Noninterference and the Composability of Security Properties

Daryl McCullough

Odyssey Research Associates, Ithaca, NY

In this paper, I discuss the problem of composability of multi-level security properties, particularly the noninterference property and some of its generalizations. Through examples I attempt to show that some of these security properties do not compose—it is possible to connect two systems, both of which are judged to be secure, such that the composite system is not secure. Although the examples are "cooked up" to make a point, there is nothing especially tricky done; I make sure that outputs from one system become inputs to the other machine at the same security level, and use a standard notion of parallel composition of systems (see [Hoare 85]).

The final property I introduce, which I call *restrictiveness* (formerly it was called "hook-up security"), is generally composable, so that two restrictive systems connected legally results in a new restrictive composite system. (For those interested in the proof, see [McC 88]). A new feature in the brief discussion of restrictiveness is a state-machine version of the property.

1 Why Care About Composability?

There are several reasons that a security engineer should care about having a definition of security that is composable:

- Few systems are entirely stand-alone; there are always incentives to hook small systems together into networks in order to share information or computational resources. In anticipation of future connections, the security engineer should make sure that whatever confidence he has in the security of his system carries over to the new composite system.
- Even for a single computer system, there are in general components which are partially independent, such as the disk drives, the CPU, terminals and printers. An accurate treatment of any system might require considering the interactions of several concurrent components.
- In the design of a large, complex system it may be easier to break up the system design into smaller subsystems and analyze the security of the components rather than try to prove the security of the system as a whole. For this "divide and conquer" approach is to be successful, one needs a criterion for the security of the pieces which is sufficient to guarantee the security of the whole.
- Through time-slicing, concurrency is simulated on a single processor, and so a useful model for a system is that of several user processes and the operating system running "semi-concurrently". From the viewpoint of this model, it is not enough to consider the security of the operating system alone; it is necessary to consider the interactions of the operating system with the other programs. Protection against "Trojan horse" programs can thus be enhanced by considering the security of the operating system in the context of being hooked-up to a collection of other, possibly malicious processes. (The idea connecting composability of security with protection against Trojan horses is due to John Millen of Mitre.)

Given that it is important to consider composability in the security of a system, the question becomes: what is a composable model of security?

I will distinguish between two kinds of insecure systems. The first kind contains some security loop-hole or trap door which allows the spy to bypass normal access controls and to directly receive classified data. The Bell-LaPadula security model[BLP 76] is intended to prevent this kind of insecurity; for a system to be secure in the sense of Bell-LaPadula, every possible sequence of system state transitions must result in a secure state; i.e., one in which no user has access to classified data unless he is officially authorized to have that access.

[1]This work was supported by the Air Force Systems Command at Rome Air Development Center under Contract No. F30602-85-C-0098

The second kind of insecure system is one which disallows *direct* access of data by unauthorized users, but nevertheless allows for *covert channels.* A covert channel is an indirect communication path between users. For the second kind of insecure system, it is often necessary for the spy to have a partner which is privileged to see classified data and can signal this information to the spy. The partner in high places can either be a human, (in which case it is unnecessary to communicate through the system at all; the two can pass notes in the cafeteria), or a *Trojan horse* program. An important thing to realize about the Trojan horse program is that it does not necessarily violate any rules of security; it may even be *proven* to be secure according to some formal model of security. However, if the system on which the program is running is insecure, it may be possible for the Trojan horse to communicate classified information to the spy through the side effects of perfectly normal innocuous operations. The prevention of low-level side effects of the behavior of high-level programs is the goal of various versions of noninterference requirements on systems, including the Goguen–Meseguer noninterference property[GoMes 84] and the Bell–LaPadula "star" property (which disallow the writing of low-level data by high-level programs.)

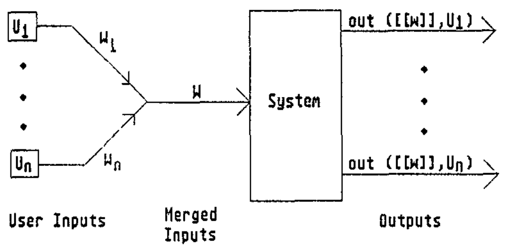

Figure 1: Goguen–Meseguer Machines

2 Goguen–Meseguer : MLS Noninterference

Goguen and Meseguer[GoMes 84] take the principal notion behind security to be that of *noninterference* rather than information flow. Noninterference can be used to give information flow restrictions; rather than saying that person A is not allowed to receive information from source $\mathcal{B}$, one can instead say that source $\mathcal{B}$ is not allowed, either directly or indirectly, to *interfere with* person A. The link between the two statements is the plausible assumption (which can be formalized and proved; this was done by Sutherland[Suth 86]) that A cannot learn anything about $\mathcal{B}$ unless $\mathcal{B}$ has some effect (or interferes with) something visible to A.

To make the notion of noninterference more precise, Goguen and Meseguer give an abstract model for a class of information processing systems and define noninterference for that class. Their original model was for state machines, and for general noninterference policies (and not simply for multilevel security). I will modify their treatment in two ways:

1. I will only consider MLS noninterference properties.
2. I will give the definition in terms of input and output sequences rather than state machines.

In their model, there is a collection of users, which I will call u_1, u_2, ... u_n. They each issue a sequence of commands w_1, w_2, ... w_n which are merged, forming the sequence w. The system computes a function of its input sequence for each user and the output is sent to the appropriate user. This setup is illustrated in figure 1.

Each user u_i has a security level assigned to him. If the system obeys the MLS noninterference policy, then for every pair of users u_i and u_j, if the level of u_i is not less than or equal to the level of u_j, then the inputs of u_i may not interfere with the outputs of user u_j.

Formally, letting $out([[w]], u_j)$ be the outputs to user u_j resulting from input sequence w, and letting $PG_{u_i}(w)$ be the result of purging from w all inputs from user u_i, the requirement of noninterference becomes, for all i, j, w:

$$level(u_i) \not\leq level(u_j) \rightarrow out([[w]], u_j) = out([[PG_{u_i}(w)]], u_j)$$

2.1 Assumptions Behind the Model

The model of information processing assumed by Goguen and Meseguer is not completely general. By assuming that the output is a function of the input sequence, they have restricted the set of systems to which their model applies in at least two ways:

1. They only consider deterministic systems. For nondeterministic systems, the output is not a function of the input sequence, since more than one output sequence can result from the same input sequence.
2. They only consider uninterruptable systems. This restriction is closely related to the first, because a system with interrupts will look nondeterministic when one looks at the observable behavior. For example: consider a system which allows the user to abort a computation that is taking too long. For such a system, the same input sequence, the start command followed by the abort command, can give rise to two different output sequences depending on whether the abort command comes before or after the calculation is completed. (If it comes in time, the system may respond with *Ok*. If it comes too late, the system may instead respond with *No processes running.*) Note that interrupts only lead to nondeterminism because time is not explicitly considered in the model.

The above assumptions are not necessary, since many perfectly reasonable systems do not meet them, but they have the very nice feature that if a system design is proved to be MLS noninterfering, and if the output function really captures everything about the system that is visible to the user, then any implementation will also be noninterfering, since *everything* visible to the user would already be decided at the design stage. If interrupts and nondeterminism were included, then there would be the possibility that the implementation could affect the precise way that the interrupts or the nondeterminism worked, thus possibly invalidating the proof of security.

2.2 How to Make a System Deterministic

When inputs are coming from a human being, it may be plausible to treat the system as *infinitely* fast; that is, the system can complete any calculation between two inputs from the user. However, when inputs come not from a human, but from another machine, this is no longer a reasonable assumption; the possibility arises that the system may be given inputs faster than they can be handled. Since I am ultimately going to be discussing the composition of several machines, it's necessary to give this matter a little thought. On first analysis, there seem to be three ways of handling the untimely arrival of inputs:

1. Assume that there is an unbounded input buffer. Then it would be possible for the system to leisurely take the inputs as it has time for them, and the result is the same as if the inputs had come more slowly. This solution has the disadvantage of being impossible to implement in this finite world. However, there may be situations in which a finite but very long buffer can be treated as if it were infinite, if it is known that there is a very small probability of the buffer becoming full.
2. Assume that there is a finite buffer, and that the process or person making the inputs will *block*; that is, patiently wait for the buffer to have space before making an input. The state of the process is unchanged by blocking, so that when it becomes unblocked, it cannot "know" how long it was blocked, or even whether it was ever blocked. This solution is actually implemented in such programming languages as Gypsy, and as long as the inputing process does not "remember" being blocked, the result will be the same as if the buffer were infinite from the point of view of that processes.
3. Assume that there is a finite buffer, and that if it is full, the next "send" to the buffer will cause that message or another to be dropped (it may or may not inform the sender of this fact).

2.3 The Problem with Blocking Inputs: Noncomposability of Noninterference

Although for some purposes blocking is an elegant solution to the problem of finite buffers, it can produce headaches for designers trying to make a secure system. The reason for this is that, by introducing blocking, a designer also introduces covert channels; ways to send information without sending messages. If a buffer is full, then it is possible to receive information from the buffer by *sending* to it; if the send is successful, then the sending process learns that someone has unblocked the buffer.

Now, it is possible to argue at this point that this is a pretty small amount of information; after all, if the buffer never becomes unblocked, then the sending process will never become active again in order to pass along this fact. If the buffer eventually does become unblocked, then the process does not learn anything, either, since it will not "remember" ever being blocked in the first place. The only information available to the process is this: if it is not blocked, it knows that it is not blocked.

This argument, however, is incorrect. In the context of several concurrently executing processes. It is possible to arrange two processes which only receive information through buffer blocking such that the resulting composite system al-

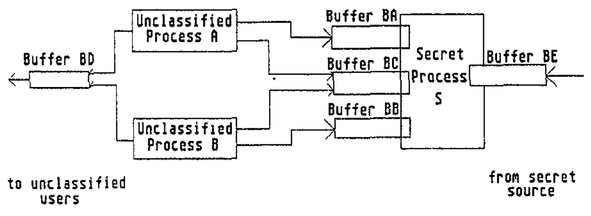

Figure 2: Using Buffer Blocking as a Channel

lows arbitrarily large amounts of information to leak. This possibility is illustrated in figure 2.

In this figure, there are three processes, $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{S}$, the first two being unclassified processes, and the third being a secret process. There is one externally visible output buffer, B_U, which can be read by unclassified users, and one externally visible input buffer, B_S, which receives a sequence of *zeros* and *ones* from some secret source.

The secret process has three input buffers: B_A, which process $\mathcal{A}$ may write to, B_B, which $\mathcal{B}$ may write to, and B_C, which both $\mathcal{A}$ and $\mathcal{B}$ may write to.

Considering process $\mathcal{S}$ together with its input buffers, one sees that it outputs no messages at all. Therefore, it trivially obeys the Goguen–Meseguer noninterference property. Likewise, since $\mathcal{A}$ and $\mathcal{B}$ have no high–level inputs, there cannot be any interference of low–level events. Thus, if all the buffers were infinite, this set–up would certainly be secure, since no information at all would leak from the secret input buffer to the unclassified input buffer. If buffers B_A and B_B are finite, though, things are much different.

I will look at the case in which buffers B_A and B_B each have length one, and I will assume the following behavior for the processes: Process $\mathcal{A}$ first sends a message to buffer B_A, (filling it) and then tries to send one more. After it sends the second message, it sends a *zero* to buffer B_U, and then sends to buffer B_C (informing the secret process that $\mathcal{A}$ has completed a cycle). Process $\mathcal{B}$ acts analogously, sending twice to buffer B_B and then sending a *one* to buffer B_U, and then to buffer B_C. $\mathcal{A}$ and $\mathcal{B}$ then cycle through the above actions over and over again.

Process $\mathcal{S}$ repeatedly receives from B_S, receives from B_C and then receives from either B_A or B_B depending on whether it receives a *zero* or a *one* from the secret buffer B_S.

When these processes are connected and allowed to run, the first thing that happens is that $\mathcal{A}$ and $\mathcal{B}$ fill up buffers B_A and B_B, respectively, while process $\mathcal{S}$ receives instructions from buffer B_S. Process $\mathcal{S}$ then unblocks one or the other of the buffers by receiving from it, the choice being determined by what it received from B_S. This releases the corresponding process to send to buffer B_S and then to B_C. Process $\mathcal{S}$ waits for something to appear in buffer B_C (indicating that process $\mathcal{A}$ or $\mathcal{B}$ has just completed a "send" to B_U. Then it starts over, with $\mathcal{S}$ receiving from B_S.

The net effect of all of this is to copy the sequence of *zeros* and *ones* from the secret buffer B_S into the unclassified buffer B_U, where they can be read by unclassified users, which is about as blatant a security violation as can be imagined.

The moral of this story is that, if you want to be able to follow information flow through a system by simply watching who sends to whom, then you cannot generally allow the blocking of inputs. Thus, the blocking of inputs is *not* a way to achieve the same level of security for the finite–buffer case Goguen–Meseguer machine as held in the case of unbounded buffers.

2.4 The Problem with Not Blocking Inputs: Nondeterminism

An alternative to blocking inputs when a buffer is filled is to respond with an error message *Busy* if a message arrives when the system is not ready to take it. As shown in figure 3, this leads to nondeterminism, exactly as the existence of interrupts did.

2.5 Parallel Composition Leads to Nondeterminism

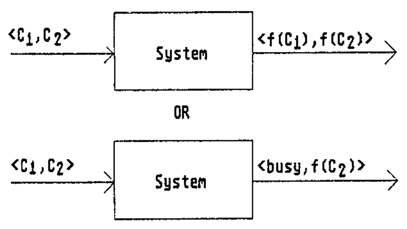

Figure 3: Not Blocking Inputs

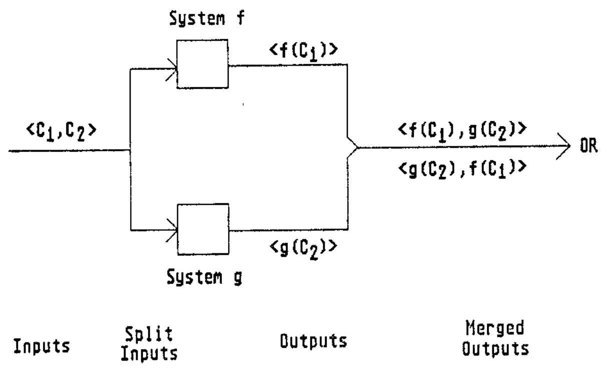

Figure 4: The Nondeterministic Composition of Deterministic Machines

Even if we assume infinite buffers, nondeterminism creeps in when we consider connecting several processes. This is illustrated in figure 4. Here, we see two processes f and g, each of which is deterministic. However, they are connected together in such a way that their outputs are merged. Two different output sequences of the resulting composite system are possible, depending on which process, f or g, finishes first. Thus the resulting system is nondeterministic (although determinism would be restored if the relative processing speeds of f and g were taken into account.)

2.6 When Noninterference is Composable

To preserve the MLS noninterference property when processes are composed it is necessary to require the following:

- All processes must communicate through unbounded buffers, since bounded buffers seem either to lead to covert channels not easily analyzed or to nondeterminism.
- There must be no merging of the outputs of processes; that is, two different processes may not both send to the same process or buffer, since this leads to nondeterminism, which cannot be handled by the Goguen and Meseguer definition of noninterference. This rules out the kind of uses of multiprocessing in which the results of many parallel computations are funneled into a centralized source, where they can be acted upon as they arrive.

If these restrictions are bearable, then Goguen and Meseguer can be used in composing systems. If they seem hard to live with (or if the assumption of infinite buffers seems unrealistic) then there is a motivation to look for a generalization of Goguen and Meseguer noninterference. As I mentioned earlier, the assumption of deterministic processes, which seems to be at the root of the problems, is not apparently relevent to security. I will turn next, then, to nondeterministic generalizations of noninterference.

3 Generalized Noninterference: Incorporating Nondeterminism

To simplify the discussion of noninterference, I will assume that there are only two levels: *high* and a lower level, *low*. The generalization to more than two levels is straight–forward.

If noninterference is generalized to incorporate nondeterminism and interrupts, then it will no longer be possible to talk about the output as a function of the input sequence, for two reasons:

1. There may be more than one output that may follow from a given input sequence.
2. The outputs may depend not only on the sequence of inputs, but also on the way the inputs are interleaved with the system's outputs.

For a nondeterministic system there is, in general, a set of possible future possibilities at any given time. To say that high-level users do not interfere with low-level users intuitively implies that:

- The input of a high-level signal may not alter the possible future sequences of low-level events.

Using $\alpha^\wedge\gamma$ to mean the sequence of events in α followed by the sequence in γ, I will say that γ is a *possible future* for α if $\alpha^\wedge\gamma$ is a possible history. If γ_1 is the sequence of low-level events occuring in some possible future of α, then γ_1 will be said to be a *possible low-level future* of α.

A system will have the generalized noninterference property if for every history (sequence of inputs and outputs) of the system α,

- If x is a high-level input, then the set of possible low-level futures of $\alpha^\wedge\langle x\rangle$ is equal to the set of possible low-level futures of α.

3.1 Generalized Noninterference is not in General Composable

Consider a system, called $\mathcal{A}$, which has the following set of traces: each trace starts with some number of high-level inputs or outputs followed by the low-level output *stop_count* followed by the low-level output *odd* (if there have been an odd number of high-level events prior to *stop_count*) or *even* (if there have been an even number of high-level events prior to *stop_count*). The high-level outputs and the output of *stop_count* leave via the right channel of the process, and the events *odd* and *even* leave via the left channel. The high-level outputs and the output of *stop_count* can happen at any time.

A typical event sequence of system $\mathcal{A}$ is portrayed in the top left corner of figure 5.

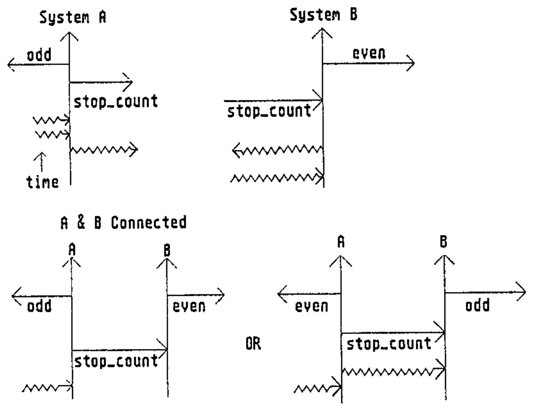

Figure 5: Noninterference is not Composable

System $\mathcal{A}$ actually obeys the generalized noninterference policy; regardless of high-level inputs, the possible low-level sequences are *stop_count* followed by *odd* or *stop_count* followed by *even*. A high-level input does not affect these possibilities, because it is always possible for such an input to be followed by a high-level output, and the pair would leave the low-level outputs unaffected.

System $\mathcal{B}$ behaves exactly like system $\mathcal{A}$, except that:

- its high-level outputs are out its left channel
- its *even* and *odd* outputs are out its right channel.
- *stop_count* is an input to its left channel, rather than an output.

A typical event sequence of system $\mathcal{B}$ is shown in the upper right corner of figure 5.

If systems $\mathcal{A}$ and $\mathcal{B}$ are connected, so that the left channel of $\mathcal{B}$ is connected to the right channel of $\mathcal{A}$ then we have the situation pictured in the bottom of figure 5. Now we see that the combined system no longer obeys the generalized noninterference policy: now, since the number of shared high-level signals is the same for $\mathcal{A}$ and $\mathcal{B}$, the fact that $\mathcal{A}$ says *odd* while $\mathcal{B}$ says *even* (or vice-verse) means that there has been at least one high-level input from outside. If all high-level inputs are deleted, then systems $\mathcal{A}$ and $\mathcal{B}$ will necessarily both say *even* or both say *odd.* Therefore, generalized noninterference is not composable for systems with feedback (two-way communication between processes).

3.2 When Generalized Noninterference is Composable

It turns out that this property is composable if there is no feedback between systems; that is, if no two systems are allowed to pass messages in both directions. An example of such a composition without two-way communication is a system connected up to an infinite buffer such that all its inputs come through the buffer. The communication between buffer and system is one-way; from the buffer to the system. Although the proof will not be given here, any collection of noninterfering processes with buffered inputs can be connected together to still have a composite system which has the noninterference property.

4 Deducibility Security

Deducibility security (or information flow security) was introduced by Sutherland[Suth 86]. It is more general than deterministic noninterference, in that deterministic, uninterruptable systems are noninterfering if and only if they are deducibility secure, but the definition of deducibility security does not require deterministic systems.

Informally, a system can be defined to be deducibility secure if it is impossible for a user, through observing events visible to him, to deduce anything about the sequence of inputs made by a second user unless the level of the second user is lower than the level of the first.

If inputs are never blocked, then it turns out that this definition is equivalent to a simpler definition: a system is deducibility secure if for every level l and every trace (or sequence of inputs and outputs possible for that system) there is a second trace with the same behavior visible to users of level l or less, but which has *no* inputs that are not less than or equal to level l. This is a kind of noninterference assertion—it is always possible to delete high-level inputs and leave low-level inputs and outputs alone.

4.1 When Deducibility Security is Composable

It is easily shown that a system which obeys generalized noninterference is also deducibility secure. The converse does not hold. Therefore, deducibility security is even less composable than generalized noninterference. It is not even composable for systems without feedback. However, if in addition to requiring that systems be deducibility secure, one also requires that there be no "unsolicited write-ups" then deducibility security becomes composable. An unsolicited write-up occurs whenever the system makes a high-level output at a time when there has been no high-level input requesting it. Deducibility security, strengthened with this additional restriction becomes a property which is sometimes called *strong noninterference.* Doug Weber at Odyssey has proved that this is a composable security property.

However, there are several undesirable features of strong noninterference:

- It forbids some systems that are obviously secure.

  To see this, consider a system which simply upgrades information; it takes in low-level signals and outputs high-level signals. It is manifestly secure; it can't possibly give away high-level information to the low-level user, since it never even *sees* any high-level information.

- It is not preserved by upgrading outputs.

  Intuitively, upgrading outputs (increasing their security level) on a secure system should leave the system secure; there is *less* information available to low-level user than before. However, such an upgrade can change a system which obeys strong noninterference into one which does not, since the transformation may produce new unsolicited high-level outputs.

- It permits systems which are intuitively insecure.

  Consider a system that behaves in the following way:

  - There is two possible low-level input commands; *begin_eavesdrop*, and *end_eavesdrop.*
  - If the low-level user issues *begin_eavesdrop*, and then at a later time issues *end_eavesdrop*, the system will respond by repeating to the low-level user the sequence of inputs made during the period between low-level commands, if any.
  - If no high-level inputs are made at all in the period between *begin_eavesdrop* and *end_eavesdrop*, then the system will send to the low-level user a fake response made up of randomly selected outputs.

The system described above is deducibility secure. A low level user can never deduce anything about the sequence of high-level inputs, since it is always possible that absolutely any sequence was made in a time that did not fall between *begin_eavesdrop* and *end_eavesdrop*. (Actually, a low-level user can deduce something; he can deduce that certain sequences of high-level inputs were definitely *not* made during the eavesdropping period. However, this deduction is allowed by the definition of deducibility security.)

5 Restrictiveness: A Composable Security Property

An important thing to notice about the failure of composability for generalized noninterference is that, although a system obeying the property insures that no *single* high-level input will affect the future low-level behavior, it does not guarantee that a *pair*, consisting of a high-level input followed immediately by a low-level input, will have the same effect on the low-level behavior as the low-level input alone. From figure 5, it is clear that system $\mathcal{B}$ does not insure this latter, stronger form of noninterference. For example, the pair consisting of a high-level input followed by *stop_count* does *not* have the same effect as *stop_count* alone.

The additional requirement can be intuitively understood as follows: Only some facts about the past of a system are relevent for the future low-level behavior of the system. These relevent facts can be thought of as defining the "low-level state" of the system. The requirement of noninterference is that a high-level input may not change the low-level state of the system. Therefore, the system should respond the same to a low-level input whether or not a high-level input was made immediately before. Systems which obey this property, called *restrictiveness* are said to be *restrictive*. Restrictiveness was described under the name of "hook-up security" in [McC 87], and is a composable security property.

5.1 A State Machine Characterization of Restrictiveness

A state machine is characterized by giving a set $\mathcal{E}$ of events, a set $\mathcal{I}$ of input events, a set of $\mathcal{O}$ of output events, a set $\mathcal{S}$ of states, an initial state S_0, and a set $\mathcal{T}$ of transitions of the form

$$S_1 \xrightarrow{\gamma} S_2$$

meaning that the machine may start in state S_1, engage in event sequence γ, and end up in state S_2. The set of traces produced by a state machine is the set of event sequences produced by the transitions starting in the initial state.

To formalize restrictiveness for state machines, I will once again consider only the case in which there are two levels: *low* and *high*. For a state machine to be restrictive with respect to a set of low-level events, it is sufficient that a state machine have the following statements hold:

- It is *input total*, meaning that inputs are never blocked, and so for every input signal and every state there must be a transition leading out of that state with that input signal.
- There is an equivalence relation $\equiv$ between states (indicating when two states correspond to the same low-level state) such that
  1. If e is a high-level input, and $S_1 \xrightarrow{\langle e \rangle} S_1'$ then $S_1 \equiv S_1'$.
     This says that high-level inputs may not affect the low-level part of the system state. This can be thought of as a kind of "write only up" policy.
  2. If $S_1 \equiv S_2$ and e is a low-level input, and $S_1 \xrightarrow{\langle e \rangle} S_1'$, then for some state S_2' such that $S_2' \equiv_l S_1'$, it must be that $S_2 \xrightarrow{\langle e \rangle} S_2'$.

     This requirement says that the low-level part of the final state following a low-level input depends only on the input and the low-level part of the state before the transition.
  3. If $S_1 \equiv S_2$ and γ is a high-level output sequence, and $S_1 \xrightarrow{\gamma} S_1'$, then for some state S_2' such that $S_2' \equiv S_1'$, and some high-level output sequence γ', it must be that $S_2 \xrightarrow{\gamma'} S_2'$.
  4. If $S_1 \equiv S_2$ and γ and δ are high-level output sequences, and e is a low-level output, and $S_1 \xrightarrow{\gamma^\wedge \langle e \rangle^\wedge \delta} S_1'$, then for some high-level output sequences γ' and δ' and some state S_2' such that $S_2' \equiv S_1'$, it must be that $S_2 \xrightarrow{\gamma'^\wedge \langle e \rangle^\wedge \delta'} S_2'$,
     These two rules state that if two states have the same low-level parts, then for any possible output sequence leading from one state there must be an equivalent output sequence leading from the other state, and the resulting final states must be equivalent. This can be thought of as a "read only down" policy; the possible output sequences for low-level events are completely determined by the low-level equivalence class.

The rules above are equivalent to the trace definition of restrictiveness given in [McC 87] in that:

* For any set of traces which is restrictive, there

is some restrictive state machine which produces exactly that set of traces.

* For any state machine which is restrictive, the set of traces produced by it is also restrictive.

There are, however, some state machines which are definitely *not* restrictive, which nevertheless give rise to a restrictive set of traces.

5.2 The Composability of Restrictiveness

To see that the security property of state machines is composable, one needs to prove that if machine $\mathcal{A}$ and machine $\mathcal{B}$ both obey state machine restrictiveness, then the composite machine formed by connecting them obeys state machine restrictiveness. For the connection to make sense, the two machines must be compatible, in the sense that output events for one machine are input events for the other, and that the two machines agree on which events are considered low-level.

The states of the composite machine are pairs of states of $\mathcal{A}$ with states of $\mathcal{B}$.

The transitions of the composite machine are given by: $\langle A_1, B_1 \rangle \xrightarrow{e} \langle A_2, B_2 \rangle$ only if one of the following cases holds:

1. $A_1 \xrightarrow{e} A_2$ is a legal transition for machine $\mathcal{A}$, and e is not an event of $\mathcal{B}$, and $B_1 = B_2$
2. $B_1 \xrightarrow{e} B_2$ is a legal transition for machine $\mathcal{B}$, and e is not an event of $\mathcal{A}$, and $A_1 = A_2$
3. $A_1 \xrightarrow{e} A_2$ and $B_1 \xrightarrow{e} B_2$ are legal transitions for $\mathcal{A}$ and $\mathcal{B}$, respectively, and e is an input for one machine and an output for the other.

The equivalence relation on the states of the composite machine is obtained from the equivalence relation for the components as follows: $\langle A_1, B_1 \rangle$ is equivalent to $\langle A_2, B_2 \rangle$ if and only if A_1 is equivalent to A_2 and B_1 is equivalent to B_2.

It is straight–forward to prove that for such a composite machine, it will obey state machine restrictiveness if the component machines do. (This is in contrast to the original definition of restrictiveness, which was defined in terms of possible histories instead of state transitions.)

It is a little more complicated to show that restrictive state machines also obey the generalized noninterference property, but they do, and so they cannot be used for even half–bit covert channels.

5.3 What's Good about Restrictiveness?

Restrictiveness is a useful property because:

* Any legal (connecting outputs to inputs of the same level) composition of restrictive systems is restrictive.
* If a system is restrictive, and some output events are hidden (made to become unobservable internal events) then the resulting system is restrictive. (This means that the property of being restrictive is preserved by any modification of the system which changes only the behavior of internal events.)
* Restrictiveness applies to nondeterministic, as well as deterministic systems.

6 Summary : How do the properties rate?

1. Goguen and Meseguer's noninterference property
   On the positive side:
   * It is intuitively appealing.
   * Every implementation of a secure system is secure.

   On the negative side:
   * It is only composable for forward branching architectures (a process may not receive inputs from two different sources).
   * It is only composable if unbounded buffers are assumed. (As a matter of fact, for deterministic systems with unbounded buffers, it can be shown that a system obeys the MLS noninterference property if and only if it is restrictive.)
   * It applies only to deterministic systems.
   * It applies only to uninterruptable systems.
   * Its connection with information flow is not explicit.
2. Generalized Noninterference
   On the positive side:
   * It applies to nondeterministic and interruptable systems.
   * It is a straight–forward generalization of Goguen–Meseguer.
   * It is somewhat composable.

   On the negative side:
   * It is only composable for networks with no feedback (no two–way communication between processes).
   * It is not preserved by implementation.
3. Sutherland's Deducibility Security
   On the positive side:
   * Its connection with information flow is explicit.
   * It can be applied to all systems.

* It is composable, under the additional assumption that unsolicited write-ups are disallowed.

On the negative side:

* It permits intuitively insecure systems.
* With the no write-up condition, it disallows some systems which are manifestly secure.
* With the no write-up condition, it is not preserved by upgrading outputs.
* It is not preserved by implementation.

4. Restrictiveness

On the positive side:

* It is composable for all network architectures.
* It applies to all systems; nondeterministic or interruptable.
* Restrictiveness is preserved by hiding outputs.
* Because restrictiveness is composable, one can make a restrictive large system is by composing restrictive smaller systems.

On the negative side:

* It may be difficult to prove in some circumstances.
* It is not preserved by implementation. (If one makes a system more deterministic

may invalidate the security proof, since it may produce new correlations between high-leve inputs and low-level events. This problem is inherent in any definition of security for nondeterministic systems.)

References

[Hoare 85] C.A.R. Hoare, *Communicating Sequential Processes*, (Prentice-Hall, London, 1985)

[McC 88] Daryl McCullough, *The Theory of Security in Ulysses*, (Technical Report, Odyssey Research Associates, Ithaca, NY, 1988)

[BLP 76] Bell, D.E. and LaPadula, L.J., *Secure Computer System: Unified Exposition and Multics Interpretation* (Technical Report no. ESD-TR-75-306, Electronics Systems Division, AFSC, Hanscom AF Base, Bedford MA, 1976)

[GoMes 84] Goguen, J.A. and Meseguer, J. *Unwinding and Inference Control* (Proceedings of the 1984 Symposium on Security and Privacy)

[Suth 86] Sutherland, D. *A Model of Information* (Proceedings of the 9th National Computer Security Conference, 1986)

[McC 87] McCullough, Daryl *Specifications for Multi-Level Security and a Hook-Up Property* (Procedings of the 1987 Symposium on Security and Privacy)