Using Traces Based on Procedure Calls to Reason About Composability

Catherine Meadows
Code 5543
Naval Research Laboratory
Washington, DC 20375

Abstract

Information flow models are usually conceived in terms of requirements on system traces, while verification that a system satisfies information flow requirements is usually done in terms of a state machine specification. The necessary translation from one model to another may result in a loss of understandability and expressiveness. Recently, McLean has pointed the way to a solution to this problem by showing how a language based on traces of procedure calls may be used to reason about security, and how one may prove that a program satisfies a specification written in that language. The language that he uses, however, does not easily lend itself to specification of composition of communicating processes. In this paper we modify the language so that it is possible to specify the composition of systems. We also describe several different information flow properties analogous to properties that have been defined for other systems and discuss their composability.

1 Introduction

A number of information flow models are stated in terms of requirements on traces of a system: for example, Goguen and Meseguer's noninterference [2], McCullough's restrictiveness [10], Sutherland's nondeducibility [18], and McLean's FM [15]. This is a natural approach. The ability of a system to hide data is measured in terms of what a user sees, and what a user sees is a sequence of inputs and outputs. Everything about this sequence, including the order in which outputs appear, and the way in which inputs and outputs are interleaved, may leak information about sensitive data. Moreover, when we want to reason about the composition of systems, it is easy to construct the composition of traces. One simply identifies certain outputs events in one system with certain input events in the other. A trace in the composed system is legal if it is made up of an interleaving of the traces in each subsystem and its restriction to each subsystem is a trace of that system. However, specifying a system in terms of permissible sequence of inputs and outputs does not usually lead easily to an implementation. Thus, trace requirements are often translated into state machine requirements, as in Goguen and Meseguer's unwinding theorem [5] and McCullough's state machine version of restrictiveness [11]. Although this may be convenient, one loses something by this, since the state machine version is not as closely connected to what the user sees.

In [13] McLean develops a version of noninterference for a trace specification language originally proposed by Bartussek and Parnas [1] in which a system, instead of being specified in terms of permissible sequences of events, is specified in terms of sequences of procedure calls that under certain circumstances will return specified values. We will refer to this type of specification as a call-based trace specification, as opposed to the more usual event-based specification, in which a trace is a sequence of communication events, which may be input events, output events, or internal events. McLean shows how one would prove that a specification satisfies noninterference, and also how one can show that a program satisfies the specification in a straightforward way. Thus, this version of trace specification, while it retains the intuitive appeal of the event-based traces, can also be used to assist in system specification and verification, as well as in the abstract definition of a security property.

However, call-based traces also suffer several disadvantages. One of the most important is that, when we are given specifications of two communicating systems, it is not easy to see how one composes them into a specification of one system. Given the recent interest in composability of secure systems, this can be a serious drawback. Thus, the purpose of this paper is to extend the trace specification language used by McLean in [13] to allow for composition of communicating systems. We also define several different information flow properties for nondeterministic trace specifications and discuss their composability.

The rest of the paper is organized as follows. In Section 2, we give a brief description of the version of the Bartussek-Parnas call-based trace specification language that we will use, and we compare it with the more common event-based notion of traces. In Section 3 we describe how to express equivalence of traces in a nondeterministic Bartussek-Parnas specification. In Section 4 we present our extension of the language to allow composition. In Section 5 we present several ver-
sions of noninterference analogous to other versions of noninterference for nondeterministic systems and discuss their composability. Section 5 concludes the paper.

2 Call-Based Traces

2.1 Bartussek-Parnas Traces

The version of call-based traces that we use is the one originally proposed by Bartussek and Parnas [1] as it was formalized by McLean [12] and further modified by Hoffman and Snodgrass [7].

According to McLean’s definition, a trace specification for a module consists of two parts: (1) a syntax section that gives the procedure names and types the module comprises, and (2) a semantics section that gives the behavior that the module’s procedures must exhibit. Procedure behavior is given by listing assertions that describe the behavior of sequences of procedure calls, written \( \{ \text{call}1, ..., \text{call}n \} \) (where the ‘..’ denotes concatenation), known as traces. These assertions tell us what traces are legal, what the values of traces ending in function calls are and what traces are equivalent. The empty trace, \( \{ \} \), is always assumed to be legal, and so is any initial subtrace of a legal trace. Two traces \( S \) and \( R \) are equivalent, and only if for any trace \( T \), \( L(S,T) \) if \( L(R,T) \) and for non-null \( T \), \( V(S,T) = V(R,T) \), if defined. We denote legality of a trace \( T \) by the predicate \( L(T) \), value by the function \( V(T) \), and equivalence of two traces \( S \) and \( T \) by the predicate \( S \equiv T \).

In Hoffman and Snodgrass’ modification, a set of normal form traces is defined and denoted by the predicate \( \text{isnf} \). The empty trace is always in normal form, and any initial subtrace of a normal form trace is also in normal form. Any assertion about the equivalence of traces must be of the form

\[
\text{Conditions } \implies A.C \equiv B
\]

where \( A \) and \( B \) stand for normal form traces \( C \) is a single procedure call, \( \text{Conditions} \) is a set of conditions on \( A \), \( C \), and \( B \), and \( A \cdot C \) is not in normal form. Any assertion about the value of a trace must be of the form

\[
\text{Conditions } \implies V(A.C) = D
\]

where \( A \) is in normal form, \( B \) is a single procedure call returning a value, and \( \text{Condition} \) is a set of conditions on \( A \) and \( C \). A trace is considered to be legal if it can be reduced to a normal form via application of a sequence of equivalence assertions.

As an example, consider the following specification of a stack. The stack specification has three types of procedure calls, push, pop, and top. A push call inserts an integer into the stack, the pop call removes the top integer on the stack, and the top call returns the top integer on the stack. The predicate \( \text{all}(P,T) \) used below is true if \( T \) consists entirely of procedure calls of type \( P \).

\[
\begin{align*}
\text{SYNTAX stack IS} & \quad \text{PROCEDURE CALL push : integer;} \\
& \quad \text{PROCEDURE CALL pop; } \\
& \quad \text{PROCEDURE CALL top } \Rightarrow \text{ integer END}
\end{align*}
\]

\[
\begin{align*}
\text{SEMANTICS stack IS} & \quad \text{all(push, } T) \\
& \quad \implies \text{isnf}(T); \\
& \quad \text{isnf}(T).push(N)) \implies T.push(N).pop \equiv T; \\
& \quad \text{isnf}(T) \implies T.top \equiv T; \\
& \quad \text{isnf}(T.push(N)) \implies V(T.push(N).top) = N
\end{align*}
\]

The use of Hoffman and Snodgrass’ method of defining traces provides us with a ready-made procedure for executing them. For example, if one wishes to compute the value of a trace \( T.C \) where \( C \) is a procedure call returning a value, one finds the shortest initial subtrace that is not in normal form, uses one of the rules to reduce it to normal form, and iterates this process until one has produced an equivalent trace \( U.C \) such that \( U \) is in normal form. One then applies the relevant rule of the specification to compute the value. A similar procedure can be used to determine whether or not the trace is legal. In [14, 16] we describe in greater detail how we can use this basic procedure to produce executable trace specifications.

2.2 Comparison of Event and Call-Based Traces

We have already indicated the advantages of event-based traces as far as specifying secure systems is concerned. First of all, it is easy to specify intermodule communication; one simply identifies the same event in the traces of two different modules as the output of one module and the input of another. Furthermore, this means of specifying inter-module communication allows one to build new modules out of component modules in a natural way. If module \( M \) has a set of outputs \( O \) that is identical to a set of inputs of module \( N \), one can define the composition of \( M \) and \( N \) by defining the possible traces of the composition of \( M \) and \( N \) to be the set of traces obtained by taking each trace made up of events from \( M \) and \( N \) whose restriction to the events of \( N \) is a possible trace of \( N \) and removing the events from \( O \), which may now be considered internal events of the system.

Bartussek-Parnas traces do not lend themselves so easily to the specification of intermodule communication. This style of trace specification was developed, not in order to reason about interprocess communication, but
to facilitate information hiding. One specifies only interfaces of modules; one does not specify their communication directly. Thus Bartussek-Parnas traces contain less information than Hoare traces. It is not possible, for example, to specify interleaving of inputs and outputs or information about internal events of composed modules. Thus, if we wish to use Bartussek-Parnas traces to specify interprocess communication we will need to modify them in some way.

However, the fact that Bartussek-Parnas traces have certain advantages over event-based traces which we would like to retain makes it worthwhile to consider extending them so that they can be used to specify interprocess communication. The fact that they present the reader with less information means that in many cases they can be used more effectively when that information would be irrelevant. For example, the notion of equivalence of traces provides a relatively efficient way of computing values and legality. One merely reduces the trace to its relatively simple normal form and computes the value of the normal form. Hoffman's heuristics, which provide a general-purpose algorithms for computing normal form equivalents, make this procedure even easier. This is an advantage when one wants to construct executable specifications such as [16]; one is given a ready-made algorithm for execution.

Bartussek-Parnas traces have another, less obviously apparent, advantage, in that they support a notion of system refinement that is more compatible with the needs of specifying security than that usually associated with event-based traces. Consider, for example, the notion of refinement used in CSP.

According to [5] a specification \( S \) is stronger than (or a refinement of) another specification \( T \) if the traces that are possible in \( S \) are also possible in \( T \). As Jacob noted in [8], this notion of refinement does not in general preserve security. Thus it is necessary to develop versions of refinement that preserve security, as is done in [8] and [4].

Bartussek-Parnas traces allow another interpretation of refinement, due to McLean [13]. Informally summarized, McLean's definition of refinement is that a specification \( P_1 \) refines \( P_2 \) if any statement that can be derived from \( P_2 \) can also be derived from \( P_1 \). This means that \( P_1 \) refines \( P_2 \) if

1. The procedure calls of \( P_2 \) are a subset of the procedure calls of \( P_1 \);
2. If \( C \) is a procedure call in \( P_2 \) and \( P_1 \), then it returns a value in \( P_1 \) if and only if it returns a value in \( P_2 \);
3. If \( T \) is legal in \( P_2 \), then it is legal in \( P_1 \), and;
4. If \( T \) is legal in \( P_2 \), and \( X \) is a possible value of \( T \) in \( P_1 \), then \( X \) is a possible value of \( T \) in \( P_2 \).

Note that Condition 1 allows refinement by adding functionality, while Condition 4 allows refinement by reducing uncertainty. Condition 3 can be thought of as allowing refinement by adding functionality or by reducing uncertainty, depending upon whether a trace was originally not legal because it was assumed never to occur, or because the response to it had not yet been specified. Unlike in CSP, it is not possible to eliminate functionality in refinement.

In [13] McLean develops a version of noninterference for trace specifications that allow a limited form of nondeterminism, and shows that it is preserved under this notion of refinement, if we restrict Condition 1 so that the sets of procedure calls of \( P_1 \) and \( P_2 \) are identical.

McLean's definition of refinement also has an advantage in that, if disallow the addition of procedure calls in refinement, it is possible to construct meaningful specifications for which no nontrivial refinement exists, namely specifications in which all traces are legal and each trace that ends in a procedure call that returns a value can return exactly one value. Thus, if necessary, one can sidestep the refinement issue by considering all the "most refined" refinements of a specification.

Thus, although Bartussek-Parnas specifications were not designed with inter-module communication in mind, their advantages from the point of view of executable specification design and specification refinement suggest that, rather than switch over to event-based traces, it might be advantageous to see how Bartussek-Parnas traces can be modified to so that they can be used to model inter-process communication without losing their useful features.

3 Expressing Nondeterminism

The trace specification language allows both deterministic and nondeterministic specifications. However, the fact that equivalence is defined in terms of traces can lead to awkwardness when future values and legality can depend, not only upon the trace itself, but upon its previous values. We thus extend the way in which we talk about traces as follows.

We introduce nondeterminism by allowing the assignment of values to be nondeterministic in that a trace may possibly have more than one value. We make the distinction clear by altering the symbol for value assignment. The assignment \( V(T) = X \) is replaced by the relation \( V(T, X) \). The notation \( V(T) = X \) will now be used to mean that \( T \) has taken on the value \( X \).

If the value or legality of a trace depends upon the values of the previous initial subtraces, then equivalence must be expressed in terms of inputs as well as outputs. For example, consider the following specification of a bag.

**SYNTAX**

```
bag is
  PROCEDURE CALL insert : integer

PROCEDURE CALL pick : ⇒ integer
END
```

**SEMANTICS**

```
bag IS
```

179
Definition 3.1 Let S be a trace specification, and let T be a trace of S. We define a value history of T to be a sequence of possible values of the initial substraces of T. We define the history set of T to be the set of all value histories of T. If h is a value history of a legal trace T, we say (T,h) is a trace-history pair of S.

Thus, in the bag specification, V(insert(1),insert(2).pick) has value histories 1.2 and 2.1. In the first case, 1 is the value of insert(1).insert(2).pick allowed by 1.2, and 2 is the value of insert(1).insert(2).pick allowed by 1.2.

We can now define equivalence and legality. Equivalence is now over traces and value histories instead of just traces.

Definition 3.2 Let S be a trace specification, let T and T' be traces of S, and let h and h' be possible value histories of T and T'. We say that (T,h) and (T',h') are equivalent if, for every nonempty trace X, and history p, (T,X,h,p) is a trace-history pair if and only if (T',X,h',p) is a trace-history pair. We say that a trace T is equivalent to a trace W if, for all histories h and q such that (T,h) and (W,q) are trace-history pairs, we have (T,h) is equivalent to (W,q).

Thus, the trace-history pair (insert(1).insert(2).pick,1) is equivalent to (insert(2),{}), where {} denotes the empty value history.

Note that, although we define equivalence in terms of trace-history pairs, we do not need to change our style of specification writing to reflect this. Instead, we just include conditions of previous values of traces as part of the condition for equivalence. This is possible to do if we write our specification so that the normal form traces consist of procedure calls that do not return values, which in general is the case.

4 Modifying Call-Based Traces to Allow Composition

The idea of modifying Bartussek and Parnas's model to allow inter-module communication is not original with us. In [6] Hoffman proposes a modification to Bartussek-Parnas traces so that they can be used to model communication protocols. In Hoffman's model, a module trace is still a sequence of procedure calls, but includes, not only the calls made on the module (designated in-calls by Hoffman), but calls made by the module on other modules (designated out-calls by Hoffman). In effect, this means that module traces become sequences of communication events in the life of the module, instead of sequences of procedure calls. Such an approach allows one to model communication protocols and other instances of inter-module communication. However, modifying Bartussek-Parnas traces in this way means that one loses many of their valuable features. This is because even though out-calls are outputs of a module, they are not explicitly treated as such. Thus, one cannot use Hoffman's heuristics for computing values to assist in the computation of out-calls. Moreover, since the clear distinction between input and output is lost, we can no longer easily use the notion of system refinement supplied by McLean. Thus we decided to take a different approach, in which communication events could be treated as values returned by procedure calls as well as procedure calls themselves.

Our basic approach is to modify the Bartussek-Parnas style of specification so that values returned by procedure calls can be procedure calls on other modules. A procedure call returned as a value is prefixed by the name of the module to which it refers. We make three restrictions on procedure calls that return procedure calls as values.

1. No loops or infinite spirals are allowed. That is, if C1 is a procedure call from specification S1 returning procedure call C2 on S2 as a value, and C2 returns call C3 on S3, and so on, then the sequence eventually ends, and all the Ci are distinct (although the Si's do not have to be).

2. If C1 is a procedure call from specification S1 returning procedure call C2 on S2 as a value, then S1 and S2 are distinct.

3. The origin and destination of procedure calls returned as values must be unambiguous. Thus, if module S1 sends a procedure call to module S2, then no other module can have produced that call and no other module could receive that call. The destination of a call returned as a value is denoted by the convention AI B, where A is the value and B is the call. We do not enforce any convention for origins of calls. In particular, we do not require that a module can not receive the same type of call from two or more other modules, only that there must be some way of determining from which
module the call originated; for example, one could require the name of the originator of the call to be an argument of the call.

We can now define the composition of two specifications.

We begin by making some basic definitions regarding traces.

**Definition 4.1** Let $T$ be a trace, and let $A$ be a set of procedure calls. We define $T|_A$, the restriction of $T$ to $A$, to be the subtrace of $T$ consisting only of calls from $A$. We define $T - A$ to be the subtrace of $T$ obtained by removing all calls from $A$.

We now define the composition of traces.

**Definition 4.2** Let $S_1$ and $S_2$ be specifications. Let $A_i$ be the set of procedure calls on $S_i$, and let $B_i$ be the set of possible values returned by traces of $S_i$. Let $(T_1,h_1)$ be a trace-history pair from $S_1$, and let $(T_2,h_2)$ be a trace-history pair from $S_2$. Let $A = h_1[A_1]$, and let $B = h_2[A_2]$. Let $T^*$ be an interleaving of $T_1$, $T_2$, $h_1$ and $h_2$, such that, whenever $U.C$ is an initial subtrace of $T_i$, where $C$ is a single procedure call, and $D$ is the value of $U.C$ allowed by $h_i$, then $D$ immediately follows $C$ in $T^*$. Let $T = T^*$ - $(B_1 \cup B_2)$, and let $h = h^* - (A_1 \cup A_2)$. We say that $(T,h)$ is a composition of $(T_1,h_1)$ and $(T_2,h_2)$.

**Example:** Consider specification "flintstones" with procedure calls fred, wilma, barney, and betty. Consider specification "peanuts" with procedure call charley, lucy, linus, and schroeder. Suppose that the following hold:

$V(fred,wilma, peanuts!lucy)$
$V(fred.wilma, peanuts!barney, peanuts!linus)$
$V(charley,lucy,linus,schroeder, flintstones!betty)$
$V(charley,linus, 1)$
$V(fred), 2)$

Suppose that we want to compose the trace-history pairs

$(fred.wilma, barney, betty, 2, peanuts!lucy, peanuts!linus)$

and

$(charley, lucy, linus, Schroeder, flintstones!betty)$

We construct the traces

$(fred.2.wilma, peanuts!lucy, barney, peanuts!linus, betty)$

and

$(charley, lucy, linus, 1, Schroeder, flintstones!betty)$

and then obtain an interleaving

$(fred.2.charley, wilma, lucy, barney, linus, 1, Schroeder, betty)$.

We remove all values to get the trace

$(fred.charley.wilma.barney.schroeder)$

and we remove all procedure calls to get the value history

$(2.1)$

Thus the composed trace-history pair is

$(fred.charley.wilma.barney.schroeder, 2)$

and we assign values as before.

The following technical lemma will be useful in induction proofs.

**Lemma 4.1** The trace-history pair $(T,C,h)$ is a composition of $(T_1,h_1)$ from $S_1$ and $(T_2,h_2)$ from $S_2$, where $C$ is a single procedure call from $S_1$ if and only if there exist $(U_1,k_1)$, $(U_2,k_2)$, $V_1$, $V_2$, $r_1$, and $r_2$ such that:

1. $T_1 = U_1.C.V_1$ and $T_2 = U_2.V_2$;
2. $h_1 = k_1.V_2.r_1$ and $h_2 = k_2.V_1.r_2$, where $r_i = V(T_i)$ if it is defined and not a procedure call on another module, and is empty otherwise;
3. $(T,k)$ is a composition of $(U_1,k_1)$ and $(U_2,k_2)$ and;
4. $h = k.r$ where $r$ is either empty or the nonempty $r_i$ from Part 2.

**Proof:** The proof follows directly from the definition of composition of traces.

Note that we have defined the composition of specifications so that the composition of two deterministic specifications is also deterministic. If we wish to introduce nondeterminism, we can do so, for example, by introducing a specification of a buffer that nondeterministically delays the receipt of messages.

Now that we have defined composition of traces, we need a way of computing with them. Our strategy will be to use equivalence to reduce traces with multiple value histories to traces with single (preferably empty) value histories. We then compute with the simplified traces.

Unfortunately, equivalence is not always preserved under composition; that is, it is not always true that if $(T,h)$ is a composition of $(T_1,h_1)$ and $(T_2,h_2)$, and $(W,p)$ is a composition of $(W_1,p_1)$ and $(W_2,p_2)$, and $(T_1,h_1) \equiv (W_1,p_1)$ and $(T_2,h_2) \equiv (W_2,p_2)$, then $(T,h) \equiv (W,p)$. Consider the following example.

**Example:** Let $S_1$ contain procedure calls $a$ and $b$, and let $S_2$ contain procedure calls $x$, $y$, $z$, and $q$. Suppose
that all traces in $S_1$ are legal, and the $V(T,a,S_2y)$ and $V(T,a,S_2z)$ both hold. Suppose that all traces in $S_2$ are legal, and that $V(T,q)$ is the number of occurrences of $x$ and $y$ in $T$. Then:

1. All trace-history pairs in $S_1$ are equivalent.
2. In $S_2$, $(z,x,\{\}) \equiv (x,\{\})$, but $(y,x,\{\})$ is not equivalent to $(x,\{\})$.
3. The trace-history pair $(b,x,\{\})$ is a composition of $(b,\{\})$ and $(x,\{\})$. These are the only two pairs whose composition is $(b,x,\{\})$.
4. The trace-history pair $(a,x,\{\})$ is a composition of $(a,S_2\{z\})$ and $(z,x,\{\})$, where $(b,x,\{\}) \equiv (a,S_2\{z\})$, and $(x,\{\}) \equiv (x,\{\})$.
5. The trace-history pair $(a,x,\{\})$ is also a composition of $(a,S_2\{y\})$ and $(y,x,\{\})$.
6. The only value possible for $b.x,q$ is 1. However, there are two possible values for $a.x,q$: 1, since the value of $z.x$ is 1, and 2, since the value of $y.x$ is 2. Thus $(b,x,\{\})$ and $(a,x,\{\})$ are not equivalent, even though they are composed of equivalent pairs of trace-history pairs.

The following two lemmas will help us identify situations under which equivalence is preserved.

**Lemma 4.2** Let $S_1$ and let $S_2$ be specifications, and let $S$ be their composition. Let $(T,h)$ be a composition of $(T_1,h_1)$ and $(T_2,h_2)$. Suppose that for each initial subtrace $W_i$ of $T_i$ such that $V(W_i)$ is defined and a procedure call on the other specification, there is at most one possible value for $W_i$. Then $(T_1,h_1)$ and $(T_2,h_2)$ are the only trace-history pairs whose composition yields $(T,h)$.

**Proof.** The proof is by induction on the length of $T$. The result is trivially true for the empty trace. Suppose that it is true for all traces of length less than $n$, and that we wish to prove it for traces of length $n$.

Let $T.C$ be a trace of length $n$ from $S$, where $C$ is a single procedure call. Without loss of generality we may assume that $C$ is from $S_1$. By Lemma 4.1, we have that if $(T.C,h,r)$ is a composition $(T_1,h_1)$ and $(T_2,h_2)$ then $(T_1,h_1) = (U_1.C.V_1.q_1.V_2.r_1)$ and $(U_2.C.V_2.q_2.V_1.r_2)$, where $(T,h)$ is a composition of $(U_1.q_1)$ and $(U_2.q_2)$. By the induction hypothesis, $(U_1.q_1)$ and $(U_2.q_2)$ are the only pairs whose composition is $(T,h)$. By the hypothesis $V_1$ and $V_2$ are also unique, and in order for the pairs to be composable into $(T,h)$, $r_1$ and $r_2$ are unique. Thus $(T_1,h_1)$ and $(T_2,h_2)$ are unique.

**Lemma 4.3** Let $S_1$ and $S_2$ be specifications, and let $S$ be their composition. Let $(T,h)$ and $(W,p)$ be trace-history pairs of $S$. Suppose that, whenever $(T,h)$ is a composition of $(T_1,h_1)$ and $(T_2,h_2)$, there exist $(W_1,p_1)$ and $(W_2,p_2)$ whose composition is $(W,p)$ such that $(T_1,h_1) \equiv (W_1,p_1)$ and $(T_2,h_2) \equiv (W_2,p_2)$. Suppose likewise that the reverse holds for all $(W_1,p_1)$ and $(W_2,p_2)$ whose composition is $(W,p)$. Then $(T,h) \equiv (W,p)$.

**Proof.** We need to show that, for any nonempty trace $X$, $h.q$ is a value history of $T.X$ if and only if $p.q$ is a value history of $W.X$. We prove that $p.q$ is a value history of $W.X$ if $h.q$ is a history of $T.C$, the reverse is similar.

Suppose that $(T,X,h.q)$ is a composition of $(U_1,v_1)$ and $(U_2,v_2)$. By repeated application of Lemma 4.1, we have that $(U_1,v_1) \equiv (T_1.X_1.h_1.q_1)$ and $(U_2,v_2) \equiv (T_2.X_2.h_2.q_2)$, where $(T,h)$ is a composition of $(T_1,h_1)$ and $(T_2,h_2)$. By the hypothesis, there exist pairs $(W_1,p_1)$ and $(W_2,p_2)$ whose composition is $(W,p)$ such that $(W_i,p_i) \equiv (T_i,h_i)$. Thus $(W_1.X_1.p_1.q_1)$ and $(W_2.X_2.p_2.q_2)$ are also trace-history pairs, and, again by repeated application of Lemma 4.1, their composition is $(W.X,p.q)$.

As an example, consider the following specification of two modules, sender and receiver. Characters are inserted into the sender module, and nondeterministically sent to the receiver module whenever it is presented with a send call. The receiver module receives characters from the sender, and may also be used to read characters received from the sender.

**SYNTAX** sender IS
PROCEDURE CALL insert : character
PROCEDURE CALL send : \Rightarrow procedure call
END

**SEMANTICS** sender IS
all(insert, T)
IMPLIES isnf(T);
isnf(T1.insert(C).T2) IMPLIES V(T1.insert(C).T2.send, receiver!get(C));
V(send,"error");
send \equiv \{\};
isnf(T1.insert(C).T2)
& V(T1.insert(C).T2.send) = receiver!get(C)
IMPLIES T1.insert(C).T2.send \equiv T1.T2;
END

**SYNTAX** receiver IS
PROCEDURE CALL get : character
PROCEDURE CALL read : integer \Rightarrow character
END

**SEMANTICS** receiver IS
all(get, T)
IMPLIES isnf(T)

isnf(T)
IMPLIES T.read(N) \equiv T

isnf(T1.get(C).T2)
& length(T1) = N-1
IMPLIES V(T1.get(C)).T2.read(N, C)

isnf(T)
& length(T) \leq N
IMPLIES V(T.read(N), "error")

END

The specification for the composition of the two modules is as follows.

SYNTAX comp IS
PROCEDURE CALL insert : character
PROCEDURE CALL send
PROCEDURE read : integer

SEMANTICS comp IS
isnf(insert(C));

isnf(T)
IMPLIES isnf(T).send;

isnf(T).send.insert(C)
IMPLIES isnf(T.send.insert(C).send);

isnf(T1.T2)
& all(insert,T2)
& (T1 = or T1 = T3.send)
& T2 = T4.insert(C).T5
IMPLIES T1.T2.send
\equiv T1.insert(C).send.T4.T5;

isnf(T1.send.T2)
& all(insert,T2)
& T1.send = T3.insert(C).send.T4
& length(T3) = 2*(N-1)
IMPLIES V(T1.send.T2.read(N),C);

isnf(T)
& count(insert,T) < N
IMPLIES V(T.read(N),"error");

isnf(T)
IMPLIES T.read(N) \equiv T

END

Note that it is not trivial to compute the specification of the composition of the sender and receiver modules. In particular, although the normal form traces of the composition module are made up of procedure calls on the sender module, they are not equal to the normal form traces of that module. This is because the sender module "forgets" that a character has been inserted once it has been sent, and thus only inserts that have not yet been sent have any effect on the future behavior of the trace. However, the receiver module remembers whenever a character has been received, and so the receipt of a character always has an effect on a trace's future behavior. Thus we have shown that if S1 through Sk are specifications with composition S, and T and U are traces in S such that the restrictions of T and U to S are equivalent, it does not follow that T and U are equivalent. Thus it is difficult to tell what the normal form traces of a composition “look like” merely from looking at the projections of a trace onto its component traces.

However, we were able to make use of equivalence to make the task of computing the composition easier. We began by finding a representative set of traces of the send module so that each trace has a unique value. These were not the normal form traces of the send module; however, they did have the property that every trace was equivalent to at least one of these traces. We then found representative traces of the receive module with which these traces could be composed. These turned out to be the normal form traces of the receive module, that is, the traces consisting entirely of gets. We then composed the traces to get candidate normal form traces of the composed module. Next we derived rules for computing value using the rules for computing value for traces in the original send and receive modules. We then proved the following lemma:

Lemma 4.4 Suppose that (T,h) trace-history pair in the composed specification. Then there exists a unique (up to equivalence) pair of representative trace-history pairs (W1,p1) and (W2,p2) such that

1. There exists a composition (W,p) of (W1,p1) and (W2,p2), and;
2. If(T,h) is a composition of (T1,h1) and (T2,h2), then (W1,p1) \equiv (T1,h1) and (W2,p2) \equiv (T2,h2).

Proof Sketch: The proof is by induction on the length of T. If T is empty, the result is trivial.

Suppose that the result is true for T of length less than n. Consider (T,C,h) where T.C is of length n. By Lemma 4.1, (T,C,h) is a composition of (T1.X1,h1.q1) and (T2.X2,h2.q2), where (T,h) is a composition of (T1,h1) and (T2,h2).

By the induction hypotheses, (T1,h1) and (T2,h2) are equivalent to a unique (up to equivalence) pair of composables traces (W1,p1) and (W2,p2). Thus (T1.X1.h1.q1) \equiv (W1.X1,p1.q1) and (T2.X2,h2.q2) \equiv (W2.X2,p2.q2). Thus all we need to do is show the result for (W1.X1,p1.q1) and (W2.X2,p2.q2).

We have three cases to consider:
C \rightarrow \text{read}(N): \text{In this case, } X_1 = q_1 = \{\}, X_2 = \text{read}(N), \text{and } q_2 = A \text{ for some value } A.

C \rightarrow \text{insert}(A): \text{In this case, } X_1 = \text{insert}(A), \text{and } q_1 = X_2 = q_2 = \{\}, \text{and both } (W_1.X_1,p_1.q_1) \text{ and } (W_2.X_2,p_2.q_2) \text{ are representative traces.}

C \rightarrow \text{send}: \text{Recall that } W_1 = W_3.W_4, \text{where } W_4 \text{ consists entirely of inserts, and } W_3 \text{ consists of alternating inserts and sends. If } W_4 \text{ is empty, then } X_1 = \text{send}, q_1 = \text{receiver.error}, X_2 = \{\}, \text{and } q_2 = \{\}. \text{And for each pair } (W_1.X_1,h_1.q_1) \equiv (W_1,h_1). \text{ If } W_4 \text{ is not empty, then } X_1 = \text{send}, q_1 = \text{receiver.get}(A), \text{where insert}(A) \text{ appears in } W_4, X_2 = \text{get}(A), \text{and } q_2 = \{\}. \text{In this case, } (W_1.X_1,h_1.q_2) \text{ is equivalent to the representative pair } (W_3.\text{insert}(A).\text{send}.W_5,h_1.\text{receiver.get}(A)), \text{where } W_5 \text{ is obtained by removing an occurrence of } \text{insert}(A) \text{ for } W_4.

One then checks that each representative pair so obtained is unique up to equivalence.

From Lemma 4.4 one can conclude that, for any trace-history pair \((T,h)\) in the composed module, there is a pair of representative traces \((W_1,p_1)\) and \((W_2,p_2)\) with composition \((W,p)\) such that, if \((T,h)\) is a composition of \((T_1,h_1)\) and \((T_2,h_2)\), then \((T_1.h_1) \equiv (W_1.p_1)\) and \((T_2.h_2) \equiv (W_2.p_2)\). Since \((W_1.p_1)\) and \((W_2.p_2)\) are the only trace-history pairs whose composition is \((W,p)\), the converse is also true. Thus the hypothesis of Lemma 4.3 is satisfied, and we have \((T,h) \equiv (W,q)\).

It follows that the compositions of the representative traces are an acceptable set of normal form traces for the composed module.

5 Applications to Security

In [13] McLean develops a notion of security for call-based trace specifications, and shows how one might prove a specification secure using this notion. The definition of security used is a version of Goguen and Meseguer's noninterference [2]. A purging function is defined over traces. A trace specification is noninterfering for all legal traces \(T.C\) where \(T\) is a low-level procedure call returning a value, \(V(T.C) = V(\text{purge}(T),C)\).

McLean indicates the proof techniques that can be used to prove that a specification is secure by working through an example specification of a multi-level stack. He first proves that a trace is legal if and only if its purge is legal. This fact is necessary in order to prove that the specification satisfies noninterference, since if a trace has no legal purge, then noninterference is trivially not satisfied. He then shows that every legal trace has a normal form. He then shows that if the result holds for all normal form traces, then it holds for all traces. Finally, he proves the result for all normal form traces.

In adapting this approach to our system, we first have to define an acceptable notion of security for nondeterministic traces. A considerable amount of work has already been done in this area, for example that of Sutherland [18], McCullough [11], McLean [15], Jacob [9], and O'Halloran [17]. Much of this recent work, in particular [11], has been devoted to constructing properties that are composable, so that when two specifications satisfying the property are composed in some arbitrary matter, the composition also satisfies the property. In this paper, we are not so much interested in a particular composable security property as in our ability to prove theorems about composable security properties in general. We may also be interested in security properties that are not composable in general, as long as we can discover conditions under which they do compose.

In this paper we look at several generalizations of noninterference to nondeterministic systems. We begin by defining multilevel trace specifications and a purge function.

**Definition 5.1** Let \(S\) be a trace specification. We say \(S\) is multilevel if there is a function level from the procedure calls and possible values of traces of \(S\) to the a partial order \(L\) such that all values returned by a procedure call are of the same level as that call. If \(l\) is an element of \(L,\) define \(\text{purge}_l(T)\) to be the subtrace of \(T\) obtained by removing all procedure calls whose level is not dominated by \(l.\) If \(h\) is a value history of \(T,\) we define \(\text{purge}_l(h)\) to be the history obtained by removing values not dominated by \(l.\) If \(H\) is a set of histories, we define \(\text{purge}_l(H)\) to be the set of all \(\text{purge}_l(h)\) such that \(h\) is an element of \(H.\)

The following technical lemma will be useful later on.

**Lemma 5.1** Let \(S_1\) and \(S_2\) be two multilevel trace specifications, and let \(S\) be their composition. Let \(T.C.V_1\) be a trace of \(S_1,\) let \(T_2.V_2\) be a trace of \(S_2,\) and let \(T.C\) be a composition of \(T_1.C.V_1\) and \(T_2.V_2.\) Then \(C, V_1, \) and \(V_2\) are all at the same level.

**Proof.** The result follows trivially from the fact that all calls in \(V_2\) are either returned by or return calls in \(C\) and \(V_1,\) and the value returned by a procedure call is always at the same level as the call itself.

From now on, we denote \(\text{purge}\) by \(\text{purge}\) when we can avoid confusion, and refer to the calls and values dominated by \(l\) as high, and the calls and values not dominated by \(l\) as low.

We can now define several different versions of noninterference. We start with the weakest.

**Definition 5.2** Let \(S\) be a multilevel trace specification. We say \(S\) is noninterferring if, for all trace-history pairs \((T,h),\) the pair \((\text{purge}(T),\text{purge}(h))\) is also a trace-history pair of \(S.\)

Our definition of noninterference is a special case of the notion of noninterferring used by O'Halloran in [17].
that paper, O’Halloran defines a notion of noninference for two disjoint alphabets A and B so that a specification is noninferring if, for all traces T, either T|B is a trace or T - A is a trace. This reduces to our definition of noninference in the case that A and B make up the entire alphabet.

O’Halloran proves that noninference is composable using a category theoretic framework. We give the proof of composability for our version of noninference below.

**Proposition 5.1** Let S1 and S2 be two noninferring multilevel trace specifications, and let S be their composition. Let \((T_1,h_1)\) be a trace-history pair of S1, let \((T_2,h_2)\) be a trace-history pair of S2, and let \((T,h)\) be a composition of \((T_1,h_1)\) and \((T_2,h_2)\). Then \((\text{purge}(T),\text{purge}(h))\) is a composition of \((\text{purge}(T_1),\text{purge}(h_1))\), and \((\text{purge}(T_2),\text{purge}(h_2))\).

**Proof:** Since S1 and S2 are noninferring, we have that \((\text{purge}(T_1),\text{purge}(h_1))\) and \((\text{purge}(T_2),\text{purge}(h_2))\) are trace-history pairs. It remains to show that \((\text{purge}(T),\text{purge}(h))\) is their composition.

The proof will be by induction on the length of T. Clearly, if T is the empty trace, then the result follows. Suppose the result is true for T of length n. We want to prove the result for T.C, where C is a single procedure call. Without loss of generality, we may assume that C is a procedure call from S1. By Lemma 4.1 we have that the trace-history pair \((T\cdot C,h\cdot r)\) is a composition of \((T_1\cdot C\cdot V_1\cdot h_1\cdot V_2\cdot r_1)\) and \((T_2\cdot V_2\cdot h_2\cdot V_1\cdot r_2)\), where \((T,h)\) is a composition of \((T_1,h_1)\) and \((T_2,h_2)\). By the induction hypothesis, \((\text{purge}(T),\text{purge}(h))\) is a composition of \((\text{purge}(T_1),\text{purge}(h_1))\), and \((\text{purge}(T_2),\text{purge}(h_2))\).

We wish to prove the same for T.C.

There are two cases to consider, the case in which C is high, and the case in which it is low. We begin with the high case.

Suppose that C is high. Then by Lemma 5.1, we have that V1, V2, r1, and r2 are all high, and so

1. \(\text{purge}(T\cdot C) = \text{purge}(T)\);
2. \(\text{purge}(T_1\cdot C\cdot V_1) = \text{purge}(T_1)\);
3. \(\text{purge}(T_2\cdot V_2) = \text{purge}(T_2)\);
4. \(\text{purge}(h_1\cdot V_2\cdot r_1) = \text{purge}(h_1)\), and;
5. \(\text{purge}(h_2\cdot V_1\cdot r_2) = \text{purge}(h_2)\).

Thus the result follows directly from the induction hypothesis.

Suppose now that C is low. Then we have:

1. By Lemma 5.1, we have that V1, V2, r1, and r2 are all high, and so
   \(\text{purge}(T_1\cdot C\cdot V_1) = \text{purge}(T_1)\cdot C\cdot V_1\), \(\text{purge}(T_2\cdot V_2) = \text{purge}(T_2)\cdot V_2\),
   \(\text{purge}(h_1\cdot V_2\cdot r_1) = \text{purge}(h_1)\cdot V_2\cdot r_1\), and
   \(\text{purge}(h_2\cdot V_1\cdot r_2) = \text{purge}(h_2)\cdot V_1\cdot r_2\).

2. By the induction hypotheses, we have that the pair \((\text{purge}(T),\text{purge}(h))\) is a composition of \((\text{purge}(T_1),h_1)\) and \((\text{purge}(T_2),h_2)\).

3. By Steps 1 and 2 and noninference, we have that \((\text{purge}(T\cdot C),V_1,\text{purge}(h_1),V_2\cdot r_1)\) and \((\text{purge}(T_2),V_2,\text{purge}(h_2),V_1\cdot r_2)\) are trace-history pairs.

4. By Step 3 and Lemma 4.1, \((\text{purge}(T\cdot C),h\cdot r)\) is a composition of \((\text{purge}(T_1\cdot C\cdot V_1),\text{purge}(h_1),V_2\cdot r_1)\) and \((\text{purge}(T_2\cdot V_2),\text{purge}(h_2),V_1\cdot r_2)\).

Noninference is composable, but it is a rather weak security property. It tells us that, if the low user sees a trace-history pair, then that trace-history pair is compatible with there having been no high inputs. But we may be able to tell what the high inputs were if there were any, and we may be able to tell if there were no high inputs.

**Example:** Let S be a trace specification with one low procedure call a, and one high procedure call b. Suppose that V(T.a) = 1 if V contains at least one occurrence of b, and V(T.a) = 0 or 1 otherwise. Then S is noninfering. But, if V(T.a) = 0, a low user can infer that no high inputs have occurred.

We would like to extend noninference to a stronger security property that is also preserved under composition. We consider the following properties.

**Definition 5.3** Let S be a multilevel trace specification. We say that S is weakly noninfering if S is noninfering and, whenever h is a value history of \(\text{purge}(T)\), there is a history \(h'\) of T such that \(\text{purge}(h') = h\).

Weakly noninfering specifications hide evidence of high input, but not all evidence of high output.

**Example:** Let S be a specification with a low procedure call a and a high procedure call b. Suppose that V(T.b) is either 0 or 1, that V(T.a) is 0 or 1 if T contains no occurrences of b, and otherwise V(T.a) is the exclusive-or of all V(Q.b) such that Q.b is an initial subtrace of T. The specification S is weakly noninfering, but it reveals information about the values of high procedure calls, if there are any in the trace.

Weak noninference is similar to McCullough’s generalized noninfering, and like it, is not composable. We demonstrate this by an example similar to the one in [1].

**Example:** Module A has two high procedure calls insert(X) and insert.from(B,X), where X can be 0 or 1. It has two low procedure calls, stop.count and query. The value of T.insert(X) is Blinset(Y), where Y may be 0 or 1. The value of T.query is either 0 or 1 if T contains no high events, or it is the par-
history of insert(1).insert_from_B(1) is B\text{insert}(1), then V(\text{insert}(1).\text{insert_from_B}(1).\text{stop_count}.\text{query}) = 1. The value of T.stop_count is B.stop_count.

Module B has one high procedure call: insert(X). The value of T.insert(X) is A\text{insert_from_B}(1) = B\text{insert}(1), where Y can be 0 or 1. Module B has one low procedure call, stop_count. If T contains no high procedure calls, then V(T.stop_count) may be 0 or 1. If T contains high procedure calls, then V(T.stop_count) is the parity of the arguments of all high events.

In the composed specification V(T.stop_count) is the parity of the arguments of all high events occurring in A. The only events not shared between A and B are the events in the history of insert(1).insert_from_B(1) is B\text{insert}(1), possible specification, the possible value histories of (purge(T),purge(h)).

There is also a stronger notion of noninterference that we define below. This hides evidence of high output well.

Definition 5.4 Let S be a multilevel trace specification. We say that S is strongly noninterfering if S is weakly noninterfering and, whenever (T,h) is a trace-history pair, and (purge(T).C,purge(h).r) is a trace-history pair for a low procedure call C, then (T.C,h.r) is a trace-history pair.

Strong noninterference is similar to, but slightly stronger than, the version of noninterference proposed by Graham-Cumming and Sanders in [4]. It can be shown equivalent under very certain conditions.

Definition 5.5 Let S be a multilevel trace specification. We say that S is high input-total if, for every trace-history pair (T,h) and every high procedure call C, there is an r such that (T,C,h.r) is a trace-history pair.

We present a Bartussek-Parnas version of Graham-Cumming and Sanders' definition of noninterference below.

Definition 5.6 Let S be a multilevel trace specification, and let (T1,h1) and (T2,h2) be two trace-history pairs. We say that (T1,h1) and (T2,h2) are low-equivalent if, for every (X,q), we have (T1.X,h1.q) = (T2.X,h2.q) is a trace-history pair if and only if (T2.X,h2.q) is.

Definition 5.7 Let S be a multilevel trace specification. We say that S is GCS non-interfering if every trace-history pair (T,h) is low-equivalent to (purge(T),purge(h)).

Proposition 5.2 Let S be high input-total multilevel trace specification. Then S is strongly noninterfering if and only if it is GCS noninterfering.

Proof: Suppose that S is strongly noninterfering. We need to show that, if (T,h) is a trace-history pair, then it is low-equivalent to (purge(T),purge(h)).

Let X be a trace and q a value history. We need to show that (T.X,h.q) is a pair if and only if (purge(T).X,purge(h).q) is. If (T.X,h.q) is a trace-history pair, it follows from weak noninterference that (purge(T).X,purge(h).q). It (purge(T).X,purge(h).q) is a trace-history pair, it follows from repeated applications of the definition of strong noninterference that (T.X,h.q) is.

Suppose, conversely, that S is GCS noninterfering. We need to show that:
1. If (T,h) is a trace-history pair, then so is (purge(T),purge(h)).
2. If T is a legal trace such that (purge(T).h) is a trace-history pair, then there exists an h' such that purge(h') = h and (T.h') is a trace-history pair.
3. If (T,h) is a trace-history pair, and (purge(T).C,purge(h).r) is a trace-history pair for some low procedure call C, then so is (T.C,h.r).

Proof of (1): Suppose that (T,h) is a trace-history pair. The proof will be by induction on the number of low procedure calls in T.

If the trace T does not contain any low procedure calls, then (purge(T),purge(h)) = (\{\},\{\}), which is trivially a trace-history pair. Suppose that the result is true for Suppose that T contains low procedure calls, and let C be the last one. Thus (T.h) = (T1.C.T2.h1.r,h2), where T2 and h2 are all high. Then (T1.h1) and (T1.C.h1.r) are both trace-history pairs and (purge(T),purge(h)) = (purge(T1.C.purge(h1)).r), which is a trace-history pair by GCS noninterference.

Proof of (2): The proof will be by induction on the length of T. It is trivially true for empty T.

Suppose that the result is true for T of length n, and we want to prove it for T.C, where T is of length n, and C is a single procedure call. There are two cases to consider, the case in which C is low and the case in which it is high.

In the case in which C is low, suppose that (purge(T.C).h.r) is a trace-history pair, where (purge(T.C).h) is a trace-history pair. By the induction hypothesis there exists an h' such that purge(h') = h and (T.h') is a trace-history pair. Since (purge(T.C),h.r) = (purge(T.C,purge(h')).r) is a trace-history pair, it follows by GCS noninterference that (T.C,h'.r) is a trace-history pair.

In the case in which C is high, again suppose that (purge(T.C).h) it a trace-history pair. Since C is high,
we have that \((\text{purge}(T),h)\) is a trace-history pair. Thus by the induction hypothesis there exists an \(h'\) such that \(\text{purge}(h') = h\) and \((T,h')\) is a trace-history pair. If we can show that there exists an \(r\) such that \((T.C,h.r)\) is a trace-history pair, then we are done. But this follows directly from the fact that \(S\) is high-input total.

**Proof of 3.** Let \((T,h)\) be a trace-history pair. And suppose that \((\text{purge}(T).C,\text{purge}(h).r)\) is a trace-history pair for some low procedure call \(C\). Since \((T,h)\) is low-equivalent to \((\text{purge}(T),\text{purge}(h))\), we have that \((T.C,h.r)\) is also a trace-history pair. Thus by the induction hypothesis, we have that \((\text{purge}(T),h)\) is a trace-history pair. We conclude that \((T.C,h.r)\) is a trace-history pair. \(\square\)

We are interested in conditions under which weak and strong noninterference are composable. The following condition on the composition of two specifications gives us one.

**Definition 5.8** Let \(S1\) and \(S2\) be two multilevel specifications and let \(S\) be their composition. We say that the composition of \(S\) out of \(S1\) and \(S2\) has the lifting property, if whenever \((\text{purge}(T),h)\) is a composition of \((T1,h1)\) and \((T2,h2)\), then there are trace-history pairs \((T1',h1')\) and \((T2',h2')\) whose composition is \((T1,h1)\) and such that \(T1 = \text{purge}(T1')\) and \(h1 = \text{purge}(h1')\).

If \(S\) is the composition of two noninterfering specifications, and the composition has the lifting property, it is easy to see that \(S\) is weakly non-interfering. The fact that a composition of strongly noninterfering specifications with the lifting property is also strongly noninterfering is somewhat less obvious, and we prove it below.

**Proposition 5.3** Let \(S1\) and \(S2\) be two strongly noninterfering specifications, and let \(S\) be a composition of \(S1\) and \(S2\) with the lifting property. Then \(S\) is strongly noninterfering.

**Proof:** Since \(S\) is a composition with the lifting property, it is weakly interfering. Thus all we have to show is that, for every trace-history pair \((T,h)\) and low procedure call \(C\) such that \((\text{purge}(T).C,\text{purge}(h).r)\) is a trace-history pair, we have that \((T.C,h.r)\) is a trace-history pair.

Without loss of generality, we assume that \(C\) is from \(S1\). Since \((T,h)\) is in \(S\), we have that

1. By Lemma 4.1, the trace-history pair \((\text{purge}(T).C,\text{purge}(h).r)\) is a composition of \((T1.C.V1,h1.V2.r1)\) and \((T2.V2,h2.V1.r2)\), where \((\text{purge}(T),\text{purge}(h))\) is a composition of \((T1,h1)\) and \((T2,h2)\).

2. By the lifting condition, there exist \(T1', T2', h1',\) and \(h2'\), such that \(\text{purge}(T') = T1', \text{purge}(h') = h1'\), and \((T,h)\) is a composition of \((T1',h1')\) and \((T2',h2')\).

3. By Lemma 5.1, \(\text{purge}(T1'.C.V1) = T1.C.V1,\) \(\text{purge}(T2'.V2) = T2.V2,\) \(\text{purge}(h1'.V2.r1) = h1.V2.r1\) and \(\text{purge}(h2'.V1.r2) = h2.V1.r2\).

4. By Step 3 and repeated application of strong interference, we have \((T1'.C.V1,h1'.V2.r1)\) and \((T2'.V2,h2'.V1.r2)\) are trace-history pairs.

5. By Lemma 4.1 and Steps 2 and 4, \((T.C,h.r)\) is a composition of \((T1'.C.V1,h1'.V2.r1)\) and \((T2'.V2,h2'.V1.r2)\).

We conclude that \((T.C,h.r)\) is a trace-history pair. \(\square\)

We now consider how we would actually prove that a specification satisfies one of our versions of noninterference. One way of doing so would be by making use of the notion of equivalence. For example, in [13] McLean shows his specification of a multilevel stack is noninterfering by showing that, if the normal form traces are noninterfering (using his definition of noninterference), then all traces are noninterfering. It is then relatively straightforward to show that all normal form traces are noninterfering.

We could prove similar results for our nondeterministic versions of noninterference. Suppose that we had a trace specification in which no normal form traces contained values that returned procedure calls. If we were able to prove that, if all traces \(T.C\) where \(T\) is a normal form trace and \(C\) is a procedure call returning a value were noninterfering, then the specification was noninterfering, then the task of showing that all traces were noninterfering would be reduced to the task of showing that a set of traces with value histories of length one was noninterfering. This would be much easier than considering value histories of arbitrary length. It would also be interesting to see if there were any easily verifiable properties of a specification that would guarantee that this was true. In particular, the notion of equivalence seems promising, as long as we can prove that equivalence is preserved by a particular composition.

**6 Conclusion**

In this paper we have discussed the applicability of trace specification of procedure calls to the specification of compositional security properties. In particular, we have shown how composition of systems can be defined and how one can define a meaningful security property for such a specification, and we have discussed techniques for showing that security properties are preserved under composition.

However, much work still remains to be done. In particular, it would be helpful to gain more insight into what conditions on a composition make a property composable or not. The concept of equivalence, which has proven useful for developing compositions of specifications in this paper, and in proving security properties of deterministic specifications in McLean's work, may provide assistance here.
7 Acknowledgements
I am grateful to John McLean, Jim Gray, Ira Moskowitz, Paul Syverson, and Jeremy Jacob for their comments and advice.

References