Proving Noninterference and Functional Correctness Using Traces

John McLean

Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, D. C. 20375

The trace method of software specification is extended to provide a natural semantics for a procedural programming language. This extension provides a method for proving program correctness that permits a direct proof of program Noninterference without having to produce an intermediate finite state machine and unwinding conditions. This approach provides a uniform framework for reasoning about abstract software system specifications and their implementations. It also allows us to prove security at an abstract level so that changes to programs that do not affect functional behavior will not affect the security proof.

1. Introduction

Noninterference, the requirement that high-level users of a system cannot affect what low-level users see [8], is regarded by many as the best specification of security available for deterministic systems in which high-level output may not be generated from low-level input [27]. One reason for this high regard is that it abstracts away from implementation mechanisms and focuses on the user visible input-output relation, thus eliminating implementation clutter that makes it hard to glean necessary program behavior from implementation-dependent artifact. Besides making specifications more difficult to reason about, such clutter unduly constrains implementors and introduces unnecessary module coupling that makes software hard to understand and hard to change [12, 22, 28].

Unfortunately, the abstractness Noninterference enjoys may not be preserved by the traditional approach to proving that a programming system is noninterfering. Although Noninterference is most naturally formulated as a trace specification,[1] the traditional approach to proving a system noninterfering consists of constructing a finite state machine model of system operation, showing that the constructed machine satisfies a set of unwinding conditions that are sufficient for establishing Noninterference, and mapping the state machine onto the software.[2] Since the state machine constructed tends to embody, not an abstract specification of system behavior, but a concrete mechanism for implementing that behavior, the benefits of abstraction may be lost.[3] [22]

This is not to say that state machines have no place in the software development process. Often they are convenient and sometimes extremely useful, but usually not as

1. For example, see [19].
2. This approach was first proposed in [9] and has been universally followed since then [7].
3. As Lamport has argued in [18], abstraction can be maintained when state machines are used carefully. However, state machines necessitate the local storage of event history that is relevant to a state's behavior. This often tempts specifiers to optimize this storage and thereby, introduce implementation detail into the specification.

specifications, but rather as models, i. e., artificial constructs that satisfy a specification. Model building is most useful in constructing software if, as occasionally happens in the security arena as well as elsewhere, an abstract specification gives a programmer too little guidance on how actually to build a system that satisfies the specification [26]. However, in many cases model building is an unnecessary step. When it is relatively clear how to implement an abstract specification (as it now often is in security), model construction simply introduces unnecessary effort into the software construction process. This extra work appears not only during system construction, but throughout the system's life-cycle. Since a model supports a particular implementation, whenever we change implementations we must change the model accordingly and redo any proof that was formulated on the model-level of abstraction.

In this paper we advocate showing that an abstract functional specification satisfies Noninterference directly and then showing that the program satisfies the functional specification. By being carried out on as abstract a level as possible, our security proof can survive implementation changes that do not affect the user interface to the system.[4] We then show how to prove that a program satisfies a trace specification of Noninterference directly, without having to introduce an intermediary state machine. We go beyond this rather local concern, however, by showing how to define a programming language in terms of traces and how to prove a program correct vis-a-vis a trace specification. As such, this paper should also be of interest to those who view trace specifications as a promising specification method, but who do not see how to apply trace specifications to program verification [1, 6]. The resulting method provides a uniform framework for reasoning about specifications and programs.

We begin with an introduction to trace specifications, followed by a demonstration of how to prove that a specification satisfies Noninterference. We then show how to define a programming language using traces and use this definition to prove a program correct and, a fortiori, secure.

2. Trace Specification

This section gives an informal presentation of the trace method for the abstract specification of software [2, 22]. Those interested in a more formal treatment of traces are referred to [22]. Those interested in studying the use of traces to specify real systems should consult [5, 15, 17, 29]. A software system for analyzing trace specifications is described in [25].

A trace specification for a module consists of two parts: (1) a syntax section lists the procedure names and types the module comprises, and (2) a semantics section specifies the external behavior that the module's procedures must exhibit. This behavior is specified by assertions that describe the behavior of sequences of procedure calls, written $call_1.call_2.\cdots.call_n$, called *traces*. The assertions are based on first order logic, supplemented by the predicate L, which is true when applied to a *legal* trace (a trace that is acceptable to the module), and the function V, which gives a return value when applied to a legal trace ending in a function call. The null trace, denoted by [], is always legal and is such that for any trace T, $T=[].T=T.[]$. Two traces T and S are *equivalent*,

4. Similar remarks apply, e. g., to safety.

written $T \equiv S$, if and only if for any trace R, $L(T.R)$ iff $L(S.R)$ and for non-null R, $V(T.R) = V(S.R)$, if defined. That is, if two traces are equivalent, then they are indistinguishable as far as L and V are concerned with respect to future program behavior. Formally,

$$
\begin{aligned}
&T \equiv S \leftrightarrow \\
&\quad (R)((L(T.R) \leftrightarrow L(S.R)) \wedge \\
&\qquad (R \neq [\,] \rightarrow ((\exists x)V(T.R)=x \rightarrow V(T.R)=V(S.R))))^{5}
\end{aligned}
$$

Note that if $T \equiv S$, then $T.R \equiv S.R$ (since, for example, $T \equiv S \rightarrow (L(T.R.W) \leftrightarrow L(S.R.W))$), but it is not necessarily the case that $R.T \equiv R.S$. As an example of how to use traces, consider the following specification for a stack:

SIMPLE STACK SPECIFICATION

Syntax:
  PUSH: integer
  TOP: $\rightarrow$ integer
  POP:

Semantics:
  (1) L(T) $\rightarrow$ T≡T.PUSH(i).POP
  (2) L(T.TOP) $\leftrightarrow$ L(T.POP)
  (3) L(T.TOP) $\rightarrow$ T.TOP≡T
  (4) L(T.PUSH(i)) $\rightarrow$ V(T.PUSH(i).TOP)=i

The syntax section says that the procedure *PUSH* takes an integer as a parameter, that *TOP* returns an integer, and that *POP* neither takes nor returns a parameter. The first assertion in the semantics section says that if T is a legal sequence of procedure calls, then T is equivalent to the sequence of calls consisting of T followed by a call to *PUSH* and a call to *POP*. It follows from this that if T is legal, then so is $T.PUSH(i)$ and $T.PUSH(i).POP$. The second assertion says that $T.TOP$ is a legal sequence of procedure calls if and only if $T.POP$ is a legal sequence of procedure calls. The third assertion says that if $T.TOP$ is a legal sequence of procedure calls, then it is equivalent to T. The fourth assertion says that *TOP* returns the last value pushed on the stack that has not previously been popped. Note that given semantics for L and V (and hence, derivatively, for ≡), *PUSH*, *POP*, and *TOP* are completely and unambiguously specified, yet their specification is consistent with any correct implementation, whether we chose to implement the stack as an array, linked list, etc.

It may seem that the specification would be satisfied by a model in which no trace is legal. Such a model is ruled out by an assumption that the empty trace is always legal. This and other assumptions of the method are presented in [22], where a model-theoretic semantics specifies the semantic consequences of a specification, a derivation system specifies the proof-theoretic consequences of a specification, and completeness and soundness theorems show that an assertion is a semantic consequence of a specification if and only if it is derivable from the specification. This foundation supports coextensive

5. This is a simpler definition than given in [22], but the two definitions can easily be shown to be equivalent.

semantic and syntactic definitions of *totalness* and *consistency*, and formal techniques for proving these properties for a specification. For our purposes here, the most important elements of this foundation are the following axioms from the derivation system.

TRACE DEDUCTIVE SYSTEM AXIOMS

(1) $L([])$
(2) $L(T.S) \rightarrow L(T)$
(3) $V(T)=x \rightarrow L(T)$

The first axiom states that the empty trace is always legal, and the second that any initial segment of a legal trace is legal. The third axiom states that only legal traces can return a value. The first and second appear verbatim in [22]. The third is a consequence of a stronger axiom that appears in [22].

3. Proving Specification Noninterference

To focus our discussion of Noninterference, we shall consider a multi-level stack, that is a module where users of various security levels can access stacks of various security levels. To prevent high-level users from passing information to low-level users, a user will be forbidden to read stacks whose classification exceeds his clearance and a user will be forbidden to push or pop stacks whose classifications are exceeded by her clearance. In the following specification we assume the existence of a linearly ordered set of security levels, which includes a least level, denoted by $\perp$, and a function *cl* such that *cl*(*user*) is the security level (clearance) of the user and *cl*(*stack*) is the security level (classification) of the stack.

MULTI-LEVEL STACK SPECIFICATION

Syntax:
PUSH: (user, stack, data)
TOP: (user, stack) $\rightarrow$ data
POP: (user, stack)

Semantics:

(1) $(cl(u)>\perp \lor cl(s)>\perp) \rightarrow (L(T.S) \leftrightarrow L(T.PUSH(u,s,x).S))$
(2) $L(T.POP(u,s)) \leftrightarrow L(T.TOP(u,s))$
(3) $(cl(u)>\perp \lor cl(s)>\perp) \rightarrow (L(T.S) \leftrightarrow L(T.POP(u,s).S))$

(4) $cl(u)>cl(s) \rightarrow T.PUSH(u,s,x) \equiv T$
(5) $cl(u)>cl(s) \rightarrow T.POP(u,s) \equiv T$
(6) $(L(T) \land cl(s) \geq cl(u) \land cl(s) \geq cl(w)) \rightarrow T.PUSH(u,s,x).POP(w,s) \equiv T$
(7) $r \neq s \rightarrow T.PUSH(w,r,x).POP(u,s) \equiv T.POP(u,s).PUSH(w,r,x)$
(8) $L(POP(u,s)) \rightarrow POP(u,s) \equiv []$
(9) $L(T.TOP(u,s)) \rightarrow T.TOP(u,s) \equiv T$

(10) $(L(T) \land cl(s)>cl(u)) \rightarrow V(T.TOP(u,s))=$*violation*
(11) $(L(TOP(u,s)) \land cl(u) \geq cl(s)) \rightarrow V(TOP(u,s))=$*empty*
(12) $(L(T) \land cl(s) \geq cl(u) \land cl(w) \geq cl(s)) \rightarrow$
$(V(T.PUSH(u,s,x).TOP(w,s))=x)$
(13) $(L(T.TOP(u,s)) \land r \neq s) \rightarrow$

V(T.PUSH(w,r,x).TOP(u,s))=V(T.TOP(u,s))

This specification differs from the stack specification given in the previous section by containing explicit mention of users as parameters to procedure calls and multiple stacks. A more important difference is that it requires that every program call executed by a user a user whose security level is greater than $\perp$, be legal. This is to avoid a channel from high-level users to low-level users in which information is passed by high-level users systematically crashing the system. Further, it requires that any procedure call by a bottom-level user (i. e., a user whose security level is $\perp$) to a higher-level stack is legal since the return value such a call elicits must be completely specified to prevent, e. g., a channel where a high-level user can pass information by flushing or not flushing a high-level stack which a low-level user tries to access. It is not the case that every call a bottom-level user makes to a bottom-level stack must be legal, and this gives the programmer flexibility when implementing, e. g., a POP by a bottom-level user to an empty bottom-level stack. However, we shall see that this looseness does not permit nonsecure implementations.

Noninterference is the property that under no conditions shall a user's output be affected by the input of a user who has a higher clearance. To define this property formally we shall first define a purge function, *PURGE* (*user* ,*trace*), which when applied to a user and a trace, removes all procedure calls from that trace that are executed by a user whose clearance exceeds that of the given user. Noninterference can then be defined as the property that the output to a user given a trace is always identical to the output the user would see given the purged trace.

PURGE(user, trace)

PURGE(u,[])=[]

cl(w)>cl(u) $\rightarrow$ PURGE(u,T.PUSH(w,s,x))=PURGE(u,T)
cl(w)>cl(u) $\rightarrow$ PURGE(u,T.POP(w,s))=PURGE(u,T)
cl(w)>cl(u) $\rightarrow$ PURGE(u,T.TOP(w,s))=PURGE(u,T)

cl(u)≥cl(w) $\rightarrow$ PURGE(u,T.PUSH(w,s,x))=PURGE(u,T).PUSH(w,s,x)
cl(u)≥cl(w) $\rightarrow$ PURGE(u,T.POP(w,s))=PURGE(u,T).POP(w,s)
cl(u)≥cl(w) $\rightarrow$ PURGE(u,T.TOP(w,s))=PURGE(u,T).TOP(w,s)

NONINT $\leftrightarrow$ (u)(s)((L(Θ.TOP(u,s)) $\leftrightarrow$ L(PURGE(u,Θ).TOP(u,s)))
$\wedge$ (L(Θ.TOP(u,s)) $\rightarrow$ V(Θ.TOP(u,s))=V(PURGE(u,Θ).TOP(u,s)))), for any trace constant, Θ, where a trace constant is a trace expression that contains no variables [22].

The Noninterference property is actually an axiom schema that says for every variable-free trace Θ, (1) Θ.TOP(u,s) returns a value if and only if PURGE(u,Θ).TOP(u,s) returns a value, and (2) if Θ.TOP(u,s) and PURGE(u,Θ).TOP(u,s) return values, then they return the same value. We chose to state it this way, rather than to quantify over all traces, since stating it as a universal quantification would require that infinitely long traces, which may exist in some models of our axioms, must exhibit Noninterference as well. Such a requirement is not necessary and would entail the introduction of an induction schema into our trace language. Although the proof of Noninterference requires induction, the induction will be performed in the metalanguage, which, for technical reasons, is quite a different matter.

As stated in the Introduction, the traditional method for proving that our specification satisfies Noninterference would require developing a state machine model of the specification and unwinding conditions for the state machine sufficient to show that the machine satisfies Noninterference. We now show how to bypass the additional effort of constructing such a machine and to prove directly that the specification satisfies Noninterference.

Theorem: NONINT

Proof: Our theorem follows from four lemmas, whose proofs we shall sketch. The first lemma shows that a trace is legal if and only if its purge is legal. The second lemma shows that every legal trace has a normal form. The third shows that if our theorem holds for the normal form of a trace, then it holds for the trace. The fourth lemma shows that our theorem holds for all normal form traces. It follows that our theorem holds for all traces. It should be noted that our arguments occur on the semantic level. The fact that the trace derivation system is sound and complete allows us to draw conclusions about what is and is not derivable from properties of models and vice versa.

Lemma 1: For any constant trace ϕ, $L(\phi)$ is true in a model if and only if $L(PURGE(u,\phi))$ is true in the model.

Proof Sketch of Lemma 1: Proof is by strong induction on trace length, where we assume the lemma for traces consisting of fewer than n procedure calls and prove the lemma for traces consisting of n calls. If $n=0$ then ϕ is the empty trace and the lemma is trivially true since $PURGE(u,\phi)=\phi$. If $n>0$, then ϕ is of the form $\psi.proc$ for some procedure call $proc$. If $L(\psi)$ is not true in some model M, then neither is $PURGE(u,\psi)$, by the induction hypothesis, and neither is ϕ nor $PURGE(u,\phi)$ by axiom (2) of the trace deductive system. If $L(\psi)$ is true in some model M, then so is $L(PURGE(u,\psi))$ by the induction hypothesis. If $L(PURGE(u,\phi))$ is true then $L(\phi)$ is true since specification axiom (1) allows us legally to insert anywhere in a trace PUSH's by users whose level exceeds u (since cl(u)≥⊥)), axiom (3) allows us legally to insert POP's by users whose level exceeds u, and axioms (2) and(9) allows us legally to convert any POP to a TOP. Hence, we can legally construct ϕ from $PURGE(u,\phi)$. If $L(\phi)$ is true, then by axiom (1) we can legally remove all PUSH's made by user's whose security level exceed's $cl(u)$, by axiom (3) we can legally remove all POP's by user's whose level exceeds $cl(u)$, and by axiom (9) we can legally remove all TOP's by user's whose level exceeds $cl(u)$. The result is $PURGE(u,\phi)$, and we are done.

Lemma 2: For every constant trace ϕ that is legal in some model there is a trace ψ, which we shall call the *normal form* of ϕ, such that (1) $\phi \equiv \psi$ in all models in which ϕ is legal, and (2) ψ is a subsequence of ϕ consisting entirely of secure PUSH's, i. e., push's of the form PUSH(u,s,x) where cl(s)≥cl(u).

Proof Sketch of Lemma 2: Use axioms (4) and (5) to remove nonsecure PUSH's and POP's. Since the trace is legal we can use axiom (9) to remove all TOP's, and axioms (6), (7), and (8) to remove any remaining POP's starting with the leftmost first. Since the trace is finite, this procedure will terminate, and by we are done.

Lemma 3: If V(PURGE(u,ϕ).TOP(u,s))=V(ϕ.TOP(u,s)) is true in some model where ϕ is the normal form of constant trace θ, then V(PURGE(u,θ).TOP(u,s))=V(θ.TOP(u,s)) is true in all models in which $\theta.TOP(u,s)$ is legal.

Proof Sketch of Lemma 3: If $\theta.TOP(u,s)$ (and, hence, θ) is legal, then V(θ.TOP(u,s))=V(ϕ.TOP(u,s)) since $\theta \equiv \phi$ by Lemma 2. Hence, we need show only that V(PURGE(u,θ).TOP(u,s))=V(ϕ.TOP(u,s)). Also, if $\theta.TOP(u,s)$ is legal, then by Lemma 1 PURGE(u,θ.TOP(u,s)) is legal, which is equal to PURGE(u,θ).TOP(u,s) by the definition of *PURGE*. If cl(s)>cl(u), then V(PURGE(u,θ).TOP(u,s))=V(ϕ.TOP(u,s))=*violation*, by Axiom 10. If cl(u)≥cl(s) then let ψ be the normal form of PURGE(u,θ). Note that a PUSH to *s* is in ϕ if and only if it is in ψ since if a PUSH is in θ but not in ϕ or not in ψ then it must have been nonsecure or cancelled by a call POP(w,s) where cl(u)≥cl(s)≥cl(w). (Note that a PUSH to *s* could have been purged from θ only if it were nonsecure.) If the PUSH to *s* were nonsecure, then it would be in neither ϕ nor ψ. If it were cancelled by a POP, then this POP would cancel the PUSH in both θ and PURGE(u,θ). If there are no PUSH's to *s* in ϕ or ψ, then V(ϕ.TOP(u,s))=V(ψ.TOP(u,s))=*empty* by repeated application of axiom (13) and axiom (11). If there is such a PUSH, let PUSH(w,s,x) be the rightmost one. By repeated application of axiom (13) and axiom (12) V(ϕ.TOP(u,s))=V(ψ.TOP(u,s))=x, and we are done.

Lemma 4: If ψ is in normal form and L(ψ.TOP(u,s)) is true in some model, then V(PURGE(u,ψ).TOP(u,s))=V(ψ.TOP(u,s)) is true in all models in which ψ.TOP(u,s) is legal.

Proof Sketch of Lemma 4: The proof is by induction. As a base case note that if T=[] the theorem follows from Axiom (11). For the induction step assume that θ is in normal form and V(PURGE(u,θ).TOP(u,s))=V(θ.TOP(u,s)). We shall prove that V(PURGE(u,θ.PUSH(w,r,x)).TOP(u,s))=V(θ.PUSH(w,r,x).TOP(u,s)) where cl(r)≥cl(w). There are two cases. The first is if cl(u)≥cl(w). In this case PURGE(u,θ.PUSH(w,r,x)).TOP(u,s)=PURGE(u,θ).PUSH(w,r,x).TOP(u,s). If r=s then since cl(r)≥cl(w) and cl(u)≥cl(s), we can use Axiom (12). If r≠s then we can use Axiom (13) and the induction hypothesis. The second case is if cl(w)>cl(u). In this case PURGE(u,θ.PUSH(w,r,x)).TOP(u,s)=PURGE(u,θ).TOP(u,s). Since cl(r)≥cl(w)>cl(u)≥cl(s), r≠s. Hence, we can use Axiom (13) and the induction hypothesis, and we are done with our proof of the lemma. The theorem follows by completeness.

4. A Note on Refinement

It should be noted that our theorem is stronger than it may seem at first glance. It states, not merely that our multi-level stack satisfies Noninterference, but that any instantiation of the stack satisfies Noninterference. To see this, say a specification S* is a refinement of S if and only if S* |- S, using the trace derivation system defined in [22]. The intuition behind this definition is that a refinement of S is at least strong as S in that we can derive all the requirements specified in S from the refinement. Hence, if we can derive $L(\theta)$ from S for some trace θ, then we must be able to derive $L(\theta)$ from S*, and if we can derive $V(\theta)=a$ from S, then we must be able to derive $V(\theta)=a$ from S* as well. As a trivial example, every specification is a refinement of the trace deductive system. Similarly, every noninterfering specification is a refinement of the the trace specification of Noninterference.

Given the soundness and the completeness of the trace derivation system, S* |- S if and only if S* |= S, i. e., we can derive S from S* if and only if any model that makes S* true also makes S true. Further, given the transitivity of "|-", any refinement S** of S* is a refinement of S. Hence, any refinement of our multi-level stack specification satisfies Noninterference.[6] As a result, we cannot prove, in general, that a nondeterministic specification where, for example, we can derive $V(T)=a \vee V(T)=b$ for some trace, but nothing stronger, satisfies Noninterference. The problem with such specifications is that an implementor may take advantage of its looseness to pass information from high-level users to low-level users by manipulating when the low-level user sees a and when he sees b. This is not an issue for the classical notion of Noninterference, which does not permit such loose specifications [8].

In this sense, our notion of Noninterference is more general than the classical notion since although we ultimately assume that our implementation is deterministic (V is a function), we do not completely rule out nondeterminism in the form of specification looseness.[7] For example, as noted earlier, our multi-level stack specification does not uniquely characterize the behavior a correct implementation must display when a user whose level is $\perp$ attempts to POP or TOP an empty stack of the same level. The program may abort both the POP and the TOP, may ignore the POP and return *empty* for the TOP, may ignore the first attempt to POP an empty stack but abort the second, etc. The upshot of our proof is that this looseness in functional description does not allow the programmer enough latitude to introduce a security violation. That is, a program's decision to ignore or abort a bottom-level POP of a bottom level stack cannot depend on any high-level information.

It should be noted that not every specification of a noninterfering program can be proven to be noninterfering since it may permit both interfering and noninterfering refinements. This is, in some sense, trivial since, as noted above, every program is a refinement of the trace deductive system, which has no requirements for noninteference. However, every noninterfering program can be viewed as a refinement of some specification that permits only noninterfering refinements. In the worst possible case, the program can be regarded as its own specification, and we can show that the program satisfies Noninterference directly. As argued in the Introduction, however, it is best to prove noninterference on the most abstract-level possible so that our noninterference proof can survive program changes that leave the functional behavior intact. The level we suggest in this paper is the most abstract level at which all refinements are noninterfering.[8] What is perhaps surprising is that our stack specification shows that this level still permits some specification looseness. Given such a specification, we can show that a program is noninterfering by showing that it satisfies the specification. It is to this problem we now turn.

---

6. As noted by Jacob, this is not true for the notion of functional refinement of which the usual notions of refinement used in CSP are examples [16]. McCullough has made similar observations [20].

7. We believe that if we allow true nondeterminism, then the effects of probabilistic channels must be explicitly ruled out as described in [27] and [11].

8. Alternative approaches based on choosing noninterference-preserving refinements of specifications that permit interfering refinements are discussed in [16] and [10].

5. Using Traces for Program Semantics

Our ultimate goal is not simply to prove that a specification is noninterfering, but to prove that a program system is noninterfering. Our method will be to prove that our program system correctly implements a noninterfering specification. We could, of course, reason about program correctness using the standard pre-condition/post-condition approach as developed in [13] and [14]. However, this would involve translating the formal system of trace specifications into the formal system of Hoare-style logic. One problem with this approach is that it is simply inelegant. Why do we need to switch formal systems in midstream rather than using the same formal system to reason about program correctness that we use to reason about specification correctness? A second problem is that the Hoare-style assertions that result from a straightforward translation are unwieldy. Intuitively, the reason for this is that Hoare-style logics are extremely general: they specify the effects of executing a trace by defining the effect for all possible states. As we shall see below, traces must consider only the effect on those states that are reachable.

In this section we present a procedural programming language and show how traces can be used to give the semantics of the language. To this end we extend the notion of *trace* to include strings of program statements. This necessitates introducing program variables and operators into the language as primitives, and introducing statement variables that are like trace variables except that they range over sequences of program statements instead of sequences of procedure calls. Since a sequence of procedure calls is a sequence of program statements, a sequence of procedure calls or procedure call variables is a valid substitution instance for a statement variable.[9] We use T, possibly subscripted, as a procedure call variable and S, possibly subscripted, as a statement variable. The rest of the language is unaffected except for the predicate V. Whereas in [22] V was a function from traces to value domains, in the extended language V takes two arguments, a trace and a program variable, and returns the value of the program variable after the execution of the given trace. When the trace ends in a procedure call and the program variable is the return variable of that call, the second argument can be elided.[10]

Since the trace language contains Boolean expressions and can easily be extended to contain other data types such as integers, lists, or arrays, we are not concerned with their semantics. For simplicity we limit our programming language to integers, and arrays of integers, and we assume axioms for integer functions and relations. Our primary focus is on giving the semantics of the control structures of our language. We are not concerned with developing a complete program semantics, but rather simply to show that reasoning about trace specifications directly is a viable alternative to introducing the extra machinery of finite state machines.

---

9. But a sequence of program statements is not a valid substitution for a procedure call variable. The reason for the distinction is to restrict assertions, such as the induction schema to be given below, to sequences of procedure calls.

10. In [22] V was defined only on legal traces ending in a function call, but in this extended system V can be defined for illegal traces as well since we may wish to reason about program behavior when specifications are violated. This is of small matter, however, since we can either alter [22] to fit by regarding V as being defined on an unspecified set of traces that includes its original domain or by treating V as being systematically ambiguous, letting context decide. If the latter course is taken, then for any function call C that returns variable *ret*, $V(T.C)=x$ if and only if $L(T.C) \wedge V(T.C,ret)=x$.

This limited goal motivates the language *SIMPLE* defined below. We assume the set VBL of integer program variables, ARV of integer array variables, BOOL of Boolean-valued integer expressions, and EXPR of integer-valued integer expressions. For simplicity we assume that the indices for array variables are always expressions of integers and variables. Hence, we do not allow variables of the form $\alpha[POP]$ where α is an array. We are not interested in variable declarations.

SIMPLE

PROGRAM $\rightarrow$
  PROCEDURE NAME[**(**VBL, $\cdots$,VBL**)**]**:** [**RETURN(**VBL**)**] STATEMENT**.**

STATEMENT $\rightarrow$
  skip **|**
  ASSIGNMENT **|**
  IF THEN ELSE **|**
  WHILE DO **|**
  PROCEDURE CALL **|**
  STATEMENT**;** STATEMENT

ASSIGNMENT $\rightarrow$
  VBL **:=** EXPR
  VBL **:=** PROCEDURE NAME[**(**VBL, $\cdots$,VBL**)**]

IF THEN ELSE $\rightarrow$
  if (BOOL**) then {**STATEMENT**} else {**STATEMENT**}**

WHILE DO $\rightarrow$
  while (BOOL**) {**STATEMENT**}**

We assume without loss of generality that all variables are global and that integer variables are initialized to 0. The effect of local variables can be achieved by judicious naming. This does not bring us the full power of block structure, but we will not address this difficult problem or the problem of parameter transmission here. We abbreviate program statements of the form **if (**ϕ**) then {**γ**} else {skip}** as **if (**ϕ**) then {**γ**}**.

Let c be any integer constant, x and y integer variables, a and b any simple or array variables, t, t_1 and t_2 any integer-valued expressions, and β_1 and β_2 Boolean-valued expressions. The semantics for SIMPLE is given by the following axioms (where parentheses and brackets are omitted around Greek letters for readability):

PROGRAM SEMANTIC AXIOMS

(1) $V(S,c)=c$
(2) $V([],n)=0$
(3) $V(S.n:=t,n)=V(S,t)$
(4) $V(S,t_1\ op\ t_2)=V(S,t_1)\ op\ V(S,t_2)$ for arithmetic operation op.
(5) $V(S,\beta_1\ op\ \beta_2)\leftrightarrow V(S,\beta_1)\ op\ V(S,\beta_2)$ for Boolean operation op.
(6) $V(S.a:=t,b)=V(S,b)$ where b is independent, as defined below, of a.
(7) $V(S,\phi)=V(S,\psi)\rightarrow V(S,\alpha[\phi])=V(S,\alpha[\psi])$ where α is an array.
(8) $V(S,\phi)\neq V(S,\psi)\rightarrow V(S.\ \alpha[\phi]:=t,\alpha[\psi])=V(S,\alpha[\psi])$ where α is an array
(9) $V(S,\beta_1)\rightarrow V(S.if\ \ \beta_1\ then\ \ \theta\ else\ \ \psi,x)=V(S.\ \theta,x)$
(10) $\neg V(S,\beta_1)\rightarrow V(S.if\ \ \beta_1\ then\ \ \theta\ else\ \ \psi,x)=V(S.\ \psi,x)$
(11) $V(S.while\ \ \beta_1\ do\ \ \theta,x)=V(S.if\ \ \beta_1\ then.\{\theta.while\ \ \beta_1\ do\ \ \theta\},x)$
(12) $V(S.skip,t)=V(S,t)$

We say that a is *dependent* on b if (1) a is an expression containing b or (2) if a is an array variable of the form $\alpha[\phi]$ and either ϕ is dependent on b or b is of the form $\alpha[\psi]$ for the same array α. We say that a is *independent* of b if a is not dependent on b. The intuition behind the definitions is to restrict axiom (6) so that altering x may alter, e. g., $x+y$, $\alpha[x+y]$ or $\alpha_1[\alpha_2[x]]$, and altering $\alpha[x]$ may alter $\alpha[y]$ if $x=y$.

For proving correctness we shall also find use for the following induction schema:

INDUCTION SCHEMA

Let P be any formula in the trace language that contains a trace constant ϕ and let $P(\psi)$ be the formula that results from replacing ϕ by ψ in P. If $P([])$ and $P(T)\rightarrow(x_1)\cdots(x_n)P(T.C(x_1,\cdots,x_n))$ for each procedure call C that take n variables where $x_1,\cdots,x_n$ are not in P, then $P(T)$ where T does not occur in P.

The schema states that if we can prove that the empty trace has a property P and that if P is preserved by extending a trace by a single procedure call, then we can conclude that all traces have P. It is stronger than it may first seem since it allows us to conclude that any infinite traces which may exist in our model (but, of course, which we cannot denote directly in our syntax) have P. The schema is sound with respect to the semantics given in [22] if we limit the domain of traces to the null trace and finite sequences of procedure calls. This was not done in the original presentation where infinite traces were allowed for the sake of a complete proof theory. The domain limitation renders the resulting system incomplete in the sense that although $\{P([]),P(C),P(C.C),\cdots\}\models P(T)$ in a specification that has the single parameterless procedure call C, we cannot derive $P(T)$ from the given assumption set. In general, it will no longer be true that we can derive every consequence of an infinite set of assumptions from that set, but we can still derive every consequence of a finite set of assumptions from that set. In technical terms, we have given up *compactness*. Compactness can be regained by adequately modifying our proof theory, but we will not go into that here.[11]

The axioms are sufficient for proving weak program correctness, i. e., that if a program terminates then it produces the correct answer. However, they run counter to the

11. See [21] for a discussion of this problem and its solution in the setting of temporal logic.

spirit of the foundation for traces presented in [22] in that axiom (11) says that two expressions may be equal even if they intuitively fail to denote (for example, if a loop fails to terminate).[12] Nondenoting terms are allowed in [22], but any equality involving such terms is considered false. Hence, to render axiom (11) consistent with [22] we would have to supplement our domain with nonstandard denotations. A simpler approach is to limit semantic axiom (11) to terminating programs.

To this end consider the relation $ACC(\phi,\psi,\theta)$, which intuitively says that trace θ can be formed from trace ϕ by appending zero or more occurrences of ψ. We can recursively define such a predicate by adding the following axiom to our system:

(13) $\mathrm{ACC(R,S,R)} \wedge (\mathrm{ACC(R,S,T)} \rightarrow \mathrm{ACC(R,S,T.S)})$

Given this axiom, we can replace axiom (11) by the following axiom, which enables us to prove strong correctness:

(11’) $(\neg \mathrm{V}(\mathrm{S},\phi) \wedge \mathrm{ACC}(\mathrm{T},\theta,\mathrm{S})) \rightarrow$
$\mathrm{V}(\mathrm{T.while}\ \phi\ \mathrm{do}\ \theta,\mathrm{x})=\mathrm{V}(\mathrm{T.if}\ \phi\ \mathrm{then}\ \{\theta.\mathrm{while}\ \phi\ \mathrm{do}\ \theta\},\mathrm{x})$

The resulting axioms are similar to the standard Hoare axioms as presented in [13]. However, the semantic axioms given here differ from Hoare’s in that they are stated in terms of variable values instead of in terms of general preconditions and postconditions. A Hoare-style correctness assertion such as $A\{p\}B$ (if A is true before executing program p, then B is true after p if p successfully terminates) can be stated in the present system as $A^T \rightarrow B^T$ where A^T is the same as A except that every term t is replaced by $V(S,t)$ for some statement variable S and B^T is the same as B except that every term t is replaced by $V(S.p,t)$. Hence, $x=0\{x:=x+1;y:=x\}y=1$ is equivalent to $V(S,x)=0 \rightarrow V(S.x:=x+1.y:=x,y)=1$.

It might seem as though the correct translation should be $V([],x)=0 \rightarrow V(x:=x+1.y:=x,y)=1$. However, such a strategy fails when we consider the assertion $x=1\{x:=x+1;y:=x\}y=2$ since $V([],x)=1$ is false given our assumption that all uninitialized integers are 0. If there is no sequence of statments that can set x to 0, then there will be no S such that $V(S,x)=0$, and hence, our translation will be trivially true. This may seem to be a disadvantage of the trace approach, but it fits well with our desire for abstraction. If a certain state is unrealizable by a module, we have no business placing restrictions on what would happen if that state were realized. As noted above, it is, in fact, this inability to restrict unreachable states that makes the trace notation more compact.

Since we can translate Hoare-style assertions into trace assertions, we can define any language construct that is definable by Hoare-style assertions. We can also give first-order definitions of the constructs of dynamic logic [24].

6. Proving Program Noninterference

Given a specification that satisfies Noninterference, we can prove a program satisfies Noninterference by showing that it satisfies the specification. Hence, security is established *a fortiori* as a by-product of functional correctness. Of course as in all

12. For simplicity, we are not considering nondenoting terms within program statements, e. g., in the assignment **x:=5/0**.

correctness proofs, we must assume that our semantics correctly models the programming language used in the implementation. Further, we must assume that the programming language enforces modularity, i. e., that modules can be accessed only through their component programs and not directly through their internal data structures.[13] Given these assumptions a proof that a program is functionally correct consists of a derivation of the program's specification from the program using the the axioms of [22] and the axioms that define the semantics of SIMPLE. That is, we must show that the program, when interpreted by the semantic axioms of SIMPLE, is a refinement of the specification. To do this we must first determine how to treat some of the constructs found in our specification [23]. For example, consider our initial single-level stack specification and the following implementation:

SIMPLE STACK MODULE

```
int array stack[1...];
int vbl top init(0);

PUSH(int: i):
{top:=top+1; stack[top]:=i}

POP:
{top:=top-1}

TOP return(int: ret)
{ret:=stack[top]}.
```

It is clear that we could show that $V(PUSH(i).TOP)=i$ by using our program semantic axioms to derive the assertion that results from replacing $PUSH(i)$ and TOP by their respective implementations, i. e., $V(top:=top+1.stack[top]:=i.ret:=stack[top],ret)=i$. However, it is not obvious how to derive $L(T.PUSH(i)) \rightarrow V(T.PUSH(i).TOP)=i$. We could side-step this problem by rewriting our program so that $V(T.PUSH(i).TOP)=i$ is always true, but such a "solution" may not always be available. Further, legality is not the only troublesome concept; the same problem arises for equivalence as well.

The problem stems from the fact that legality and equivalence are specification-dependent concepts. For example, we cannot determine whether a sequence of procedure calls is legal simply by looking at an implementation; we must know what the implementation is expected to do.[14] What we need are program-counterparts to legality and equivalence and criteria we can use to show that the program-counterparts chosen are acceptable. Formally, the counterparts will be interpretations of the symbols L and $\equiv$ of our specification language into our program semantic language. To show that a particular interpretation is acceptable, we must show that it preserves the truth of the relevant axioms from the trace derivation system. Given an acceptable interpretation, we must

13. This assumption corresponds to the assumption made in the traditional approach of proving Noninterference that the view function completely captures the low-level user's view of the system.

14. This problem is not limited to program verification. Analogous problems arise in program testing where one must determine whether user invisible effects of running test data on component programs affect system correctness. For example, see [4].

show that we can derive our interpreted specification axioms by applying our program semantic axioms to our implementation. In effect, we are showing that the program is a model of the specification and trace derivation system if we interpret L and $\equiv$ by their program-counterparts.

In general, program counterparts are used to derive "implementation-relative" versions of the corresponding specification assertions and trace axioms. This does not imply that the original implementation-free versions are otiose, however. The implementation-free specification provides a foundation for all implementations; it is what programmers use to understand what their modules should do. Similarly, the implementation-free trace axioms provide a foundation for all specifications. Programmers use both to formulate program-relative concepts of legality and equivalence. If we were to consider different implementations of the same specification, the correctness assertions would differ to reflect different program-relative definitions of *legality* and *equivalence*, but they would still be counterparts of the same specification assertions and trace axioms. The advantage of the trace approach is that it combines abstract specifications with program correctness statements in a clear, coherent manner.[15] The program specification leaves programmers free to choose the best implementation, yet provides a framework for them to formulate program correctness assertions from their programs later.

To make the discussion more concrete, consider the following stack implementation. For simplicity we assume only two users, *high_user* and *low_user*, and two stacks, *high_stack* and *low_stack*.

2-LEVEL STACK MODULE

```
int array high_stack[1...], low_stack[1...];
int vbl high_top init(0), low_top init(0);
char const high_user, low_user;

PUSH(char: user, int array: stack, int: i):
{if (user=high_user | user=low_user) & stack=high_stack
    then
    {high_top:=high_top+1;
     high_stack[high_top]:=i};
 if user=low_user & stack=low_stack
    then
    {low_top:=low_top+1;
     low_stack[low_top]:=i}}
```

15. This work is obviously related to Hoare's use of abstraction functions and representation invariants [14]. As described above, the advantage of the method proposed here is compactness and uniformity of framework: there is a single language and derivation system for reasoning about module specifications and their implementations.

```
POP(char: user, int array: stack)
{if (user=high_user | user=low_user) & stack=high_stack
   then
   {if high_top=0
      then skip;
      else high_top:=high_top-1};
 if user=low_user & stack=low_stack
   then
   {if low_top=0
      then skip;
      else low_top:=low_top-1}}


TOP(char: user, int array: stack) return(int: ret)
{if user=high_user & stack=high_stack
   then
   {if high_top=0
      then ret:="empty";
      else ret:=high_stack[high_top]};
 if (user=high_user | user=low_user) & stack=low_stack
   then
   {if low_top=0
      then ret:="empty";
      else ret:=low_stack[low_top])};
 if user=low_user & stack=high_stack
   then ret:="violation"}}.
```

Given such an implementation, the program counterpart for legality is trivial since, in effect, everything is legal. Hence, we can say that $L(T)$ is true whenever $T=T$. The program-counterpart for equivalence that suggests itself is one where two traces are equivalent if they agree in value for both *high_top* and *low_top* and if they agree in value for $high\_stack[1], \cdots, high\_stack[high\_top]$ and for $low\_stack[1], \cdots, low\_stack[low\_top]$. In other words T is equivalent to S if and only if $(V(T, high\_top) = V(S, high\_top) \wedge (V(T, low\_top) = V(S, low\_top) \wedge (1 \le i \le V(T, high\_top) \rightarrow V(T, high\_stack[i]) = V(S, high\_stack[i]))). \wedge (1 \le i \le V(T, low\_top) \rightarrow V(T, low\_stack[i]) = V(S, low\_stack[i])))$.

To show that our program is correct, and *a fortiori* secure, we need to establish that the relevant legality and equivalence axioms of [22] hold for our interpretation and that we can derive the stack specification given our interpretation. In other words, we must show that we can derive the stack specification and our trace axioms from our program semantic axioms, our stack implementation, and the two axioms $L(T) \leftrightarrow T=T$ and $T \equiv S \leftrightarrow (V(T, high\_top) = V(S, high\_top) \wedge (V(T, low\_top) = V(S, low\_top) \wedge (1 \le i \le V(T, high\_top) \rightarrow V(T, high\_stack[i]) = V(S, high\_stack[i]))). \wedge (1 \le i \le V(T, low\_top) \rightarrow V(T, low\_stack[i]) = V(S, low\_stack[i])))$. The proof of correctness is contained in the Appendix. Proofs of a more involved example can be found in [24].

An examination of the proof reveals that it contains no assumptions about the security levels of local variables such as *high_top* and *low_top*. In fact, the proof does not assume that local variables have an associated security level at all. One benefit of the abstract verification approach suggested here is that since local variables are invisible to the user except insofar as they affect user-visible variables, there is no need for an implementation to assign to them a security level. This is in contrast to the standard approach of proving Noninterference where the unwinding conditions require that every object in the system state must have a security level [9] and to the approach of the Bell and LaPadula model where every object must have a security label for the model to be applicable [3, 26]. In the approach presented here there is no reason not to replace *high−top* and *low−top* by a single variable that encodes both values. Such a program would fail to satisfy Bell and LaPadula and can only be handled by unwinding if we stopped unwinding before we reached the level of abstraction in which our encoding was introduced.

7. Conclusions and Future Work

We have shown how to prove Noninterference directly. By maintaining specification abstraction this approach seems more straightforward than the traditional approach and has the advantages of abstraction. Our method extends the trace formal abstract specification language to a program semantics language and shows how such a language can be used to prove program correctness. This addresses concerns that trace specifications are unusable for program correctness proofs and offers a uniform framework that allows us to move between abstract assertions and concrete assertions in a formal manner. We hope this paper will encourage others to try the approach on larger systems to see how well it scales up and to develop trace-based verification systems.

Acknowledgements

I am grateful to Jim Gray, Jeremy Jacob, Richard Kemmerer, Carl Landwehr, Catherine Meadows, and several anonymous referees for providing helpful comments on various versions of this paper. The method for proving Noninterference described here was first presented at the workshop, Mathematical Concepts of Dependable Systems, held at Mathematisches Forschunginstitut Oberwolfach.

References

1. J. Andrews and G. MacEwen, ‘‘A Review of Tools and Methods for System Assurance,’’ Report 2207-6-AF54/02-SV, Canadian Department of National Defence Research and Development Branch, 1990.
2. W. Bartussek and D. L. Parnas, ‘‘Using Traces To Write Abstract Specifications For Software Modules,’’ Report TR 77-012, University of North Carolina, Chapel Hill, N. C., December 1977.
3. D. E. Bell and L. J. LaPadula, ‘‘Secure Computer System: Unified Exposition and Multics Interpretation,’’ MTR-2997, MITRE Corp., Bedford, MA, March, 1976. Available as NTIS AD A023 588.
4. L. Bougé, N. Choquet, L. Fribourg, and M. Gaudel, ‘‘Application of Prolog to Test Sets Generation from Algebraic Specifications,’’ in *Proc. Intl. Joint Conference on*

Theory and Practice of Software Development, Springer-Verlag, March, 1985.

5. C. Cross, ‘‘A Trace Specification of the MMS Security Model,’’ NRL Report 6216, Naval Research Laboratory, Washington, D. C., 1988.
6. O. J. Dahl, ‘‘Object Orientation and Formal Techniques,’’ in *Lecture Notes in Computer Science, vol 428*, Springer Verlag, New York, 1990.
7. T. Fine, ‘‘Constructively Using Noninterference to Analyze Systems,’’ in *Proc. 1990 IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society Press, May 1990.
8. J. A. Goguen and J. Meseguer, ‘‘Security Policies and Security Models,’’ in *Proc. 1982 IEEE Symposium on Security and Privacy*, pp. 11-20, IEEE Computer Society Press, April, 1982.
9. J. A. Goguen and J. Meseguer, ‘‘Unwinding and Inference Control,’’ in *Proc. 1984 IEEE Symposium on Security and Privacy*, pp. 75-85, IEEE Computer Society Press, April, 1984.
10. J. Graham-Cumming and J. Sanders, ‘‘On the Refinement of Non-Interference,’’ *Proc. Computer Security Foundations Workshop IV*, pp. 35-42, Franconia, New Hampshire, June 1991.
11. J. Gray, ‘‘Toward a Mathematical Foundation for Information Flow Security,’’ *Proc. 1991 IEEE Symposium on Security and Privacy*, pp. 21-34, IEEE Computer Society Press, Oakland, CA., 1991.
12. C. Heitmeyer and J. McLean, ‘‘Abstract Requirements: A New Approach and Its Application,’’ *IEEE Transactions on Software Engineering*, vol. SE-9, no. 5, pp. 580-589, September 1983.
13. C. A. R. Hoare, ‘‘An Axiomatic Basis for Computer Programming,’’ *Comm. ACM*, vol. 12, no. 10, pp. 576-580, October 1969.
14. C. A. R. Hoare, ‘‘Proof of Correctness of Data Representations,’’ *Acta Informatica*, vol. 1, pp. 271-281, 1972.
15. D. Hoffman, ‘‘The Trace Specification of Communication Protocols,’’ *IEEE Transactions on Computers*, vol. c-34, pp. 1102-1113, Dec. 1985.
16. J. Jacob, ‘‘On the Derivation of Secure Components,’’ *Proc. 1989 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Oakland, CA., 1989.
17. D. Lamb, *Software Engineering: Planning for Change*, Prentice Hall, Englewood Cliffs, 1988.
18. L. Lamport, ‘‘A Simple Approach to Specifying Concurrent Systems,’’ *Communications of the ACM*, vol. 32, no. 1, pp. 32-45, January 1989.
19. D. McCullough, ‘‘Specifications for Multi-Level Security and a Hook-up Property,’’ in *Proc. 1987 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, April 1987.
20. D. McCullough, ‘‘Noninterference and the Composability of Security Properties,’’ in *Proc. 1988 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, April 1988.

21. J. McLean, ‘‘A Complete System of Temporal Logic for Specification Schemata,’’ in *Logics of Programs*, ed. D. Kozen, pp. 360-370, Springer-Verlag, New York, 1984.

22. J. McLean, ‘‘A Formal Method for the Abstract Specification of Software,’’ *J. ACM*, vol. 31, no. 3, pp. 600-627, July 1984.

23. J. McLean, ‘‘Two Dogmas of Program Specification,’’ *Proc. of Verification Workshop III*. In *ACM SIGSOFT Softw. Eng. Notes*, vol. 10, no. 4, pp. 85-87, Aug. 1985.

24. J. McLean, ‘‘Using Trace Specifications for Program Semantics and Verification,’’ Report 9033, Naval Research Laboratory, April 1987.

25. J. McLean and C. Meadows, ‘‘The Reliable Specification of Software,’’ in *Proc. COMPASS 88*, IEEE, 1988.

26. J. McLean, ‘‘Specifying and Modeling Computer Security,’’ *IEEE Computer*, vol. 23, no. 1, pp. 9-16, January 1990.

27. J. McLean, ‘‘Security Models and Information Flow,’’ in *Proc. 1990 IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society Press, May 1990.

28. D. L. Parnas, ‘‘The Use of Precise Specifications in the Development of Software,’’ in *Proceedings of IFIP 77*, pp. 861-867, North Holland, 1977.

29. D. L. Parnas and Y. Wang, ‘‘The Trace Assertion Method of Module Interface Specification,’’ Report 89-261, Department of Computing and Information Science, Queen’s University, 1989.

Appendix

This Appendix contains the correctness proof for the multi-level stack implementation given above. The proof consists of substituting the program counterparts for legality and equivalence into the program specification assertions and the trace derivation system axioms and then showing that we can derive the resulting assertions from the trace program semantic axioms and the trace derivation system.

The proof of specification assertions (1)-(3), (10), and (11) are the easiest. The proof for assertions (1)-(3) is trivial since substituting the counterpart for legality in the assertion $L(T)$ yields the identity $T=T$. Proving specification assertions (10)-(11) requires no substitutions of counterparts, although we must note that assertions of the form $V(T.TOP)=x$ must be rewritten as $V(T.TOP, ret)=x$. Given this, assertion (10) follows from applying semantic axioms (11) and (12) to the program TOP once we realize that cl(s)>cl(u) implies that s=high_stack and u=low_user. Assertion (11) follows from applying semantic axioms (4), (11), and (12) to the program TOP once we realize that cl(u)>cl(s) implies that u=high_user and s=low_stack.

Proving specification assertions (12) and (13) also requires no substitutions, but their proofs necessitate trace induction. For each assertion the base case where T=[] is analogous to the arguments given for assertions (10) and (11). Also for each assertion if we assume the assertion holds for T, then it holds for T.PUSH(u,s,x), T.POP(u,s), and T.TOP(u,s) by analogous arguments.

Substituting the counterpart for equivalence in specification assertion (4) yields the assertion cl(u)>cl(s) → (V(T.PUSH(u,s,x),high_top)=V(T,high_top) ∧ V(T.PUSH(u,s,x),low_top)=V(T,low_top) ∧ (1 ≤ i ≤ V(T.PUSH(u,s,x),high_top) → V(T.PUSH(u,s,x),high_stack[i])=V(T,high_stack[i])) ∧ (1 ≤ i ≤ V(T.PUSH(u,s,x),low_top) → V(T.PUSH(u,s,x),low_stack[i])=V(T,low_stack[i]))). Despite its length, the assertion is straightforward to prove. The first two conjuncts of the consequent follow from arguments analogous to those used above for assertions (10) and (11). The third and fourth conjunct can be established by trace induction. Similar arguments apply for assertions (6) and (7).

Substituting counterparts in assertion (8) yields the assertion V(POP(u,s),high_top)=V([],high_top) ∧ V(POP(u,s),low_top)=V([],low_top) ∧ (1 ≤ i ≤ V(POP(u,s),high_top) → V(POP(u,s),high_stack[i])=V([],high_stack[i])) ∧ (1 ≤ i ≤ V(POP(u,s),low_top) → V(POP(u,s),low_stack[i])=V([],low_stack[i])). The first two conjuncts follow from program semantic axioms (4), (11), and (12). The second two conjuncts are vacuously true.

The counterpart for assertion (9) is the assertion V(T.TOP(u,s),high_top)=V(T,high_top) ∧ V(T.TOP(u,s),low_top)=V(T,low_top) ∧ (1 ≤ i ≤ V(T.TOP(u,s),high_top) → V(T.TOP(u,s),high_stack[i])=V(T,high_stack[i])) ∧ (1 ≤ i ≤ V(T.TOP(u,s),low_top) → V(T.TOP(u,s),low_stack[i])=V(T,low_stack[i])). This assertion can be shown by trace induction.

The only assertions left to prove are the counterparts for the trace deductive axioms $L([])$, $L(T.S) \rightarrow L(T)$, $V(T)=x \rightarrow L(T)$, and the definition of ≡. The first three are trivial given our interpretation of $L(T)$ as $T=T$. Given that TOP is the only program to return a value, the counterpart to the definition of ≡ is (V(T,high_top)=V(S,high_top) ∧ V(T,low_top)=V(S,low_top) ∧ (1 ≤ i ≤ V(T,high_top) → V(T,high_stack[i])=V(S,high_stack[i])) ∧ (1 ≤ i ≤ V(T,low_top) → V(T,low_stack[i])=V(S,low_stack[i]))) ↔ (R)(T.R=T.R ↔ S.R=S.R ∧ (u)(s)(T.R.TOP(u,s)=T.R.TOP(u,s) → V(T.R.TOP(u,s),ret)=V(S.R.TOP(u,s),ret))). Dropping trivial identities from this assertion, we have (V(T,high_top)=V(S,high_top) ∧ V(T,low_top)=V(S,low_top) ∧ (1 ≤ i ≤ V(T,high_top) → V(T,high_stack[i])=V(S,high_stack[i])) ∧ (1 ≤ i ≤ V(T,low_top) → V(T,low_stack[i])=V(S,low_stack[i]))) ↔ (R)(u)(s)V(T.R.TOP(u,s),ret)=V(S.R.TOP(u,s),ret). The implication from left to right can be shown by trace induction. Going the other direction we show that (R)(u)(s)V(T.R.TOP(u,s),ret)=V(S.R.TOP(u,s),ret) → V(T,high_top)=V(S,high_top) by reductio. Assume that V(T,high_top)=n>V(S,high_top)=m. Let cl(u)>cl(s) and let R be the trace composed of m occurrences of POP(l,s) where cl(s)=cl(l). It is straightforward to show that V(S.R,high_top)=0 and hence, that V(T.R.TOP(u,s),ret)≠V(S.R.TOP(u,s),ret)=*empty*. The fact that (R)(u)(s)V(T.R.TOP(u,s),ret)=V(S.R.TOP(u,s),ret) → V(T,low_top)=V(S,low_top) follows by analogous argument. To show that (R)(u)(s)V(T.R.TOP(u,s),ret)=V(S.R.TOP(u,s),ret) → (1 ≤ i ≤ V(T,high_top) → V(T,high_stack[i])=V(S,high_stack[i])) assume that there is some n such that 1 ≤ n ≤ V(T,high_top), yet V(T,high_stack[n])≠V(S,high_stack[n]). Let cl(u)>cl(s) and let R be the trace composed of m occurrences of POP(l,s) where cl(s)=cl(l) and m=V(T,high_top)-n. It is straightforward to show that

V(T.R.TOP(u,s),ret)≠V(S.R.TOP(u,s),ret). The argument that (R)(u)(s)V(T.R.TOP(u,s),ret)=V(S.R.TOP(u,s),ret) → (1 ≤ i ≤ V(T,low_top) → V(T,low_stack[i])=V(S,low_stack[i])) is analogous, and we are done.