition presents a considerable challenge to interpretation. The function *purgeable* is not defined formally by Haigh and Young, but in its informal definition, and in all previous uses within their paper, it is used in contexts such as "a is purgeable with respect to u in α." That is, the purgeability of an action is defined relative to a domain *and* an action sequence. In the definition of MDS-respects, however, there is no reference to an action sequence. Examination of Haigh and Young's proof of their SAT MDS Unwinding Theorem sheds no light on the interpretation of the crucial notion MDS-respects: the proof is only a sketch and does not employ formal use of definitions.

Any interpretation of MDS-respects that differs from locally respects must be either weaker or stronger than that alternative notion. A stronger notion would require $s \overset{u}{\sim} step(s, a)$ even in some circumstances where $dom(a) \rightsquigarrow u$. This does not seem very plausible, since the other conditions of the SAT MDS Unwinding Theorem are the same as for the ordinary unwinding theorem, and strengthening one of them must restrict, rather than enlarge, the class of policies admitted. We conclude that MDS-respects must allow $s \overset{u}{\not\sim} step(s, a)$ in some circumstances where $dom(a) \not\rightsquigarrow u$. The constraint on the possible values of $step(s, a)$ in this case must be provided by the other conditions of the theorem, namely output consistency, and step consistency. However, as these are both the same as in the ordinary noninterference case, it is difficult to see how adequate constraints on the effect of a state transition $step(s, a)$ with $dom(a) \not\rightsquigarrow u$ and $s \overset{u}{\not\sim} step(s, a)$ can be achieved by these constraints.

In contrast, our formulation of the unwinding theorem for intransitive policies leaves the locally respects constraint unchanged from the ordinary case, but changes the step consistency constraint to *weak* step consistency. That is, the condition:

$$s \overset{u}{\sim} t \supset step(s, a) \overset{u}{\sim} step(t, a)$$

of the ordinary case is changed to

$$s \overset{u}{\sim} t \wedge s \overset{dom(a)}{\sim} t \supset step(s, a) \overset{u}{\sim} step(t, a)$$

for the intransitive case.

The second of these conditions is very natural: its intuitive interpretation is that when an action a is performed, those elements of the system state visible to u change in a way that depends only on those same elements, plus those visible to the domain that performed the action.

The ordinary step consistency condition requires that when an action a is performed, those elements of the system state visible to u change in a way that depends on those elements *alone*. This seems more, not less, restrictive than the previous case, until we recall that for transitive policies there is always a view-partitioning that satisfies

$$u \rightsquigarrow v \wedge s \overset{v}{\sim} t \supset s \overset{u}{\sim} t.$$

In other words, those elements of the state space visible to u include all the elements of the state space visible to domains that may interfere with u.

We should now ask whether a similar explanation can provide a sound interpretation to Haigh and Young's SAT MDS Unwinding Theorem. We believe not, and we use the following example to make our case. Consider a system with four domains U, V, W, and X; U and V may interfere with W, and W may interfere with X, but U and V must not directly interfere with X. The system state is composed of three internal registers, u, v, and x, all initially zero. Each domain has a single action associated with it: U's action sets the register u to 1, V's action sets the register v to 2, W's action sets the register x to the sum of the contents of u and v, and X's action outputs the contents of the register x. It should be clear that this system satisfies the stated policy. We need to be able to distinguish it from the insecure variant in which X's action outputs the sum of the registers u and v directly. In our formulation of intransitive noninterference, U, V and X's view of the system state is restricted to the registers u, v and x respectively, while W can view both registers u and v. It is easy to see that our unwinding theorem for intransitive policies is satisfied by this assignment.

Haigh and Young's unwinding theorem is not satisfied, however, since the effect of W's action on the register x cannot be explained in terms of the objects visible to X. It seems that the set of objects visible to X must be enlarged to include the registers u and v. But how, then, are we to distinguish the system from its unsecure variant?

We conclude that all possible interpretations of Haigh and Young's SAT MDS Unwinding Theorem are unsatisfactory. Because there is no precise definition of the crucial requirement that we call "MDS-respects," it is impossible to assign a definitive status to the theorem, and its utility becomes questionable.

We have, we believe, presented adequate evidence that our unwinding theorem for intransitive policies is both true and useful; indeed, we believe it is the strongest theorem possible. We have also presented evidence that Haigh and Young's theorem is essentially different than ours—differing in the crucial step consistency condition, not just the uncertain MDS-respects condition. We therefore believe it unlikely that their theorem, if true, is as generally applicable as ours. Consequently, we consider it likely that their theorem is either false, or true but applicable to a very small class of systems and/or policies.

Chapter 6

Summary and Conclusions

We have examined the issue of transitivity in noninterference security policies. Intransitive noninterference policies would seem, intuitively, to be exactly what is required for the formal specification of channel control and type enforcement policies. We have shown, however, that the standard interpretation of noninterference does not fulfill this expectation. Fortunately, the interpretation of noninterference introduced by Haigh and Young for multidomain security (MDS) does have the properties we require. Our contribution has been the identification of intransitivity of the $\rightsquigarrow$ relation as the key distinction between channel control, type enforcement, and MDS policies on the one hand, and MLS policies on the other.

It can be considered a historical accident that the theory for the transitive case was invented and developed before the intransitive one, and has therefore become regarded as the standard case. We submit that it is now more helpful to regard the intransitive case as the basis for noninterference formulations of security, with the formerly standard treatment regarded as a specialization for the case of transitive policies. The advantage of regarding the development in this light is that one does not have to trouble with the rather difficult and informal argument that the standard treatment makes little sense for intransitive policies; one can simply present the general theory and then show that there is a simpler treatment available in the special case of transitive policies. The attempt to use the standard treatment in the case of intransitive policies simply does not arise with this approach.

Our main technical contributions have been the formulation, rigorous proof, and mechanically-checked formal verification of an unwinding theorem for intransitive polices, a demonstration that the definitions and theorems of the intransitive theory collapse to the standard ones in the case of transitive policies, and an exploration of the properties of transitive policies. Our demonstrations of the equivalence of MLS and transitive noninterference policies, of the nesting property, and of the result that all MLS secure systems satisfy the conditions of the unwinding theorem, shed some new light on the properties of transitive noninterference security policies.

However, the novel and more interesting case, and the one that prompted this investigation in the first place, is that of intransitive noninterference policies. In future work we hope to explore the practical application of intransitive noninterference formulations to problems of channel control, and to develop effective methods for verifying mechanisms that enforce such policies. We also plan to explore the connection between intransitive noninterference policies and the class of properties, discussed in [26], that can be enforced by kernelization.

Bibliography

[1] D. H. Barnes. The provision of security for user data on packet switched networks. In *Proceedings of the Symposium on Security and Privacy*, pages 121–126, Oakland, CA, April 1983. IEEE Computer Society.

[2] D. E. Bell and L. J. La Padula. Secure computer systems: Vol. I—mathematical foundations, Vol. II—a mathematical model, Vol. III—a refinement of the mathematical model. Technical Report MTR-2547 (three volumes), Mitre Corporation, Bedford, MA, March–December 1973.

[3] D. E. Bell and L. J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, Bedford, MA, March 1976.

[4] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings 8th DoD/NBS Computer Security Initiative Conference*, pages 18–27, Gaithersburg, MD, September 1985.

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

[6] EHDM *Specification and Verification System Version 5.0—Description of the* EHDM *Specification Language*. Computer Science Laboratory, SRI International, Menlo Park, CA, January 1990. See [7] for the updates to Version 5.2.

[7] EHDM *Specification and Verification System Version 5.X—Supplement to User's and Language Manuals*. Computer Science Laboratory, SRI International, Menlo Park, CA, August 1991. Current version number is 5.2.

[8] D. E. Denning. On the derivation of lattice structured information flow policies. Technical Report CSD TR 180, Purdue University, March 1976.

[9] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *6th ACM Symposium on Operating System Principles*, pages 57–65, November 1977.

[10] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 1982. IEEE Computer Society.

[11] J. A. Goguen and J. Meseguer. Inference control and unwinding. In *Proceedings of the Symposium on Security and Privacy*, pages 75–86, Oakland, CA, April 1984. IEEE Computer Society.

[12] J. Haigh and W. Young. Extending the non-interference model of MLS for SAT. In *Proceedings of the Symposium on Security and Privacy*, pages 232–239, Oakland, CA, April 1986. IEEE Computer Society.

[13] J. Thomas Haigh and William D. Young. Extending the noninterference version of MLS for SAT. *IEEE Transactions on Software Engineering*, SE-13(2):141–150, February 1987.

[14] Jeremy Jacob. A note on the use of separability for the detection of covert channels. *Cipher (Newsletter of the IEEE Technical Committee on Security and Privacy)*, pages 25–33, Summer 1989.

[15] Nancy L. Kelem and Richard J. Feiertag. A separation model for virtual machine monitors. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 78–86, Oakland, CA, May 1991. IEEE Computer Society.

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

[17] John McLean. A comment on the "basic security theorem" of Bell and La Padula. *Information Processing Letters*, 20:67–70, 1985.

[18] John McLean. Reasoning about security models. In *Proceedings of the Symposium on Security and Privacy*, pages 123–131, Oakland, CA, April 1987. IEEE Computer Society.

[19] J. K. Millen and C. M. Cerniglia. Computer security models. Working Paper WP25068, Mitre Corporation, Bedford, MA, September 1983.

[20] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752, Saratoga, NY, 1992. Springer Verlag, volume 607 of *Lecture Notes in Artificial Intelligence*.

[21] Gerald J. Popek and David R. Farber. A model for verification of data security in operating systems. *Communications of the ACM*, 21(9):737–749, September 1978.

[22] John Rushby. The design and verification of secure systems. In *8th ACM Symposium on Operating System Principles*, pages 12–21, Asilomar, CA, December 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[23] John Rushby. Verification of secure systems. Technical Report 166, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, August 1981.

[24] John Rushby. Proof of Separability—a verification technique for a class of security kernels. In *Proc. 5th International Symposium on Programming*, pages 352–367, Turin, Italy, April 1982. Springer Verlag, volume 137 of *Lecture Notes in Computer Science.*

[25] John Rushby. The security model of Enhanced HDM. In *Proceedings 7th DoD/NBS Computer Security Initiative Conference*, pages 120–136, Gaithersburg, MD, September 1984.

[26] John Rushby. Kernels for safety? In T. Anderson, editor, *Safe and Secure Computing Systems*, chapter 13, pages 210–220. Blackwell Scientific Publications, 1989. (Proceedings of a Symposium held in Glasgow, October 1986).

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

[28] C. T. Sennett and R. Macdonald. Separability and security models. Technical Report 87020, Royal Signals and Radar Establishment, Malvern, UK, November 1987.

[29] Joseph R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, MA, 1967.

[30] T. Taylor. Comparison paper between the Bell and La Padula model and the SRI model. In *Proceedings of the Symposium on Security and Privacy*, pages 195–202, Oakland, CA, April 1984. IEEE Computer Society.

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

Appendix: Formal Verification

Formulation of noninterference and derivation of the corresponding unwinding theorem is surprisingly intricate for the case of intransitive security policies. We have shown that our treatment differs somewhat from that of Haigh and Young, and have argued that their unwinding theorem is incorrect. Because of its intricacy, we presented our development in detail and described the proofs fully in the main body of this report. We also presented collateral evidence for the correctness of our results, by showing that they collapse to the familiar ones in the case of transitive policies. In this Appendix, we present additional evidence for the correctness of our development in the form of a mechanically checked proof for the intransitive unwinding theorem.[1] The proof was performed using the EHDM formal specification and verification system developed by the Computer Science Laboratory of SRI [5,6,27,31].[2]

Although we do not claim that verification in EHDM is a certification of "correctness," the successful outcome of the formal verification increases our confidence that the theorem is correct. This confidence derives from the greater understanding that a truly formal development requires as much as it does from mechanized checking of the proofs.

A secondary benefit of formal verification in EHDM is the enumeration, by its "Proof Chain Checker," of all the definitions and axioms on which a proof depends. In this way, we are able to identify the logical foundation of a verification; those definitions and axioms comprising the foundation can then be subjected to careful scrutiny and peer review. The logical foundation of the Intransitive Unwinding Theorem is listed in Section C.1 and can be seen to comprise 14 definitions needed to develop the noninterference model, plus one definition and six axioms required for the supporting theories such as lists and sets.

Another benefit of a formal verification is that it can assist in the exploration of alternative formulations and specifications. For example, we noted in Chapter 5.2

---

[1] We have also developed a mechanically-checked formal verification of the unwinding theorem for classical (i.e., transitive) noninterference. This verification, which is much simpler than the one described here, is available from the author on request.

[2] This formal specification and verification was performed in early 1991 using the then-current Version 5.2 of EHDM. The now-current Version 6.1 of EHDM is a completely new implementation that differs in some significant details from 5.2.

that our basic system model differs slightly from that of Haigh and Young [13]. When this was drawn to our attention, we were able to incorporate the changes in our formal specification and explore its consequences very quickly. In this case, we found that the proof to one theorem needed slight adjustment, but everything else could be left unchanged. In our experience, manual checking of the consequences of revised definitions and axioms is very error-prone, since proofs are seldom checked for the second time (to see if they need adjusting) with the same care that they are constructed in the first place. In contrast, a verification system explores the "ripple" effect of changes with mechanical precision and speed.

We consider that the benefits of the formal specification and verification of the intransitive noninterference model and its unwinding theorem amply repay the modest cost required. It really does not take much longer to develop a fully formal, mechanically checked specification in EHDM than it does to construct a comparably detailed model in conventional mathematical notation. Simple parsing and typechecking of the specification is sufficient to identify mismatches between the definition and use of terms, such as those in Haigh and Young's MDS Unwinding Theorem [13]. Furthermore, in an expressive specification language such as that of EHDM, the specifications are perfectly readable and compare well with those developed in conventional, informal, mathematical notation. The additional discipline of a formal specification also encourages simplicity and consistency of notation and presentation.

Appendix A

Description of the Formal Specification and Verification

The formal verification described here was performed using the EHDM system; as we do not describe EHDM in any detail here, readers unfamiliar with its specification language and verification environment are referred to [27]. The formal specification and verification follows closely the conventional mathematical presentation in Chapter 4, using the same notation and names wherever possible.

The formal specification and verification in EHDM is shown in full in Appendix D, and a cross-reference in Appendix B. Notice that EHDM proof declarations appear as part the specification: they provide a list of premises and ground substitutions for variables, and constitute the only information provided to the theorem prover. A "proof-chain" analysis of the verification appears in Appendix C. This analysis checks that the premises to each theorem are either axioms or proved theorems. Type-correctness of certain EHDM constructions (for example, recursive function definitions) requires that certain system-generated formulas called *Type Consistency Conditions* (TCCs) are proved; in addition, some modules (such as those defining induction) declare assumptions that must be proved in any instantiation of the module. The proof-chain analyzer checks that both these kinds of proof obligations are discharged.

The generation of all the tables in the appendices was performed automatically by EHDM. In addition, the specifications that appear in Appendix D were prettyprinted into conventional mathematical notation using the EHDM LaTeX-translator [7, 27].

A.1 Lists

Inspection of the development in Chapter 4 shows that several functions range over sequences, but that recursions and inductions over these sequences always occur right-recursively. Thus we do not require a general theory of sequences, only a theory of sequences constructed right-recursively; such a theory is the theory of *lists* with constructor *cons* (which we generally write as infix $\circ$), base element *nil* (which we generally write as Λ), and selectors *car* and *cdr*.[1] A *length* function is also defined on lists.

This theory is presented in the module `lists` shown on page 62. Observe that the module is parameterized by the type over which the lists are constructed. It is worth noting that a rather large number of axioms (six) are required in this module. Axioms must always be scrutinized with great care since they can introduce inconsistencies. In fact, the module `lists` can be systematically generated from a simple definitional facility for abstract data types (this is done automatically in PVS [20], our other verification system) and the soundness of the construction can be proved once and for all. For illustrative purposes, however, we specifically demonstrate the consistency of the `lists` theory by exhibiting a model. In EHDM, this is done via theory interpretation [29, Section 4.7] using the `MAPPING` mechanism [27]. The construction of such an interpretation is described in the following section.

A.1.1 The Consistency of the Lists Specification

The topic discussed in this section is somewhat technical and may be skipped without loss to the main theme.

We demonstrate the consistency of the `lists` module, and also confirm our understanding of what it specifies, by mapping it to a constructively defined module called `lists_model` shown on page 63. The latter module interprets lists as records consisting of a natural number (intuitively, this can be regarded as a pointer) and an array. The list is "stored" in consecutive locations of the array, starting at 0. The pointer points one location past the end of the list. The *cons* function simply stores a new element in the array and advances the pointer; conversely, *cdr* simply reduces the pointer. The *car* function returns the value of the location in the list immediately prior to the pointer. Arrays and records in EHDM are simply functions; "storing" a value in an array or record is performed using function modification, indicated by the `with` keyword: `f with [(x) := y]` is the function that has the same value everywhere as `f`, except that it has the value `y` at `x`.

---

[1] Although adequate for our purposes, this really is a very limited theory of lists; in fact, it is rather closer to a theory of stacks.

The specification of these functions in the module `lists_model` shown on page 63 is straightforward.[2] The module `lists_model` generates a TCC module shown on page 64; an adequate proof of its single TCC formula is given in the module `top` shown on page 81.

It is also necessary to define a *concrete-equality* predicate on the lists representation type; this predicate must be true when two representations both represent the same abstract list. This will be so if their pointers are the same and the contents of their arrays are equal at all locations between 0 and one less than the pointer. The predicate is specified as `ce` in the module `lists_model`. It is a matter of pragmatic convenience that `ce` and `cons` are specified by ordinary definitions (using a single `=`), while the other functions are specified by literal definitions (using double `==`): the literally-defined functions will be expanded automatically in proofs, whereas the more complex `ce` and `cons` must be cited explicitly.

The mapping module that links `lists` to `lists_model` is the module `lists_map` shown on page 65. This module explicitly indicates that equality on lists is to be mapped to the `ce` predicate in the interpretation; all other types and constants are to be mapped using name-identity. The system-generated mapped module `lists_map_map` is shown on page 66. The axioms of `lists`, interpreted in the theory of `lists_model` as indicated by `lists_map`, become formulas to be proven in `lists_map_map`. Notice that we are required to prove that `ce` is an equivalence relation. We should also be required to prove that it satisfies the property of substitutivity and hence is a congruence relation, but this check was missing from EHDM Version 5.2 (however, it is enforced in Version 6).

EHDM automatically generates trivial proof declarations for the formulas in mapped and TCC modules. Often, these suffice, but in the present case the trivial system-generated proofs in `lists_map_map` are inadequate; effective proofs are provided in the module `lists_map_proofs` shown on page 67.

A.1.2 List Inductions

The module `list_inductions` on page 68 states the higher-order formula `list_induction` that specifies an induction scheme used to prove properties of lists:

$$\text{list\_induction: } \textbf{Theorem } p(\Lambda) \wedge (\forall\, \alpha, x\colon p(\alpha) \supset p(x \circ \alpha)) \supset p(\gamma)$$

where p is an arbitrary predicate over lists. This formula is given as a theorem and proven from the formula `general_induction`, which states the general scheme for Nötherian induction and is given in the module `noetherian` on page 69. The module

---

[2] This entire construction is very similar to that described for *stacks* in the EHDM tutorial [27, Chapter 5]; readers seeking further explanation of mappings in EHDM should consult that description.

`noetherian` is a library module of EHDM and is described in detail in the EHDM tutorial [27, Chapter 6]. Interested readers should refer to the discussion in the tutorial for an explanation of the `well_founded` assumption in the module `noetherian`, and the manner in which it is discharged by the module `list_inductions`. The formula `general_induction` is stated as an axiom and justified by reference to standard texts (for example [16, page 6]).

A.2 Sets

The module `sets` (page 70) introduces sets and the basic set operations of union, intersection, subset, and the like. Sets are modeled by their characteristic predicates and the set operations are defined as higher-order functions. Those unfamiliar with the use of higher-order logic in specifications may find these definitions particularly interesting. Notice that the type of (the predicate representing) a given set is dependent on the type supplied as the actual parameter to the `sets` module.

A.3 Noninterference

The specification of intransitive noninterference and the proof of its unwinding theorem are presented in the module `intrans_nonint` starting on page 71. The specification begins by introducing the basic types and functions used to state the noninterference notion of security. The names of the types and functions used are the same as those in the conventional mathematical development given in the main body of this report, and the definitions are likewise fairly straightforward transliterations. The main differences occur in the specifications of the functions `run`, `sources`, and `ipurge`: whereas these are defined by (pattern matching) cases over the list constructors Λ and $\circ$ in the conventional mathematical presentation, the EHDM specification defines these functions recursively. Recursive definitions in EHDM require *measure functions* to be supplied in order to ensure that the recursion is well-founded. The measure functions used here, `step_count` and `step_count2`, employ the length of the list as the measure. The TCCs generated from these recursive definitions appear in the system-generated module `intrans_nonint_tcc` on page 78, and effective proofs (the system-generated proofs are inadequate) are given in the module `intrans_nonint_tcc_proofs` on page 80.

The equivalence relation that induces view-partitioning is written in infix notation in the form $s \overset{u}{\sim} t$ in the conventional mathematical presentation. In the EHDM specification it is written as `view_id(u, ss, tt)` and is specified as

$$\text{view\_id: function}[\mathcal{D}, \mathcal{S}, \mathcal{S} \rightarrow \text{bool}] \equiv (\,\lambda\, u, \text{st}, \text{tt}\colon \text{view}(u, \text{st}) = \text{view}(u, \text{tt}))$$

where `view` is an uninterpreted function. The pragmatic advantage of this method of specification is that `view_id` expands to an equation; thus we do not need to cite the properties of reflexivity, symmetry, and transitivity in proofs that use `view_id`.

The predicates `secure`, `output_consistent`, and `local_respect` are the formal counterparts of the properties of similar names given in the conventional mathematical presentation. The predicate `view_consistent` is introduced to name the main condition in the statement of Lemma 2. In specifying these four predicates, there is a choice as to which of the variables should be locally quantified and which should be specified as parameters in the predicate definition and quantified at a higher level. We chose to locally quantify all variables in these predicates, except the list variables in `view_consistent` and `secure`. This is really a matter of taste and other choices would work equally well.

Lemmas 2 to 5 in the conventional mathematical development are mirrored by the formulas with similar names in the formal development. The proofs of Lemmas 2 to 4 in the formal verification correspond almost directly to those in the conventional mathematical development. However, the formal verification interposes between the proofs of Lemmas 2 and 3 the statements and proofs of a number of minor technical results that are taken as obvious in the conventional mathematical presentation. The first six of these:

single_step_lemma: **Lemma** $\mathrm{run}(\mathrm{st}, a \circ \alpha) = \mathrm{run}(\mathrm{step}(\mathrm{st}, a), \alpha)$

purge_lemma: **Lemma**
$\mathrm{ipurge}(a \circ \alpha, u)$
$=$ **if** $\mathrm{dom}(a) \in \mathrm{sources}(a \circ \alpha, u)$
then $a \circ \mathrm{ipurge}(\alpha, u)$
else $\mathrm{ipurge}(\alpha, u)$
end if

sources_subset: **Lemma** $\mathrm{sources}(\alpha, u) \subseteq \mathrm{sources}(a \circ \alpha, u)$

sources_grows: **Lemma** $v \in \mathrm{sources}(\alpha, u) \supset v \in \mathrm{sources}(a \circ \alpha, u)$

sources_defn_base_case: **Lemma** $\mathrm{sources}(\Lambda, u) = \{u\}$

sources_defn_inductive_case: **Lemma**
$(\exists\, v{:}\, v \in \mathrm{sources}(\alpha, u) \wedge \mathrm{dom}(a) \rightsquigarrow v)$
$\supset \mathrm{sources}(a \circ \alpha, u) = \{\mathrm{dom}(a)\} \cup \mathrm{sources}(\alpha, u)$

are obvious and have straightforward proofs. However, the next one

in_own_sources: **Lemma** $u \in \mathrm{sources}(\alpha, u)$

requires a proof by induction and uses several intermediate lemmas. When performing a proof by induction in EHDM, it is usually convenient to define a predicate equivalent to all, or part of, the formula to be proven. The predicate should be parameterized by the induction variable in order to allow inductive instances to be written conveniently. In the present specification, the predicate `in_own_sources_pred` fulfills this role with respect to the formula `in_own_sources`.

The predicate `strong_view_id` stands in the same relationship to the equivalence relation $\overset{C}{\approx}$ in the conventional mathematical presentation as the predicate `view_id` does to $\overset{u}{\sim}$. However, because `strong_view_id` does not reduce to an equation, the properties of reflexivity, symmetry, and transitivity have to proven, and later cited, explicitly. The technical lemma

$$\begin{array}{l} \text{strong\_view\_id\_sources\_prop: } \textbf{Lemma} \\ \quad \text{strong\_view\_id}(\text{sources}(a \circ \alpha, u), \text{st}, \text{tt}) \\ \qquad \supset \text{strong\_view\_id}(\text{sources}(\alpha, u), \text{st}, \text{tt}) \end{array}$$

is also proven at this point. Following the proofs of Lemmas 3 and 4, come the definitions and lemmas that establish Lemma 5. The proof is by induction, and we introduce a predicate `lemma5_pred` to simplify its expression and proof. The module ends with the proof of the unwinding theorem. As in the conventional mathematical presentation, this is a straightforward consequence of Lemmas 2 and 5.

A.4 Top

This module serves to tie the main modules of the specification together. It also contains a proof for the TCC formula `car_TCC1` from the module `lists_model_tcc` shown on page 64. The proof cites the formula `nat_invariant`, which is the subtype invariant for natural numbers—i.e., $(\lambda n : n \geq 0)$—from the standard prelude.

Appendix B

Cross-Reference Listing

This Appendix provides two cross-reference tables to assist in reading and navigating the EHDM specifications that follow. The first provides a cross-reference listing to the identifiers declared in the EHDM specification; the second provides the translations from raw EHDM identifiers appearing in the first table and in Appendix B to the symbols appearing in the LaTeX-printed version of the specifications given in Appendix C. All the material appearing in these Appendices was generated mechanically by EHDM.

Identifier	Declaration	Module
`A`	type	`intrans_nonint`
`add`	literal-fn	`sets`
`arbitrary`	const	`lists_model`
`Astar`	type	`intrans_nonint`
`car`	function	`lists`
`car`	literal-fn	`lists_model`
`car_ax`	axiom	`lists`
`car_ax`	formula	`lists_map_map`
`car_ax_PROOF`	prove	`lists_map_map`
`car_ax_PROOF`	prove	`lists_map_proofs`
`car_TCC1`	formula	`lists_model_tcc`
`car_TCC1_PROOF`	prove	`lists_model_tcc`
`cdr`	function	`lists`
`cdr`	literal-fn	`lists_model`
`cdr_ax`	axiom	`lists`
`cdr_ax`	formula	`lists_map_map`
`cdr_ax_PROOF`	prove	`lists_map_map`
`cdr_ax_PROOF`	prove	`lists_map_proofs`
`cdr_cons_lemma1`	formula	`lists_map_proofs`
`cdr_cons_lemma1_proof`	prove	`lists_map_proofs`
`cdr_cons_lemma2`	formula	`lists_map_proofs`
`cdr_cons_lemma2_proof`	prove	`lists_map_proofs`
`ce`	defined-fn	`lists_model`
`ce_isreflexive`	formula	`lists_map_map`
`ce_isreflexive_PROOF`	prove	`lists_map_map`
`ce_isreflexive_PROOF`	prove	`lists_map_proofs`
`ce_issymmetric`	formula	`lists_map_map`
`ce_issymmetric_PROOF`	prove	`lists_map_map`
`ce_issymmetric_PROOF`	prove	`lists_map_proofs`
`ce_istransitive`	formula	`lists_map_map`
`ce_istransitive_PROOF`	prove	`lists_map_map`
`ce_istransitive_PROOF`	prove	`lists_map_proofs`
`connects`	defined-fn	`intrans_nonint`
`cons`	defined-fn	`lists_model`
`cons`	function	`lists`
`cons_ax`	axiom	`lists`
`cons_ax`	formula	`lists_map_map`
`cons_ax_PROOF`	prove	`lists_map_map`
`cons_ax_PROOF`	prove	`lists_map_proofs`
`cons_induction_proof`	prove	`list_inductions`
`D`	type	`intrans_nonint`
`difference`	literal-fn	`sets`
`discharge_well_founded`	prove	`list_inductions`
`dof`	defined-fn	`intrans_nonint`

Table B.1: EHDM Identifers used in the Specification (continues)

Identifier	Declaration	Module
`dom`	function	`intrans_nonint`
`empty`	defined-fn	`sets`
`emptyset`	literal-const	`sets`
`extends`	literal-fn	`list_inductions`
`extensionality`	axiom	`sets`
`fullset`	literal-const	`sets`
`general_induction`	axiom	`noetherian`
`in_own_sources`	formula	`intrans_nonint`
`in_own_sources_basis`	formula	`intrans_nonint`
`in_own_sources_basis_proof`	prove	`intrans_nonint`
`in_own_sources_form`	formula	`intrans_nonint`
`in_own_sources_form_proof`	prove	`intrans_nonint`
`in_own_sources_induct`	formula	`intrans_nonint`
`in_own_sources_induct_proof`	prove	`intrans_nonint`
`in_own_sources_pred`	defined-fn	`intrans_nonint`
`in_own_sources_proof`	prove	`intrans_nonint`
`interfere`	function	`intrans_nonint`
`intersection`	literal-fn	`sets`
`intrans_nonint`	module	`intrans_nonint`
`intrans_nonint_tcc`	module	`intrans_nonint_tcc`
`intrans_nonint_tcc_proofs`	module	`intrans_nonint_tcc_proofs`
`ipurge`	recursive-fn	`intrans_nonint`
`ipurge_TCC1`	formula	`intrans_nonint_tcc`
`ipurge_TCC1_PROOF`	prove	`intrans_nonint_tcc`
`ipurge_TCC2`	formula	`intrans_nonint_tcc`
`ipurge_TCC2_PROOF`	prove	`intrans_nonint_tcc`
`lemma2`	formula	`intrans_nonint`
`lemma2_proof`	prove	`intrans_nonint`
`lemma3`	formula	`intrans_nonint`
`lemma3_proof`	prove	`intrans_nonint`
`lemma4`	formula	`intrans_nonint`
`lemma4_proof`	prove	`intrans_nonint`
`lemma5`	formula	`intrans_nonint`
`lemma5_basis`	formula	`intrans_nonint`
`lemma5_basis_proof`	prove	`intrans_nonint`
`lemma5_induct`	formula	`intrans_nonint`
`lemma5_induct_proof`	prove	`intrans_nonint`
`lemma5_pred`	defined-fn	`intrans_nonint`
`lemma5_proof`	prove	`intrans_nonint`
`length`	function	`lists`
`length`	literal-fn	`lists_model`
`length_cdr`	formula	`lists`
`length_cdr_proof`	prove	`lists`
`length_cons`	axiom	`lists`

Table B.1: EHDM Identifers used in the Specification (continues)

Identifier	Declaration	Module
`length_cons`	formula	`lists_map_map`
`length_cons_PROOF`	prove	`lists_map_map`
`length_cons_PROOF`	prove	`lists_map_proofs`
`length_nil`	axiom	`lists`
`length_nil`	formula	`lists_map_map`
`length_nil_PROOF`	prove	`lists_map_map`
`length_nil_PROOF`	prove	`lists_map_proofs`
`list`	type	`lists_model`
`list`	type	`lists`
`listarray`	type	`lists_model`
`list_induction`	formula	`list_inductions`
`list_inductions`	module	`list_inductions`
`lists`	module	`lists`
`lists_map`	module	`lists_map`
`lists_map_map`	module	`lists_map_map`
`lists_map_proofs`	module	`lists_map_proofs`
`lists_model`	module	`lists_model`
`lists_model_tcc`	module	`lists_model_tcc`
`local_respect`	defined-const	`intrans_nonint`
`member`	literal-fn	`sets`
`nil`	const	`lists_model`
`nil`	const	`lists`
`nilax`	axiom	`lists_model`
`nilorcons_ax`	axiom	`lists`
`nilorcons_ax`	formula	`lists_map_map`
`nilorcons_ax_PROOF`	prove	`lists_map_map`
`nilorcons_ax_PROOF`	prove	`lists_map_proofs`
`noetherian`	module	`noetherian`
`noninterfere`	literal-fn	`intrans_nonint`
`null`	literal-fn	`lists`
`O`	type	`intrans_nonint`
`output`	function	`intrans_nonint`
`output_consistent`	defined-const	`intrans_nonint`
`purge_lemma`	formula	`intrans_nonint`
`purge_lemma_proof`	prove	`intrans_nonint`
`purge_TCC1_PROOF`	prove	`intrans_nonint_tcc_proofs`
`purge_TCC2_PROOF`	prove	`intrans_nonint_tcc_proofs`
`run`	recursive-fn	`intrans_nonint`
`run_TCC1`	formula	`intrans_nonint_tcc`
`run_TCC1_PROOF`	prove	`intrans_nonint_tcc`
`run_TCC1_PROOF`	prove	`intrans_nonint_tcc_proofs`
`S`	type	`intrans_nonint`
`secure`	defined-fn	`intrans_nonint`
`set`	type	`sets`

Table B.1: EHDM Identifers used in the Specification (continues)

Identifier	Declaration	Module
`sets`	module	`sets`
`single_step_lemma`	formula	`intrans_nonint`
`single_step_lemma_proof`	prove	`intrans_nonint`
`singleton`	literal-fn	`sets`
`sources`	recursive-fn	`intrans_nonint`
`sources_defn_base_case`	formula	`intrans_nonint`
`sources_defn_base_case_proof`	prove	`intrans_nonint`
`sources_defn_inductive_case`	formula	`intrans_nonint`
`sources_defn_inductive_case_proof`	prove	`intrans_nonint`
`sources_grows`	formula	`intrans_nonint`
`sources_grows_proof`	prove	`intrans_nonint`
`sources_subset`	formula	`intrans_nonint`
`sources_subset_proof`	prove	`intrans_nonint`
`sources_TCC1`	formula	`intrans_nonint_tcc`
`sources_TCC1_PROOF`	prove	`intrans_nonint_tcc`
`sources_TCC1_PROOF`	prove	`intrans_nonint_tcc_proofs`
`sources_TCC2`	formula	`intrans_nonint_tcc`
`sources_TCC2_PROOF`	prove	`intrans_nonint_tcc`
`sources_TCC2_PROOF`	prove	`intrans_nonint_tcc_proofs`
`sources_TCC3`	formula	`intrans_nonint_tcc`
`sources_TCC3_PROOF`	prove	`intrans_nonint_tcc`
`sources_TCC3_PROOF`	prove	`intrans_nonint_tcc_proofs`
`st0`	const	`intrans_nonint`
`step`	function	`intrans_nonint`
`step_count`	literal-fn	`intrans_nonint`
`step_count2`	literal-fn	`intrans_nonint`
`strong_view_id`	defined-fn	`intrans_nonint`
`strong_view_id_reflexive`	formula	`intrans_nonint`
`strong_view_id_reflexive_proof`	prove	`intrans_nonint`
`strong_view_id_sources_prop`	formula	`intrans_nonint`
`strong_view_id_sources_prop_proof`	prove	`intrans_nonint`
`strong_view_id_symmetric`	formula	`intrans_nonint`
`strong_view_id_symmetric_proof`	prove	`intrans_nonint`
`strong_view_id_transitive`	formula	`intrans_nonint`
`strong_view_id_transitive_proof`	prove	`intrans_nonint`
`subset`	defined-fn	`sets`
`subset_add`	formula	`sets`
`subset_add_proof`	prove	`sets`
`subset_self`	formula	`sets`
`subset_self_proof`	prove	`sets`
`TCC_proof`	prove	`top`
`test`	defined-fn	`intrans_nonint`
`top`	module	`top`
`union`	literal-fn	`sets`

Table B.1: EHDM Identifers used in the Specification (continues)

Identifier	Declaration	Module
`unwinding`	formula	`intrans_nonint`
`unwinding_proof`	prove	`intrans_nonint`
`V`	type	`intrans_nonint`
`value`	const	`lists_model`
`view`	function	`intrans_nonint`
`view_consistent`	defined-fn	`intrans_nonint`
`view_id`	literal-fn	`intrans_nonint`
`weak_step_consistent`	defined-const	`intrans_nonint`
`well_founded`	formula	`noetherian`

Table B.1: EHDM Identifers used in the Specification

Identifier	Translation
`A`	$\mathcal{A}$
`add`	$\{x\} \cup a$
`alist`	α
`Astar`	$\mathcal{A}^{\star}$
`cons`	$s \circ \alpha$
`D`	$\mathcal{D}$
`difference`	$a - b$
`dof`	do
`emptyset`	$\emptyset$
`I`	$\mathcal{I}$
`interfere`	$u \rightsquigarrow v$
`intersection`	$a \bigcap b$
`member`	$x \in b$
`nil`	Λ
`noninterfere`	$u \not\rightsquigarrow v$
`O`	$\mathcal{O}$
`singleton`	$\{x\}$
`S`	$\mathcal{S}$
`st0`	st_0
`subset`	$a \subseteq b$
`union`	$a \bigcup b$

Table B.2: Translations for Identifiers Used in the Specification

Appendix C

Proof-Chain Analysis

The following pages reproduce the output from the EHDM proof-chain analyzer in terse mode applied first to the formula `unwinding` in module `intrans_nonint` and then to all the formulas in the mapped module `lists_map_map`. The EHDM proof-chain analyzer examines the macroscopic structure of a verification by checking that all the premises used in a proof are either axioms, definitions, or formulas which are, themselves, the target of a successful proof elsewhere in the verification. If any formulas are used from a module with an `assuming` clause, then the proof-chain analyzer checks that those assumptions are discharged by successful proofs; similarly, if formulas are used from a module with a TCC module, then the proof-chain analyzer checks that all the TCCs in that module are discharged by successful proofs. The trivial system-generated proof declarations in the TCC module itself are often unsuccessful, so the user must supply more adequate proofs in another module (TCC modules cannot be altered). The proof-chain analyzer ignores unsuccessful proofs, such as system-generated TCC proofs, when a successful proof for the same formula can be found. The terse mode output reproduced here provides a commentary on only the "interesting" cases, namely proof obligations involving assuming clauses and TCCs, and a summary. A shortcoming of the analysis is that literal constants (i.e., those defined with a double `==`) are not reported. All the proofs listed in the summary were performed by the EHDM theorem prover in checking mode.

C.1 Proof-Chain for the Unwinding Theorem

The following pages reproduce the output from the EHDM proof-chain analyzer applied to the formula `unwinding` in module `intrans_nonint`. It can be seen that the proof chain is complete.

```
Terse proof chain for formula unwinding in module intrans_nonint

Use of the formula
  intrans_nonint.unwinding
requires the following TCCs to be proven
```

```
  intrans_nonint_tcc.run_TCC1
  intrans_nonint_tcc.sources_TCC1
  intrans_nonint_tcc.sources_TCC2
  intrans_nonint_tcc.sources_TCC3
  intrans_nonint_tcc.ipurge_TCC1
  intrans_nonint_tcc.ipurge_TCC2

Formula intrans_nonint_tcc.run_TCC1 is a termination TCC for intrans_nonint.run
Proof of
  intrans_nonint_tcc.run_TCC1
must not use
  intrans_nonint.run

Formula intrans_nonint_tcc.sources_TCC1 is a termination TCC for
intrans_nonint.sources
Proof of
  intrans_nonint_tcc.sources_TCC1
must not use
  intrans_nonint.sources

Formula intrans_nonint_tcc.sources_TCC2 is a termination TCC for
intrans_nonint.sources
Proof of
  intrans_nonint_tcc.sources_TCC2
must not use
  intrans_nonint.sources

Formula intrans_nonint_tcc.sources_TCC3 is a termination TCC for
intrans_nonint.sources
Proof of
  intrans_nonint_tcc.sources_TCC3
must not use
  intrans_nonint.sources

Formula intrans_nonint_tcc.ipurge_TCC1 is a termination TCC for
intrans_nonint.ipurge
Proof of
  intrans_nonint_tcc.ipurge_TCC1
must not use
  intrans_nonint.ipurge

Formula intrans_nonint_tcc.ipurge_TCC2 is a termination TCC for
intrans_nonint.ipurge
Proof of
  intrans_nonint_tcc.ipurge_TCC2
must not use
  intrans_nonint.ipurge
```

```
Use of the formula
  noetherian[lists[...].list, list_inductions[...].extends].general_induction
requires the following assumptions to be discharged
  noetherian[lists[...].list, list_inductions[...].extends].well_founded

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

The proof chain is complete

The axioms and assumptions at the base are:
  lists[EXPR].car_ax
  lists[EXPR].cdr_ax
  lists[EXPR].cons_ax
  lists[EXPR].length_cons
  lists[EXPR].nilorcons_ax
  noetherian[EXPR, EXPR].general_induction
Total: 6

The definitions and type-constraints are:
  intrans_nonint.connects
  intrans_nonint.dof
  intrans_nonint.in_own_sources_pred
  intrans_nonint.ipurge
  intrans_nonint.lemma5_pred
  intrans_nonint.local_respect
  intrans_nonint.output_consistent
  intrans_nonint.run
  intrans_nonint.secure
  intrans_nonint.sources
  intrans_nonint.strong_view_id
  intrans_nonint.test
  intrans_nonint.view_consistent
  intrans_nonint.weak_step_consistent
  sets[EXPR].subset
Total: 15

The formulae used are:
  intrans_nonint.in_own_sources
  intrans_nonint.in_own_sources_basis
  intrans_nonint.in_own_sources_form
  intrans_nonint.in_own_sources_induct
  intrans_nonint.lemma2
  intrans_nonint.lemma3
  intrans_nonint.lemma4
  intrans_nonint.lemma5
  intrans_nonint.lemma5_basis
  intrans_nonint.lemma5_induct
  intrans_nonint.purge_lemma
```

```
  intrans_nonint.single_step_lemma
  intrans_nonint.sources_defn_base_case
  intrans_nonint.sources_defn_inductive_case
  intrans_nonint.sources_grows
  intrans_nonint.sources_subset
  intrans_nonint.strong_view_id_reflexive
  intrans_nonint.strong_view_id_sources_prop
  intrans_nonint.strong_view_id_symmetric
  intrans_nonint.strong_view_id_transitive
  intrans_nonint.unwinding
  intrans_nonint_tcc.ipurge_TCC1
  intrans_nonint_tcc.ipurge_TCC2
  intrans_nonint_tcc.run_TCC1
  intrans_nonint_tcc.sources_TCC1
  intrans_nonint_tcc.sources_TCC2
  intrans_nonint_tcc.sources_TCC3
  list_inductions[EXPR].list_induction
  lists[EXPR].length_cdr
  noetherian[lists[...].list, list_inductions[...].extends].well_founded
  sets[EXPR].subset_add
  sets[EXPR].subset_self
Total: 32

The completed proofs are:
  intrans_nonint.in_own_sources_basis_proof
  intrans_nonint.in_own_sources_form_proof
  intrans_nonint.in_own_sources_induct_proof
  intrans_nonint.in_own_sources_proof
  intrans_nonint.lemma2_proof
  intrans_nonint.lemma3_proof
  intrans_nonint.lemma4_proof
  intrans_nonint.lemma5_basis_proof
  intrans_nonint.lemma5_induct_proof
  intrans_nonint.lemma5_proof
  intrans_nonint.purge_lemma_proof
  intrans_nonint.single_step_lemma_proof
  intrans_nonint.sources_defn_base_case_proof
  intrans_nonint.sources_defn_inductive_case_proof
  intrans_nonint.sources_grows_proof
  intrans_nonint.sources_subset_proof
  intrans_nonint.strong_view_id_reflexive_proof
  intrans_nonint.strong_view_id_sources_prop_proof
  intrans_nonint.strong_view_id_symmetric_proof
  intrans_nonint.strong_view_id_transitive_proof
  intrans_nonint.unwinding_proof
  intrans_nonint_tcc_proofs.purge_TCC1_PROOF
  intrans_nonint_tcc_proofs.purge_TCC2_PROOF
  intrans_nonint_tcc_proofs.run_TCC1_PROOF
```

```
  intrans_nonint_tcc_proofs.sources_TCC1_PROOF
  intrans_nonint_tcc_proofs.sources_TCC2_PROOF
  intrans_nonint_tcc_proofs.sources_TCC3_PROOF
  list_inductions[EXPR].cons_induction_proof
  list_inductions[EXPR].discharge_well_founded
  lists[EXPR].length_cdr_proof
  sets[EXPR].subset_add_proof
  sets[EXPR].subset_self_proof
Total: 32
```

C.2 Proof-Chain for the Mapping of the Lists Module

The following pages reproduce the output from the EHDM proof-chain analyzer applied to all the formulas in the mapped module `lists_map_map`. It can be seen that the proof chain is complete, thereby demonstrating the soundness of the mapping.

```
Terse proof chains of all formulas in module lists_map_map

Use of the formula
  lists_model[EXPR].ce
requires the following TCCs to be proven
  lists_model_tcc[EXPR].car_TCC1

                        SUMMARY

The proof chain is complete

The axioms and assumptions at the base are:
  lists_model[EXPR].nilax
Total: 1

The definitions and type-constraints are:
  lists_model[EXPR].ce
  lists_model[EXPR].cons
  naturalnumbers.nat_invariant
Total: 3

The formulae used are:
  lists_map_map[EXPR].car_ax
  lists_map_map[EXPR].cdr_ax
  lists_map_map[EXPR].ce_isreflexive
  lists_map_map[EXPR].ce_issymmetric
  lists_map_map[EXPR].ce_istransitive
  lists_map_map[EXPR].cons_ax
  lists_map_map[EXPR].length_cons
```

```
  lists_map_map[EXPR].length_nil
  lists_map_map[EXPR].nilorcons_ax
  lists_map_proofs[EXPR].cdr_cons_lemma1
  lists_map_proofs[EXPR].cdr_cons_lemma2
  lists_model_tcc[EXPR].car_TCC1
Total: 12

The completed proofs are:
  lists_map_proofs[EXPR].car_ax_PROOF
  lists_map_proofs[EXPR].cdr_ax_PROOF
  lists_map_proofs[EXPR].cdr_cons_lemma1_proof
  lists_map_proofs[EXPR].cdr_cons_lemma2_proof
  lists_map_proofs[EXPR].ce_isreflexive_PROOF
  lists_map_proofs[EXPR].ce_issymmetric_PROOF
  lists_map_proofs[EXPR].ce_istransitive_PROOF
  lists_map_proofs[EXPR].cons_ax_PROOF
  lists_map_proofs[EXPR].length_cons_PROOF
  lists_map_proofs[EXPR].length_nil_PROOF
  lists_map_proofs[EXPR].nilorcons_ax_PROOF
  top[EXPR].TCC_proof
Total: 12
```

Appendix D

Specifications

lists: **Module** [$\mathcal{T}$: **Type**]

Exporting all

Theory

list: **Type**

Λ: list

s, t: **Var** $\mathcal{T}$

ρ: **Var** list

null: function[list $\rightarrow$ bool] $==$ ($\lambda\, \rho : \rho = \Lambda$)

$s \circ \rho$: function[$\mathcal{T}$, list $\rightarrow$ list]

car: function[list $\rightarrow \mathcal{T}$]

cdr: function[list $\rightarrow$ list]

cons_ax: **Axiom** $\neg(s \circ \rho = \Lambda)$

car_ax: **Axiom** $\mathrm{car}(s \circ \rho) = s$

cdr_ax: **Axiom** $\mathrm{cdr}(s \circ \rho) = \rho$

nilorcons_ax: **Axiom** $\rho = \Lambda \vee \rho = \mathrm{car}(\rho) \circ \mathrm{cdr}(\rho)$

length: function[list $\rightarrow$ nat]

length_nil: **Axiom** $\mathrm{length}(\Lambda) = 0$

length_cons: **Axiom** $\mathrm{length}(s \circ \rho) = \mathrm{length}(\rho) + 1$

length_cdr: **Lemma** $\neg(\mathrm{null}(\rho)) \supset \mathrm{length}(\rho) > \mathrm{length}(\mathrm{cdr}(\rho))$

Proof

length_cdr_proof: **Prove** length_cdr **from**
nilorcons_ax, length_cons $\{\rho \leftarrow \mathrm{cdr}(\rho@\mathrm{p1}),\ s \leftarrow \mathrm{car}(\rho@\mathrm{p1})\}$

End lists

```
lists_model: Module [T: Type]

Exporting all

Theory

  listarray: Type = array[nat]of T

  list: Type = Record nextpos : nat,
                      value : listarray
                end record

  arbitrary: T

  Λ: list

  nilax: Axiom Λ.nextpos = 0

  s, t: Var T

  ρ, σ: Var list

  n: Var nat

  ce: function[list, list → bool] =
     ( λ ρ, σ :
          ρ.nextpos = σ.nextpos
            ∧ (∀ n : n < ρ.nextpos ⊃ ρ.value(n) = σ.value(n)))

  s ∘ ρ: function[T, list → list] =
     ( λ s, ρ : ρ
          with [(value) := ρ.value with [(ρ.nextpos) := s],
                (nextpos) := ρ.nextpos + 1])

  car: function[list → T] ==
     ( λ ρ :  if ρ.nextpos = 0
              then arbitrary
              else ρ.value(ρ.nextpos − 1)
              end if)

  cdr: function[list → list] ==
     ( λ ρ :  if ρ.nextpos = 0
              then Λ
              else ρ
              with [(nextpos) := ρ.nextpos − 1]
              end if)

  length: function[list → nat] == ( λ ρ : ρ.nextpos)

End lists_model
```

lists_model_tcc: **Module** [$\mathcal{T}$: **Type**]

Using lists_model[$\mathcal{T}$]

Exporting all with lists_model

Theory

ρ: **Var** list

```
(* Subtype TCC generated for the first argument to rho in car  AND
   Subtype TCC generated for cdr *)
```

car_TCC1: **Formula** $(\neg(\rho.\mathrm{nextpos} = 0)) \supset (\rho.\mathrm{nextpos} - 1 \geq 0)$

Proof

car_TCC1_PROOF: **Prove** car_TCC1

End lists_model_tcc

lists_map: **Module** [$\mathcal{T}$: **Type**]

Mapping lists[$\mathcal{T}$] **onto** lists_model[$\mathcal{T}$]

=[list[$\mathcal{T}$]] → ce

End lists_map

lists_map_map: **Module** [$\mathcal{T}$: **Type**]

Using lists_model[$\mathcal{T}$]

Exporting all with lists_model[$\mathcal{T}$]

Theory

ρ: **Var** lists_model[$\mathcal{T}$].list

s: **Var** $\mathcal{T}$

x_1: **Var** lists_model[$\mathcal{T}$].list

x_2: **Var** lists_model[$\mathcal{T}$].list

x_3: **Var** lists_model[$\mathcal{T}$].list

ce_isreflexive: **Formula** $\mathrm{ce}(x_1, x_1)$

ce_issymmetric: **Formula** $\mathrm{ce}(x_1, x_2) \supset \mathrm{ce}(x_2, x_1)$

ce_istransitive: **Formula** $\mathrm{ce}(x_1, x_2) \wedge \mathrm{ce}(x_2, x_3) \supset \mathrm{ce}(x_1, x_3)$

cons_ax: **Formula** $\neg(\mathrm{ce}(s \circ \rho, \Lambda))$

car_ax: **Formula** $\mathrm{car}(s \circ \rho) = s$

cdr_ax: **Formula** $\mathrm{ce}(\mathrm{cdr}(s \circ \rho), \rho)$

nilorcons_ax: **Formula** $\mathrm{ce}(\rho, \Lambda) \vee \mathrm{ce}(\rho, \mathrm{car}(\rho) \circ \mathrm{cdr}(\rho))$

length_nil: **Formula** $\mathrm{length}(\Lambda) = 0$

length_cons: **Formula** $\mathrm{length}(s \circ \rho) = \mathrm{length}(\rho) + 1$

Proof

ce_isreflexive_PROOF: **Prove** ce_isreflexive

ce_issymmetric_PROOF: **Prove** ce_issymmetric

ce_istransitive_PROOF: **Prove** ce_istransitive

cons_ax_PROOF: **Prove** cons_ax

car_ax_PROOF: **Prove** car_ax

cdr_ax_PROOF: **Prove** cdr_ax

nilorcons_ax_PROOF: **Prove** nilorcons_ax

length_nil_PROOF: **Prove** length_nil

length_cons_PROOF: **Prove** length_cons

End lists_map_map

lists_map_proofs: **Module** [$\mathcal{T}$: **Type**]

Using lists_map_map[$\mathcal{T}$]

Proof

ce_isreflexive_PROOF: **Prove** ce_isreflexive **from**
ce $\{\rho \leftarrow x_1,\ \sigma \leftarrow x_1\}$

ce_issymmetric_PROOF: **Prove** ce_issymmetric **from**
ce $\{\rho \leftarrow x_2,\ \sigma \leftarrow x_1\}$,
ce $\{\rho \leftarrow x_1,\ \sigma \leftarrow x_2,\ n \leftarrow n@p1\}$

ce_istransitive_PROOF: **Prove** ce_istransitive **from**
ce $\{\rho \leftarrow x_1,\ \sigma \leftarrow x_3\}$,
ce $\{\rho \leftarrow x_2,\ \sigma \leftarrow x_3,\ n \leftarrow n@p1\}$,
ce $\{\rho \leftarrow x_1,\ \sigma \leftarrow x_2,\ n \leftarrow n@p1\}$

cons_ax_PROOF: **Prove** cons_ax **from**
ce $\{\rho \leftarrow s \circ \rho,\ \sigma \leftarrow \Lambda\}$, nilax, $s \circ \rho$,
nat_invariant {nat_var $\leftarrow$ (ρ@c).nextpos}

car_ax_PROOF: **Prove** car_ax **from**
$s \circ \rho$, nat_invariant {nat_var $\leftarrow$ (ρ@c).nextpos}

ρ, σ: **Var** lists_model[$\mathcal{T}$].list

s: **Var** $\mathcal{T}$

n: **Var** nat

cdr_cons_lemma1: **Lemma** length(cdr($s \circ \rho$)) = length(ρ)

cdr_cons_lemma1_proof: **Prove** cdr_cons_lemma1 **from**
$s \circ \rho$, nat_invariant {nat_var $\leftarrow$ (ρ@c).nextpos}

cdr_cons_lemma2: **Lemma** $n <$ length(ρ) $\supset$ (cdr($s \circ \rho$)).value(n) = ρ.value(n)

cdr_cons_lemma2_proof: **Prove** cdr_cons_lemma2 **from**
$s \circ \rho$, nat_invariant {nat_var $\leftarrow$ (ρ@c).nextpos}

cdr_ax_PROOF: **Prove** cdr_ax **from**
ce $\{\sigma \leftarrow \rho,\ \rho \leftarrow$ cdr($s \circ \rho$)$\}$,
cdr_cons_lemma1,
cdr_cons_lemma2 $\{n \leftarrow n@p1\}$

nilorcons_ax_PROOF: **Prove** nilorcons_ax **from**
ce $\{\sigma \leftarrow \Lambda,\ \rho \leftarrow \rho$@c$\}$, ce $\{\sigma \leftarrow$ car(ρ) $\circ$ cdr(ρ)$\}$,
$s \circ \rho$ $\{s \leftarrow$ car(ρ), $\rho \leftarrow$ cdr(ρ)$\}$, nilax,
nat_invariant {nat_var $\leftarrow$ (ρ@c).nextpos}

length_nil_PROOF: **Prove** length_nil **from** nilax

length_cons_PROOF: **Prove** length_cons **from** $s \circ \rho$

End lists_map_proofs

list_inductions: **Module** [t: **Type**]

Using lists[t]

Theory

x: **Var** t

ρ, σ, τ: **Var** list

p: **Var** function[list → bool]

extends: function[list, list → bool] == $(\lambda\, \rho, \sigma : \sigma \neq \Lambda \wedge \rho = \mathrm{cdr}(\sigma))$

list_induction: **Theorem** $p(\Lambda) \wedge (\forall\, \rho, x : p(\rho) \supset p(x \circ \rho)) \supset p(\tau)$

Proof

Using noetherian[list, extends]

cons_induction_proof: **Prove**
list_induction $\{\rho \leftarrow \mathrm{cdr}(d_1@p1),\ x \leftarrow \mathrm{car}(d_1@p1)\}$ **from**
general_induction $\{d \leftarrow \tau,\ d_2 \leftarrow \rho\}$, nilorcons_ax $\{\rho \leftarrow d_1@p1\}$

discharge_well_founded: **Prove** well_founded {measure ← length} **from**
length_cdr $\{\rho \leftarrow b\}$

End list_inductions

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

Assuming

measure: **Var** function[dom → nat]

a, b: **Var** dom

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

Theory

p: **Var** function[dom → bool]

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

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

End noetherian

sets: **Module** [$\mathcal{T}$: **Type**]

Exporting **all**

Theory

set: **Type is** function[$\mathcal{T} \rightarrow$ bool]

x, y, z: **Var** $\mathcal{T}$

a, b: **Var** set

$x \in b$: function[$\mathcal{T}$, set $\rightarrow$ bool] $==$ $(\lambda\, x, b : b(x))$

$a \cup b$: function[set, set $\rightarrow$ set] $==$ $(\lambda\, a, b : (\lambda\, x : x \in a \vee x \in b))$

$a \cap b$: function[set, set $\rightarrow$ set] $==$ $(\lambda\, a, b : (\lambda\, x : x \in a \wedge x \in b))$

$a - b$: function[set, set $\rightarrow$ set] $==$ $(\lambda\, a, b : (\lambda\, x : x \in a \wedge \neg x \in b))$

$\{x\} \cup a$: function[$\mathcal{T}$, set $\rightarrow$ set] $==$ $(\lambda\, x, a : (\lambda\, y : x = y \vee a(y)))$

$\{x\}$: function[$\mathcal{T} \rightarrow$ set] $==$ $(\lambda\, x : (\lambda\, y : y = x))$

$a \subseteq b$: function[set, set $\rightarrow$ bool] $=$ $(\lambda\, a, b : (\forall\, z : z \in a \supset z \in b))$

empty: function[set $\rightarrow$ bool] $=$ $(\lambda\, a : (\forall\, x : \neg x \in a))$

$\emptyset$: set $==$ $(\lambda\, x :$ false$)$

fullset: set $==$ $(\lambda\, x :$ true$)$

extensionality: **Axiom** $(\forall\, x : x \in a = x \in b) \supset (a = b)$

subset_self: **Lemma** $a \subseteq a$

subset_add: **Lemma** $a \subseteq \{x\} \cup a$

Proof

subset_self_proof: **Prove** subset_self **from** $a \subseteq b$ $\{b \leftarrow a\}$

subset_add_proof: **Prove** subset_add **from** $a \subseteq b$ $\{b \leftarrow \{x\} \cup a\}$

End sets

intrans_nonint: **Module**

Using lists, sets

Exporting all with lists, sets

Theory

$\mathcal{S}$: **Type**`(* States *)`

st_0: $\mathcal{S}$`(* Initial State *)`

st, tt, wt: **Var** $\mathcal{S}$

$\mathcal{D}$: **Type**`(* security domains *)`

$\mathcal{A}$: **Type**

dom: function$[\mathcal{A} \rightarrow \mathcal{D}]$

a, b: **Var** $\mathcal{A}$

$\mathcal{A}^\star$: **Type is** list$[\mathcal{A}]$

α: **Var** $\mathcal{A}^\star$

step: function$[\mathcal{S}, \mathcal{A} \rightarrow \mathcal{S}]$

step_count: function$[\mathcal{S}, \mathcal{A}^\star \rightarrow \mathrm{nat}] == (\lambda\, \mathrm{st}, \alpha : \mathrm{length}(\alpha))$

```
run: Recursive function$[\mathcal{S}, \mathcal{A}^\star \rightarrow \mathcal{S}] =$
   $(\lambda\, \mathrm{st}, \alpha :$ if $\mathrm{null}(\alpha)$ then st else $\mathrm{run}(\mathrm{step}(\mathrm{st}, \mathrm{car}(\alpha)), \mathrm{cdr}(\alpha))$ end if)
  by step_count
```

u, v: **Var** $\mathcal{D}$

$u \leadsto v$: function$[\mathcal{D}, \mathcal{D} \rightarrow \mathrm{bool}]$

$u \not\leadsto v$: function$[\mathcal{D}, \mathcal{D} \rightarrow \mathrm{bool}] == (\lambda\, u, v : \neg u \leadsto v)$

step_count2: function$[\mathcal{A}^\star, \mathcal{D} \rightarrow \mathrm{nat}] == (\lambda\, \alpha, u : \mathrm{length}(\alpha))$

src: **Var** set$[\mathcal{D}]$

```
connects: function$[\mathrm{set}, \mathcal{D} \rightarrow \mathrm{bool}] =$
   $(\lambda\, \mathrm{src}, u : (\exists\, v : v \in \mathrm{src} \wedge u \leadsto v))$
```

```
sources: Recursive function$[\mathcal{A}^\star, \mathcal{D} \rightarrow \mathrm{function}[\mathcal{D} \rightarrow \mathrm{bool}]] =$
   $(\lambda\, \alpha, u :$
        if $\mathrm{null}(\alpha)$
          then $\{u\}$
          elsif $\mathrm{connects}(\mathrm{sources}(\mathrm{cdr}(\alpha), u), (\mathrm{dom}(\mathrm{car}(\alpha))))$
             then $\{(\mathrm{dom}(\mathrm{car}(\alpha)))\} \cup \mathrm{sources}(\mathrm{cdr}(\alpha), u)$
             else $\mathrm{sources}(\mathrm{cdr}(\alpha), u)$
          end if)
  by step_count2
```

```
ipurge: Recursive function[$\mathcal{A}^\star, \mathcal{D} \rightarrow \mathcal{A}^\star$] =
   ( $\lambda\, \alpha, u$ :
         if null($\alpha$)
           then $\Lambda$
           elsif (dom(car($\alpha$))) $\in$ sources($\alpha, u$)
              then car($\alpha$) $\circ$ ipurge(cdr($\alpha$), $u$)
              else ipurge(cdr($\alpha$), $u$)
           end if)
  by step_count2

$\mathcal{O}$ : Type (* outputs *)

output: function[$\mathcal{S}, \mathcal{A} \rightarrow \mathcal{O}$]

do: function[$\mathcal{A}^\star \rightarrow \mathcal{S}$] = ( $\lambda\, \alpha$ : run($st_0, \alpha$))

test: function[$\mathcal{A}^\star, \mathcal{A} \rightarrow \mathcal{O}$] = ( $\lambda\, \alpha, a$ : output(do($\alpha$), $a$))

secure: function[$\mathcal{A}^\star \rightarrow$ bool] =
   ( $\lambda\, \alpha$ : ( $\forall\, a$ : test($\alpha, a$) = test(ipurge($\alpha$, dom($a$)), $a$)))

V: Type

view: function[$\mathcal{D}, \mathcal{S} \rightarrow V$]

view_id: function[$\mathcal{D}, \mathcal{S}, \mathcal{S} \rightarrow$ bool] ==
   ( $\lambda\, u$, st, tt : view($u$, st) = view($u$, tt))

output_consistent: bool =
   ( $\forall\, a$, st, tt :
         view_id(dom($a$), st, tt) $\supset$ output(st, $a$) = output(tt, $a$))

view_consistent: function[$\mathcal{A}^\star \rightarrow$ bool] =
   ( $\lambda\, \alpha$ : ( $\forall\, u$ : view_id($u$, do($\alpha$), do(ipurge($\alpha, u$)))))

local_respect: bool =
   ( $\forall\, v$, st, $a$ : dom($a$) $\not\rightsquigarrow v \supset$ view_id($v$, st, step(st, $a$)))

weak_step_consistent: bool =
   ( $\forall\, u$, st, tt, $a$ :
         view_id($u$, st, tt) $\wedge$ view_id(dom($a$), st, tt)
            $\supset$ view_id($u$, step(st, $a$), step(tt, $a$)))

lemma2: Lemma view_consistent($\alpha$) $\wedge$ output_consistent $\supset$ secure($\alpha$)

unwinding: Theorem
  local_respect $\wedge$ weak_step_consistent $\wedge$ output_consistent $\supset$ secure($\alpha$)
```

Proof

Using list_inductions

lemma2_proof: **Prove** lemma2 **from**
  secure,
  view_consistent $\{u \leftarrow \mathrm{dom}(a@p1)\}$,
  output_consistent
    $\{a \leftarrow a@p1$,
     $\mathrm{st} \leftarrow \mathrm{do}(\alpha)$,
     $\mathrm{tt} \leftarrow \mathrm{do}(\mathrm{ipurge}(\alpha, u@p2f))\}$,
  test $\{a \leftarrow a@p1\}$,
  test $\{a \leftarrow a@p1,\ \alpha \leftarrow \mathrm{ipurge}(\alpha, u@p2f)\}$

single_step_lemma: **Lemma** $\mathrm{run}(\mathrm{st}, a \circ \alpha) = \mathrm{run}(\mathrm{step}(\mathrm{st}, a), \alpha)$

single_step_lemma_proof: **Prove** single_step_lemma **from**
  run $\{\alpha \leftarrow a \circ \alpha\}$,
  cons_ax[$\mathcal{A}$] $\{\rho \leftarrow \alpha,\ s \leftarrow a\}$,
  car_ax[$\mathcal{A}$] $\{\rho \leftarrow \alpha,\ s \leftarrow a\}$,
  cdr_ax[$\mathcal{A}$] $\{\rho \leftarrow \alpha,\ s \leftarrow a\}$

purge_lemma: **Lemma**
  $\mathrm{ipurge}(a \circ \alpha, u)$
    = **if** $\mathrm{dom}(a) \in \mathrm{sources}(a \circ \alpha, u)$
      then $a \circ \mathrm{ipurge}(\alpha, u)$
      else $\mathrm{ipurge}(\alpha, u)$
      end if

purge_lemma_proof: **Prove** purge_lemma **from**
  ipurge $\{\alpha \leftarrow a \circ \alpha\}$,
  cons_ax[$\mathcal{A}$] $\{\rho \leftarrow \alpha,\ s \leftarrow a\}$,
  car_ax[$\mathcal{A}$] $\{\rho \leftarrow \alpha,\ s \leftarrow a\}$,
  cdr_ax[$\mathcal{A}$] $\{\rho \leftarrow \alpha,\ s \leftarrow a\}$

sources_subset: **Lemma** $\mathrm{sources}(\alpha, u) \subseteq \mathrm{sources}(a \circ \alpha, u)$

sources_subset_proof: **Prove** sources_subset **from**
  sources $\{\alpha \leftarrow a \circ \alpha\}$,
  cons_ax[$\mathcal{A}$] $\{s \leftarrow a,\ \rho \leftarrow \alpha\}$,
  cdr_ax[$\mathcal{A}$] $\{s \leftarrow a,\ \rho \leftarrow \alpha\}$,
  car_ax[$\mathcal{A}$] $\{s \leftarrow a,\ \rho \leftarrow \alpha\}$,
  subset_self[$\mathcal{D}$] $\{a \leftarrow \mathrm{sources}(\alpha, u)\}$,
  subset_add[$\mathcal{D}$] $\{a \leftarrow \mathrm{sources}(\alpha, u),\ x \leftarrow \mathrm{dom}(a)\}$

sources_grows: **Lemma** $v \in \mathrm{sources}(\alpha, u) \supset v \in \mathrm{sources}(a \circ \alpha, u)$

sources_grows_proof: **Prove** sources_grows **from**
  sources_subset,
  $a \subseteq b$ [$\mathcal{D}$] $\{a \leftarrow \mathrm{sources}(\alpha, u),\ b \leftarrow \mathrm{sources}(a \circ \alpha, u),\ z \leftarrow v\}$

sources_defn_base_case: **Lemma** $\mathrm{sources}(\Lambda, u) = \{u\}$

sources_defn_base_case_proof: **Prove** sources_defn_base_case **from**
  sources $\{\alpha \leftarrow \Lambda\}$

sources_defn_inductive_case: **Lemma**
$(\exists\, v : v \in \text{sources}(\alpha, u) \wedge \text{dom}(a) \rightsquigarrow v)$
$\supset \text{sources}(a \circ \alpha, u) = \{\text{dom}(a)\} \cup \text{sources}(\alpha, u)$

sources_defn_inductive_case_proof: **Prove** sources_defn_inductive_case
from sources $\{\alpha \leftarrow a \circ \alpha\}$,
connects $\{\text{src} \leftarrow \text{sources}(\alpha, u),\ u \leftarrow \text{dom}(a)\}$,
cons_ax[$\mathcal{A}$] $\{s \leftarrow a,\ \rho \leftarrow \alpha\}$,
cdr_ax[$\mathcal{A}$] $\{s \leftarrow a,\ \rho \leftarrow \alpha\}$,
car_ax[$\mathcal{A}$] $\{s \leftarrow a,\ \rho \leftarrow \alpha\}$

in_own_sources: **Lemma** $u \in \text{sources}(\alpha, u)$

in_own_sources_pred: function[$\mathcal{A}^\star \rightarrow$ bool] =
$(\lambda\, \alpha : (\forall\, u : u \in \text{sources}(\alpha, u)))$

in_own_sources_form: **Lemma** in_own_sources_pred(α)

in_own_sources_basis: **Lemma** in_own_sources_pred(Λ)

in_own_sources_basis_proof: **Prove** in_own_sources_basis **from**
in_own_sources_pred $\{\alpha \leftarrow \Lambda\}$, sources_defn_base_case $\{u \leftarrow u@p1\}$

in_own_sources_induct: **Lemma**
in_own_sources_pred(α) $\supset$ in_own_sources_pred($a \circ \alpha$)

in_own_sources_induct_proof: **Prove** in_own_sources_induct **from**
in_own_sources_pred $\{u \leftarrow u@p2\}$,
in_own_sources_pred $\{\alpha \leftarrow a \circ \alpha\}$,
sources_grows $\{u \leftarrow u@p2,\ v \leftarrow u@p2\}$

in_own_sources_form_proof: **Prove** in_own_sources_form **from**
list_induction $\{\tau \leftarrow \alpha,\ p \leftarrow$ in_own_sources_pred$\}$,
in_own_sources_basis,
in_own_sources_induct $\{a \leftarrow x@p1,\ \alpha \leftarrow \rho@\text{p}1\}$

in_own_sources_proof: **Prove** in_own_sources **from**
in_own_sources_form, in_own_sources_pred

C: **Var** set[$\mathcal{D}$]

strong_view_id: function[set, $\mathcal{S}, \mathcal{S} \rightarrow$ bool] =
$(\lambda\, C, \text{st}, \text{tt} : (\forall\, v : v \in C \supset \text{view}(v, \text{st}) = \text{view}(v, \text{tt})))$

strong_view_id_reflexive: **Lemma** strong_view_id(C, st, st)

strong_view_id_reflexive_proof: **Prove** strong_view_id_reflexive **from**
strong_view_id $\{\text{tt} \leftarrow \text{st}\}$

strong_view_id_symmetric: **Lemma**
strong_view_id(C, st, tt) $\supset$ strong_view_id(C, tt, st)

strong_view_id_symmetric_proof: **Prove** strong_view_id_symmetric **from**
strong_view_id $\{v \leftarrow v@p2\}$, strong_view_id $\{\text{st} \leftarrow \text{tt},\ \text{tt} \leftarrow \text{st}\}$

strong_view_id_transitive: **Lemma**
strong_view_id(C, st, tt) $\wedge$ strong_view_id(C, tt, wt)
$\supset$ strong_view_id(C, st, wt)

strong_view_id_transitive_proof: **Prove** strong_view_id_transitive **from**
strong_view_id $\{v \leftarrow v@p3\}$,
strong_view_id $\{v \leftarrow v@p3,\ \mathrm{st} \leftarrow \mathrm{tt},\ \mathrm{tt} \leftarrow \mathrm{wt}\}$,
strong_view_id $\{\mathrm{tt} \leftarrow \mathrm{wt}\}$

strong_view_id_sources_prop: **Lemma**
strong_view_id(sources($a \circ \alpha, u$), st, tt)
$\supset$ strong_view_id(sources(α, u), st, tt)

strong_view_id_sources_prop_proof: **Prove** strong_view_id_sources_prop
from strong_view_id $\{C \leftarrow \mathrm{sources}(\alpha, u)\}$,
strong_view_id $\{C \leftarrow \mathrm{sources}(a \circ \alpha, u),\ v \leftarrow v@p1\}$,
sources_grows $\{v \leftarrow v@p1\}$

lemma3: **Lemma**
weak_step_consistent
$\wedge$ local_respect $\wedge$ strong_view_id(sources($a \circ \alpha, u$), st, tt)
$\supset$ strong_view_id(sources(α, u), step(st, a), step(tt, a))

lemma3_proof: **Prove** lemma3 **from**
strong_view_id $\{v \leftarrow v@p2,\ C \leftarrow \mathrm{sources}(a \circ \alpha, u)\}$,
strong_view_id
$\{C \leftarrow \mathrm{sources}(\alpha, u)$,
$\mathrm{st} \leftarrow \mathrm{step}(\mathrm{st}, a)$,
$\mathrm{tt} \leftarrow \mathrm{step}(\mathrm{tt}, a)\}$,
sources_grows $\{v \leftarrow v@p2\}$,
local_respect $\{v \leftarrow v@p2\}$,
local_respect $\{v \leftarrow v@p2,\ \mathrm{st} \leftarrow \mathrm{tt}\}$,
weak_step_consistent $\{u \leftarrow v@p2\}$,
strong_view_id $\{v \leftarrow \mathrm{dom}(a),\ C \leftarrow \mathrm{sources}(a \circ \alpha, u)\}$,
sources_defn_inductive_case $\{v \leftarrow v@p2\}$

lemma4: **Lemma**
local_respect $\wedge\ \neg$dom(a) $\in$ sources($a \circ \alpha, u$)
$\supset$ strong_view_id(sources(α, u), st, step(st, a))

lemma4_proof: **Prove** lemma4 **from**
local_respect $\{v \leftarrow v@p2\}$,
strong_view_id $\{C \leftarrow \mathrm{sources}(\alpha, u),\ \mathrm{tt} \leftarrow \mathrm{step}(\mathrm{st}, a)\}$,
sources_defn_inductive_case $\{v \leftarrow v@p2\}$

lemma5: **Lemma**
weak_step_consistent
$\wedge$ local_respect $\wedge$ strong_view_id(sources(α, u), st, tt)
$\supset$ view_id(u, run(st, α), run(tt, ipurge(α, u)))

lemma5_pred: function[$\mathcal{A}^{\star} \rightarrow$ bool] =
($\lambda\, \alpha$: ($\forall\, u$, st, tt :
weak_step_consistent
$\wedge$ local_respect $\wedge$ strong_view_id(sources(α, u), st, tt)
$\supset$ view_id(u, run(st, α), run(tt, ipurge(α, u)))))

lemma5_basis: **Lemma** lemma5_pred(Λ)

lemma5_basis_proof: **Prove** lemma5_basis **from**
lemma5_pred $\{\alpha \leftarrow \Lambda\}$,
strong_view_id
$\{v \leftarrow u@p1$,
$C \leftarrow$ sources($\Lambda, u@p1$),
st $\leftarrow$ st@$p1$,
tt $\leftarrow$ tt@$p1\}$,
ipurge $\{\alpha \leftarrow \Lambda,\ u \leftarrow u@p1\}$,
run $\{\alpha \leftarrow \Lambda$, st $\leftarrow$ st@$p1\}$,
run $\{\alpha \leftarrow \Lambda$, st $\leftarrow$ tt@$p1\}$,
in_own_sources $\{\alpha \leftarrow \Lambda,\ u \leftarrow u@p1\}$

lemma5_induct: **Lemma** lemma5_pred(α) $\supset$ lemma5_pred($a \circ \alpha$)

lemma5_induct_proof: **Prove** lemma5_induct **from**
lemma5_pred $\{\alpha \leftarrow a \circ \alpha\}$,
lemma5_pred
$\{u \leftarrow u@p1$,
st $\leftarrow$ step(st@$p1, a$),
tt $\leftarrow$ step(tt@$p1, a$)$\}$,
lemma5_pred $\{u \leftarrow u@p1$, st $\leftarrow$ step(st@$p1, a$), tt $\leftarrow$ tt@$p1\}$,
lemma3 $\{u \leftarrow u@p1$, st $\leftarrow$ st@$p1$, tt $\leftarrow$ tt@$p1\}$,
single_step_lemma $\{$st $\leftarrow$ st@$p1\}$,
single_step_lemma $\{\alpha \leftarrow$ ipurge($\alpha, u@p1$), st $\leftarrow$ tt@$p1\}$,
purge_lemma $\{u \leftarrow u@p1\}$,
lemma4 $\{u \leftarrow u@p1$, st $\leftarrow$ st@$p1\}$,
strong_view_id_sources_prop
$\{u \leftarrow u@p1$,
st $\leftarrow$ st@$p1$,
tt $\leftarrow$ tt@$p1\}$,
strong_view_id_symmetric
$\{C \leftarrow$ sources($\alpha, u@p1$),
st $\leftarrow$ st@$p1$,
tt $\leftarrow$ step(st@$p1, a$)$\}$,
strong_view_id_transitive
$\{C \leftarrow$ sources($\alpha, u@p1$),
st $\leftarrow$ step(st@$p1, a$),
tt $\leftarrow$ st@$p1$,
wt $\leftarrow$ tt@$p1\}$

lemma5_proof: **Prove** lemma5 **from**
list_induction $\{\tau \leftarrow \alpha,\ p \leftarrow \text{lemma5\_pred}\}$,
lemma5_basis,
lemma5_induct $\{a \leftarrow x@p1,\ \alpha \leftarrow \rho@\text{p1}\}$,
lemma5_pred

unwinding_proof: **Prove** unwinding **from**
view_consistent,
lemma2,
lemma5 $\{u \leftarrow u@p1,\ \text{st} \leftarrow st_0,\ \text{tt} \leftarrow st_0\}$,
do,
do $\{\alpha \leftarrow \text{ipurge}(\alpha, u@p1)\}$,
strong_view_id_reflexive $\{C \leftarrow \text{sources}(\alpha, u@p1),\ \text{st} \leftarrow st_0\}$

End intrans_nonint

intrans_nonint_tcc: **Module**

Using intrans_nonint

Exporting all with intrans_nonint

Theory

α: **Var** lists[$\mathcal{A}$].list

tt: **Var** $\mathcal{S}$

st: **Var** $\mathcal{S}$

v: **Var** $\mathcal{D}$

ρ: **Var** lists[$\mathcal{A}$].list

x: **Var** $\mathcal{A}$

u: **Var** $\mathcal{D}$

a: **Var** $\mathcal{A}$

`(* Termination TCC generated for run *)`

run_TCC1: **Formula**
$(\neg(\text{null}(\alpha))) \supset \text{step\_count}(\text{st}, \alpha) > \text{step\_count}(\text{step}(\text{st}, \text{car}(\alpha)), \text{cdr}(\alpha))$

`(* Termination TCC generated for sources *)`

sources_TCC1: **Formula**
$(\neg(\text{null}(\alpha))) \supset \text{step\_count2}(\alpha, u) > \text{step\_count2}(\text{cdr}(\alpha), u)$

`(* Termination TCC generated for sources *)`

sources_TCC2: **Formula**
$(\text{connects}(\text{sources}(\text{cdr}(\alpha), u), (\text{dom}(\text{car}(\alpha))))) \wedge (\neg(\text{null}(\alpha)))$
$\supset \text{step\_count2}(\alpha, u) > \text{step\_count2}(\text{cdr}(\alpha), u)$

`(* Termination TCC generated for sources *)`

sources_TCC3: **Formula**
$(\neg(\text{connects}(\text{sources}(\text{cdr}(\alpha), u), (\text{dom}(\text{car}(\alpha)))))) \wedge (\neg(\text{null}(\alpha)))$
$\supset \text{step\_count2}(\alpha, u) > \text{step\_count2}(\text{cdr}(\alpha), u)$

`(* Termination TCC generated for ipurge *)`

ipurge_TCC1: **Formula**
$((\text{dom}(\text{car}(\alpha))) \in \text{sources}(\alpha, u)) \wedge (\neg(\text{null}(\alpha)))$
$\supset \text{step\_count2}(\alpha, u) > \text{step\_count2}(\text{cdr}(\alpha), u)$

`(* Termination TCC generated for ipurge *)`

ipurge_TCC2: **Formula**
$(\neg((\text{dom}(\text{car}(\alpha))) \in \text{sources}(\alpha, u))) \wedge (\neg(\text{null}(\alpha)))$
$\supset \text{step\_count2}(\alpha, u) > \text{step\_count2}(\text{cdr}(\alpha), u)$

Proof

run_TCC1_PROOF: **Prove** run_TCC1

sources_TCC1_PROOF: **Prove** sources_TCC1

sources_TCC2_PROOF: **Prove** sources_TCC2

sources_TCC3_PROOF: **Prove** sources_TCC3

ipurge_TCC1_PROOF: **Prove** ipurge_TCC1

ipurge_TCC2_PROOF: **Prove** ipurge_TCC2

End intrans_nonint_tcc

intrans_nonint_tcc_proofs: **Module**

Using intrans_nonint, intrans_nonint_tcc

Proof

run_TCC1_PROOF: **Prove** run_TCC1 **from** length_cdr[$\mathcal{A}$] $\{\rho \leftarrow \alpha\}$

sources_TCC1_PROOF: **Prove** sources_TCC1 **from** length_cdr[$\mathcal{A}$] $\{\rho \leftarrow \alpha\}$

sources_TCC2_PROOF: **Prove** sources_TCC2 **from** length_cdr[$\mathcal{A}$] $\{\rho \leftarrow \alpha\}$

sources_TCC3_PROOF: **Prove** sources_TCC3 **from** length_cdr[$\mathcal{A}$] $\{\rho \leftarrow \alpha\}$

purge_TCC1_PROOF: **Prove** ipurge_TCC1 **from** length_cdr[$\mathcal{A}$] $\{\rho \leftarrow \alpha\}$

purge_TCC2_PROOF: **Prove** ipurge_TCC2 **from** length_cdr[$\mathcal{A}$] $\{\rho \leftarrow \alpha\}$

End intrans_nonint_tcc_proofs

top: **Module** [$\mathcal{T}$: **Type**]

Using lists_model[$\mathcal{T}$], lists_model_tcc[$\mathcal{T}$], lists_map[$\mathcal{T}$],
lists_map_proofs[$\mathcal{T}$], intrans_nonint_tcc_proofs

Proof

TCC_proof: **Prove** car_TCC1 **from**
nat_invariant {nat_var ← (ρ@c).nextpos}

End top