A Framework for Composition

Todd Fine
Secure Computing Corporation
2675 Long Lake Road,
Roseville, Minnesota 55113-2536

Email: *fine@sctc.com*

Abstract

Analysis of complex systems requires the use of a "divide-and-conquer" approach to specification and verification. Existing theories for specification composition provide a starting point for a framework for such an approach. This paper describes a new framework that is a hybrid of two existing frameworks, explains the advantages of the new framework, and illustrates its use through a simple example.[1]

1 Introduction

In this paper, we describe a variation of Lamport's TLA specification language[1, 2] and provide a framework for composition of specifications based on the work of Abadi and Lamport[1, 2] and Shankar[6]. Composition is a technique for constructing more complex specifications by building upon simpler specifications. Viewed from the other direction, the composition framework allows the specification and verification of a complex system to be decomposed into the specification and verification of simpler components. Benefits of this approach to assurance are similar to those realized when using a modular approach to software development. In particular, complex reasoning about an overall system can be reduced to simpler reasoning about a collection of components and reusable system components can be defined. After describing the framework, we provide an example of the use of the framework to specify and verify a simple example.

The framework and example have been formalized in the PVS specification language and the PVS prover has been used to prove all of the stated theorems. The PVS representation of the framework is generic and can be used to specify and verify other systems as well as the example provided here.

The organization of this document is as follows:

- Sections 2- 8 define a framework for specifying, composing, and verifying system components.
- Section 9 provides an example of the use of the framework to specify and verify a simple system. The example given is a much simplified version of a secure computing system.
- Section 10 summarizes the material presented here and contains some concluding remarks.

2 Behaviors

Our work is based on Lamport's TLA specification language. Advantages of the TLA specification language include:

- It is a simple language with clearly defined semantics.
- It is state based. Although event based specification languages are more appropriate for certain classes of problems, we have found state based languages more appropriate for our application area (analysis of secure systems). Specifying a component involves simply defining the set of possible states and identifying the allowable state transitions.
- It allows temporal properties to be specified and verified. This allows TLA to be used in a manner analogous to how event based languages are often used.
- There are existing theories of refinement and composition for TLA.

In the previous section, we stated that we are actually using a variant of TLA. By this, we mean that we have embedded a TLA-like specification language in PVS as a theory.

[1] This work was supported in part by the Maryland Procurement Office, contract MDA904-93-C-4209.

This allows us to write TLA-like specifications that can be processed using PVS. One of the major differences between TLA and our embedding is that PVS is a typed language while TLA is not. In addition, we have added some structure to the specifications beyond that required by TLA.

The basic construct in TLA is a *behavior*. A behavior consists of an infinite sequence of states $st_0, st_1, st_2, \ldots$ and an infinite sequence of agents $ag_0, ag_1, \ldots$.[2] The sequence of states represents snapshots of the system state as time progresses. The sequence of agents indicates the entity responsible for a given state transition. We define the type *trace_t* to denote a record containing the following fields:

- *sts* — denotes the sequence of states; *sts(i)* is the i^{th} state
- *ags* — denotes the sequence of agents; *ags(i)* is the agent causing the transition from the i^{th} state to the *i+1*th state

In TLA, a state represents the state of the "entire" universe at a given point in time. Generally, only a small subset of the state is relevant to a given specification. We refer to the relevant portion of the state as the *view* for that specification. Each view is required to be an equivalence relation. We use *VIEWS[X]* to denote the set of all equivalence relations on elements of type *X*.

A *behavior predicate* is an assertion about a behavior. We represent each predicate by the set of behaviors satisfying the predicate; a predicate p holds in a behavior beh when $beh \in p$.

An alternative way to view a behavior is as a sequence of *transitions*. By a transition, we mean a triple (st_1, st_2, ag) denoting that the entity identified by ag can cause the state of the system to change from st_1 to st_2. Given a sequence of transitions $tran_0, tran_1, tran_2, \ldots$ such that the final state for a given transition is the initial state for the next transition (in other words, $tran_i.st_2 = tran_{i+1}.st_1$), the sequence of transitions describes a unique behavior having:

- $sts(i) = tran_i.st_1$
- $ags(i) = tran_i.ag$

In our framework, we define the behavior type to be parameterized by both state and agent types. Instantiating the type with the state type representing the entire system state provides a behavior type appropriate for describing properties of the entire system. Using a state type representing the portion of the system state relevant to a given component provides a behavior type appropriate for describing properties of the component. Since "system" and "component" are interchangeable from the standpoint of the framework, we often use the terms interchangeably in the following.

[2] Some variants of TLA, such as that in reference [1], ignore agents and define behaviors in terms of only states.

3 Components

Abadi and Lamport usually specify components in the following normal form:

$$Init \land \Box N \land F$$

where:

- $Init$ is a state predicate characterizing the initial state,
- N is a disjunction of action predicates characterizing valid transitions (including a no-op step to allow "stuttering"),
- $\Box N$ means predicate N holds for all time, and
- F is a fairness condition that is the conjunction of "weak" and "strong" fairness conditions on steps comprising N.

This additional structure on specifications typically provides a more convenient method for describing a system component. Rather than specifying each of the possible sequences of transitions (behaviors) of the component, one merely specifies the set of initial states and the set of individual transitions associated with the component. The set of behaviors associated with a component can be derived from $Init$, N, and F.

In this paper, we choose to place even more structure on specifications. First, we ignore liveness conditions. Thus, the normal form for a component specification is $Init \land \Box N$. Furthermore, we introduce *cags* to denote the set of agents associated with a component. The processing done by the component is represented by transitions by these agents. The set of transitions by other agents that the specification of the component allows places restrictions on the processing performed by other components. In other words, these transitions define the component's environmental assumptions. Following the approach in [6], we use *guar* to denote the transitions in N that are caused by the component's agents and *rely* to denote the transitions in N that are caused by environment agents. In both cases, the transitions are state-state-agent triples indicating a permitted move from a starting state to a final state caused by an agent. A component must have a *view* specified which represents the portion of the state visible to the component. Finally, a component must have a function, *hidd*, specified which indicates the portions of the component's view that are inaccessible to each agent. One can view $hidd(ag)$ as implicitly specifying ag's interface to the component. The value returned by $hidd(ag)$ is a set of (st_1, st_2) pairs specifying parts of the state that are hidden from ag. One interpretation of this set of pairs associated with ag is as an equivalence relation specifying the component's view of the state minus ag's view. However, we allow for the possibility that the set of pairs is not an equivalence relation for generality (see Section 7 where we discuss the value of defining $hidd$ in terms of $rely$).

In summary, we define a component to be a structure having fields *init*, *guar*, *rely*, *cags*, *view*, and *hidd*. We require the following relationships to hold between the various fields:

- *init* is a set of states, *cags* is a set of agents, *guar* and *rely* are sets of transitions, *view* is an equivalence relation on states, and *hidd* is a function mapping agents to sets of state pairs.

- *init* and *cags* are non-empty.

  If *init* or *cags* were empty, then the component could never really do anything since it would either have no valid starting state or have no agents.

- The agent for each transition in *guar* is an element of *cags*.

- The agent for each transition in *rely* is not an element of *cags*.

- *guar* contains all *stuttering steps* with respect to *view* caused by agents in *cags*.

  A stuttering step with respect to an equivalence relation is a step in which the initial and final states for the transition are equivalent with respect to the relation.

  The stuttering steps serve as "place holders" for later refinements of the specification. For example, a refinement might involve dividing a single transition into a sequence of transitions that manipulate low-level portions of the state introduced as part of the refinement. In the original specification, each but the last of the sequence of transitions appears as a stuttering step (no-op) since the low-level portions of the state are not visible.

- *rely* contains all stuttering steps with respect to *view* caused by agents not in *cags*.

  These stuttering steps serve as "place holders" for refinements in the specification of the environment.

- For each ag, *view* is a subset of *hidd(ag)*. In other words, any two states for which the visible portions of the state are the same also have the portions of the state other than ag's interface with the component the same.

  The motivation for this requirement is that the data in the component that is not visible through an interface should be a subset of the entire collection of data visible to a component.

- *guar*, *view*, and *hidd(ag)* (for each ag) are well-defined with respect to *view*.

  By a set of transitions being well-defined with respect to an equivalence relation, we mean that for any two transitions that are equivalent, one transition is in the set if and only if the other is, too. By transitions being equivalent, we simply mean that the starting states are equivalent and the final states are equivalent.

  If *guar* were not well-defined with respect to *view*, then whether a transition is permitted or not would depend on more than *view* and *view* would not really capture the portion of the state visible to the component. Analagous statements can be made for $rely$ and $hidd(ag)$.

- The starting and final states for any transition in *rely* with agent ag form one of the pairs in *hidd(ag)*.

  If the starting and final states did not form one of the pairs in *hidd(ag)*, then the transition would change data outside ag's interface to the component. This would violate the implicit assumption that agents only interact with a component through their interface to the component.

In some versions of TLA, transitions are defined to be simply pairs of starting and final states. We chose to include an agent as part of a transition to record the entity responsible for a transition. Our application area is usually computer security. From a security standpoint, who performed an activity is often just as important as how the activity affects the system. When defining a component of a system, we use the agents to distinguish transitions by the component from transitions by the component's environment. In addition, we make use of agents when stating assumptions concerning how the component's environment interacts with the component. For example, certain agents might be able to access parts of a component's state while other agents might be prohibited.

For examples of how the $init$, $cags$, $guar$, $rely$, $view$, and $hidd$ can be defined for a component, see sections 9.2 and 9.3.

Just as the set of behaviors associated with $Init \wedge \Box N \wedge F$ can be derived from $Init$, N, and F, the set of behaviors associated with a component cmp can be derived from $init(cmp)$, $guar(cmp)$, and $rely(cmp)$. This set consists of all behaviors starting in a state belonging to $init(cmp)$ and containing only transitions permitted by *cmp*'s *rely* or *guar*. A system component cmp is said to satisfy a behavior predicate if each element of the set of behaviors modeling the system component satisfies the behavior predicate.

4 State and Agent Translation

It is typically the case that different components have different states and agents. This results in the properties defined for the components being type incompatible. We address this using translator functions that map elements of one type to another type. A translator must map each source element to a non-empty set of target elements in such a way that no two sets of target elements overlap. Furthermore, each element of the target type must be mapped to by some element of the source type. We use *translator_t[X,Y]* as the type denoting translators from type *X* to type *Y*. Given a set s and a translator t, we use *tmap(t,s)* to denote the set of elements to which t maps some element of s. In other words, *tmap* "maps" the translation t across the set s.

We allow the translators to return a set of values rather than a single value to address different levels of abstraction. For example, a state might be mapped to a more detailed representation in which some components are unconstrained by the components of the more abstract state. Then, multiple more detailed states might correspond to each of the more abstract states. With regard to agents, what appears to be a single agent at a certain level of abstraction might be seen to be multiple agents at a lower level of abstraction. For example, the more abstract model might view agents as being processes while a more detailed model might view agents as being threads executing within the processes.

We use the expression $vmap(t, v)$ to denote the equivalence relation on the target elements to which t maps an equivalence relation v. In other words, two elements y_1 and y_2 are related in the resulting equivalence relation exactly when there exist x_1 and x_2 such that:

- t maps x_1 and x_2 to, respectively, y_1 and y_2, and
- x_1 and x_2 are related by v

We use the expression $tr\_ac(ap, xt, yt)$ to denote the set of transitions to which xt and yt map a set of transitions ap. The result is a set of transitions with states and agents of the types mapped to by xt and yt. More specifically, a transition (x_1, x_2, y) is an element of the result exactly when there exists a_1, a_2, and b such that:

- xt maps a_1 and a_2 to, respectively, x_1 and x_2,
- yt maps b to y, and
- (a_1, a_2, b) is a transition in ap

A component can be translated to another state type and agent type by using translation functions. The translation is straightforward using *tmap*, *vmap*, and *tr_ac*. We use $tr\_cmp(cmp_1, xt, yt)$ to denote the translation of cmp_1 using xt and yt. It can be proven that any translation of a component satisfies the requirements on components defined in Section 3.

The following theorem can be proven:

> *Suppose cmp_1 and cmp_2 are components such that any valid initial state for cmp_1 is a valid initial state for cmp_2 and any transition permitted for cmp_1 is also permitted for cmp_2. Then, any property satisfied by cmp_2 is also satisfied by cmp_1.*

The proof is straightforward since predicate satisfaction is defined in terms of set inclusion and the conditions on cmp_1 and cmp_2 ensure that the behaviors of cmp_1 are a subset of the behaviors of cmp_2. This theorem is the key step in the proof of the composition theorem which is stated in Section 7.

5 State and Action Predicates

In general, we attempt to perform analysis in terms of *state predicates* and *action predicates* and use functions defined below to translate the analyzed predicates into behavior predicates.

A state predicate is an assertion about a state. We represent each predicate by the set of states satisfying the predicate. The type $STATE\_PRED$ denotes the set of all state predicates. We use $init\_satisfies(cmp, sp)$ to denote that sp holds in each of cmp's initial states.

An action predicate is an assertion about state transitions. We represent each predicate by the set of triples (st_1, st_2, ag) satisfying the predicate. Intuitively, the meaning of (st_1, st_2, ag) belonging to the set representing an action predicate is that the action predicate allows an action by ag to cause a state transition from st_1 to st_2. The type $ACTION\_PRED[ST, AG]$ denotes the set of all action predicates. We use $steps\_satisfy(cmp, ap)$ to denote that each transition ($guar$ and $rely$) allowed by cmp satisfies ap. We define:

- $stbp(sp)$ to denote the behavior predicate representing that state predicate sp holds in the initial state
- $atbp(ap)$ to denote the behavior predicate representing that action predicate ap is satisfied by each transition

Given a behavior predicate p, we use $always(p)$ to denote the behavior predicate representing that p "always holds". The formal definition is that $always(p)$ contains a behavior t if each "tail" of t satisfies p. A tail of t is any behavior resulting from the removal of a finite number of steps from the beginning of t.[3] For convenience, we define:

- $alwayss(sp)$ to denote $always(stbp(sp))$
- $alwaysa(ap)$ to denote $always(atbp(ap))$

This allows us to talk about state and action predicates "always holding" just as we talk about behavior predicates "always holding". Similarly, the standard logical operators can be defined on the various types of predicates. For example, $aandas(ap, sp)$ can be defined as the action predicate representing that ap holds for a transition and sp holds for the starting state of the transition.

We say that a state predicate is *stable* if whenever it holds in a given state, it holds in any state reachable from that state by a transition.

It is trivial to show that:

> *If sp holds initially and is stable, then sp always holds.*

This theorem captures the standard approach for analyzing systems with respect to safety properties. First, each potential initial state is shown to satisfy the property. Next, each transition that can occur is shown to maintain the satisfaction of the property. Then, induction allows one to conclude that the property holds in every reachable state.

6 Composition

The approach we use for combining specifications is a hybrid of the approaches used by Abada-Lamport and Shankar. In the Abadi-Lamport work, components are simply properties with the normal form $Init \wedge \Box N$. Composition is defined to be simply conjunction; the composition of $Init_1 \wedge \Box N_1$ with $Init_2 \wedge \Box N_2$ is $(Init_1 \wedge Init_2) \wedge \Box(N_1 \wedge N_2)$. In the Shankar approach, components are specified in terms of a tuple $(init, guar, rely)$ and composition is defined as:

$$(init_1 \wedge init_2,\\ (guar_1 \wedge rely_2) \vee (guar_2 \wedge rely_1),\\ rely_1 \wedge rely_2)$$

N in the Abadi-Lamport approach corresponds to $guar \vee rely$ in the Shankar approach. Thus, $N_1 \wedge N_2$ corresponds to:

$$(guar_1 \wedge rely_2) \vee (guar_2 \wedge rely_1) \vee (rely_1 \wedge rely_2) \vee (guar_1 \wedge guar_2)$$

The first two terms correspond to the Shankar's definition of $guar$ for the composite while the third term corresponds to Shankar's definition of $rely$ for the composite. So, other than the last term, both definitions are essentially the same. Typically, the steps by each component are disjoint so the last term does not contribute anything. In these cases, the two definitions are essentially the same. Then, the behaviors of the composite system are interleavings of behaviors of the individual component systems.

Our definition of composition is similar but slightly different. Reasons for the differences include:

- The use of a typed specification language requires that translator functions be used to make components type compatible before conjoining them.
- We have made $view$, $cags$, and $hidd$ explicit parts of the definition of a component, so we need to define $view$, $cags$, and $hidd$ for the composite system in terms of the component systems.
- In the definition of $guar$ for the composite, we replace the occurrences of $rely_i$ with $hidd_i$. Note that $hidd_i$ actually has a slightly different type than $rely_i$, so we need to do a type conversion before intersecting a $guar$ set with a $hidd$ function.

  To ensure the composition of two components is consistent, it is necessary to check that each component satisfies the assumptions the other component makes about its environment. The expression $guar_a \cap rely_b$[4] denotes the set of transitions that agents of component cmp_a can make that also satisfy the environmental assumptions of component cmp_b. Using $guar_a \cap rely_b$ as the basis for the definition of composition ensures the result is consistent by eliminating transitions that violate environmental assumptions of the other component.

  By replacing $rely_b$ with $hidd_b$, we simply remove transitions of cmp_a that modify the non-interface data of cmp_b. Our intuition is that these transitions are acceptable by cmp_a only because the hidden data of cmp_b is irrelevant to cmp_a. Removing transitions that modify cmp_b's hidden data simply makes explicit the implicit requirement that only cmp_b can modify its internal data.
- Unlike the Shankar definition, we retain the $guar_1 \cap guar_2$ term in the definition of composition.

  The main reason for doing so is to ensure the definition of composition is idempotent (in other words, the composition of a component cmp with itself is

[3] An analagous definition can be given for "eventually holds". Rather than requiring the property hold for every tail, it requires that the property hold for at least one tail.

[4] Note that we use "$\cap$" and "$\wedge$" interchangeably. Similarly, we use "$\cup$" and "$\vee$" interchangeably.

simply cmp). When both components are the same, the $guar_a \cap hidd_b$ term is simply $guar_a \cap hidd_a$. Since there is typically at least one transition in $guar$ that modifies the private data, $guar_a \cap hidd_a$ is typically a strictly smaller set of transitions than $guar_a$. However, $guar_a \cap guar_b$ is $guar_a$ when the two components are the same. Thus, the inclusion of the last term ensures idempotency.

We define the expression:

$$compose(cmp_1, cmp_2, sttr_1, sttr_2, agtr_1, agtr_2)$$

to denote the composition of components cmp_1 and cmp_2, where:

- $sttr_1$ and $sttr_2$ define translators from the state types for the individual components to a new state type appropriate for the composite system, and
- $agtr_1$ and $agtr_2$ define translators from the agent types for the individual components to a new agent type appropriate for the composite system.

Note that the state and agent types for the resulting system are typically different than those for the original components. For example, the resulting state type would typically be a "merge" of the individual state types. For simplicity, we ignore the type translation functions in the following definitions. Technically, each component must be translated (using *tr_cmp*) to common state and agent types before composing. For example, reference to the intersection of $init_1$ and $init_2$ in the following should be read as the intersection of $sttr_1(init_1)$ and $sttr_2(init_2)$.
We restrict the domain of *compose* as follows:

- cmp_1 and cmp_2 must be components as defined in Section 3.
- $sttr_1$, $sttr_2$, $agtr_1$, and $agtr_2$ must be translators as defined in Section 4.
- The intersection of $init_1$ and $init_2$ must be nonempty.

The function:

$$composable(cmp_1, cmp_2, sttr_1, sttr_2, agtr_1, agtr_2)$$

denotes that the last condition (that the intersection of $init_1$ and $init_2$ is nonempty) holds. The first two conditions above are implicit in the following. When the framework is used it is necessary to ensure that the restrictions on components and translators are satisfied by each of the terms.
Although not restrictions on the domain of *compose*, the following conditions are required by the theorem in Section 7.

- Suppose a transition is in $guar_1$, is by ag, has starting and final states that are related by $hidd_2(ag)$, and is not in $guar_2$. Then, the transition must be an element of $rely_2$. Although not quite type correct, this requirement can be loosely formalized as:

  $$(guar_1 \cap hidd_2) \setminus guar_2 \subseteq rely_2$$

  In other words, transitions by the environment of a component that do not modify the internal data for the component are allowed by the component.
- The analogous requirement with the roles of the two components reversed must hold.

Similarly, the following conditions are required by the theorem in Section 8 even though they are not included in the definition of the domain of *compose*. The intent of these conditions is to ensure that the intersection of one component's $guar$ with the other component's $hidd$ does not remove any interesting transitions. When these conditions hold, the only transitions that are removed are ones that are equivalent (and hence "redundant") to the transitions that really define the component.

- Given any transition (st_1, st_2, ag) in $guar_1$, there exists st_3 and st_4 such that:
  - st_1 is equivalent to st_3 with respect to $view_1$,
  - st_2 is equivalent to st_4 with respect to $view_1$, and
  - either:
    - (st_3, st_4, ag) is in $guar_2$, or
    - (st_3, st_4) is in $hidd_2(ag)$

  An analogous requirement must hold with the roles of components 1 and 2 reversed.

  Intuitively, this requires that any transition allowed for a component is equivalent to some other transition which is either allowed for the other component or does not modify the other component's internal data.

The above conditions can be simplified when $cags_1$ and $cags_2$ are disjoint. For example, the first condition reduces to requiring that for each (st_1, st_2, ag) in $guar_1$, there exists st_3 and st_4 so that:

- st_1 is equivalent to st_3 with respect to $view_1$
- st_2 is equivalent to st_4 with respect to $view_1$
- (st_3, st_4) is in $hidd_2(ag)$

Intuitively, this requires the effects of the transition visible to cmp_1 to be independent of data internal to cmp_2. Although not a restriction on the definition of the domain of $compose$, this condition is a reasonable requirement to place on the components comprising a system.
The result of the composition is defined to be a component for which:

- The state type is a (potentially new) type to which the original state types are mapped using $sttr_1$ and $sttr_2$.
- The agent type is a (potentially new) type to which the original agent types are mapped using $agtr_1$ and $agtr_2$.
- The set of allowable initial states for the composite is the intersection, $init_1 \cap init_2$, of the sets of initial states for the individual components.
- The set of transitions that the composite can make consists of transitions such that:
  - the transition belongs to *guar* for one component and respects *hidd* of the other component, or
  - the transition belongs to *guar* for both components

  In other words, $guar$ for the composite is:

  $(guar_1 \cap hidd_2) \cup (guar_2 \cap hidd_1) \cup (guar_1 \cap guar_2)$
- The environment transitions allowed by the composite consist of transitions that each component allows of its environment. In other words, $rely$ for the composite is $rely_1 \cap rely_2$.
- The agents for the composite consists of the union, $cags_1 \cup cags_2$, of the agents for the individual components.
- Two states appear the same to the composite only if they appear the same to both components. In other words, $view$ for the composite is $view_1 \cap view_2$.
- The portions of the state that are internal to the composite consist of portions of the state that are internal to at least one of the components. For the internal data in two states of the composite to be the same, the internal data for both components must be the same in the two states. In other words, $hidd(ag)$ for the composite is $hidd_1(ag) \cap hidd_2(ag)$.

It is straightforward to prove that the result of the composition is itself a component (in the sense defined in Section 3), and the given definition of composition is idempotent, commutative, and associative.

7 Composition Theorem

The composition theorem is:

> *Suppose two components are composable and each respects the environment assumptions of the other. Then any property that is satisfied by at least one of the components is satisfied by the composition of the two components.*

By a component satisfying the environment assumptions of another component, we mean the following conditions from Section 6:

- $(guar_1 \cap hidd_2) \setminus guar_2 \subseteq rely_2$
- $(guar_2 \cap hidd_1) \setminus guar_1 \subseteq rely_1$

The key to the proof is that whenever the components satisfy the hypotheses of the theorem, composition is defined such that the behaviors allowed for the composite are a subset of the behaviors allowed for each component. Since the composite cannot exhibit behavior beyond that exhibited by the components, any property satisfied by the components is also satisfied by the composite (see the theorem in Section 2).
The definition of composition provided here reduces to the Shankar definition if $hidd_i$ is chosen to be $rely_i$. Then, $(guar_1 \cap hidd_2) \setminus guar_2$ is $(guar_1 \cap rely_2) \setminus guar_2$ which is clearly a subset of $rely_2$. This is an advantage of Shankar's approach; the first two conditions above can be trivially proved. The danger of this approach is that since there are few proof obligations on composition, there is a greater likelihood that the resulting composition might be inconsistent. For example, if $rely_2$ does not contain any non-stuttering transitions in common with $guar_1$, then the condition holds but the composition is inconsistent; it is supposed to represent the two components working together but actually does not allow the first component to perform any meaningful transitions.
Another approach would be to define $hidd_i(ag)$ to be the equivalence relation identifying the portions of cmp_i's state that are hidden from ag. Then, the above conditions require proving that changes one component makes to the interface data of a second component are consistent with the assumptions the second component makes about its environment. This reduces the concern about inconsistencies going undetected. However, there is still a concern if the interface is identified incorrectly. For example, suppose that every non-stuttering element of $guar_1$ modifies a component of the state identified by $hidd_2$ as being internal. Then, the composition is once again inconsistent in that non-stuttering steps of cmp_1 are prohibited. In summary, this approach addresses the issue of inconsistencies between how the interface data is manipulated and how it is assumed to be

manipulated, but does not ensure that components do not manipulate internal data of other components.

The issue of the consistency of the composition is addressed next.

8 Correctness of Definition

It is interesting to note that the composition theorem would hold for many other definitions of composition. In particular, the theorem holds for any definition of composition such that the set of behaviors for the composite is a subset of the intersection of the behaviors for the components. Consequently, the composition theorem by itself is somewhat meaningless. To be of use, the definition of composition must satisfy an intuitive notion of composition as well as satisfy the composition theorem.

We propose the following as an intuitive requirement on composition:

- *Suppose that (st_1, st_2, ag) is a transition of cmp_1. Then, there must exist st_3 and st_4 such that:*
  - st_1 *and* st_3 *are equivalent with respect to* cmp_1*'s view*,
  - st_2 *and* st_4 *are equivalent with respect to* cmp_1*'s view*, *and*
  - (st_3, st_4, ag) *is an element of guar for the composite.*

  This requires that each transition by a component is equivalent (from that component's perspective) to some transition by the composite. Intuitively, this requires that the composite can do at least as much as the component.

- *An analogous condition holds with the roles of the components reversed.*

- *Any transition allowed for the environment of the composite is allowed for the environment of each component.*

By proving that the above requirements hold before composing two components, we can obtain some confidence that the result satisfies the intuitive notion of composition. The conditions given in Section 6 are sufficient to establish the above conditions. Consequently, it sufficies to prove each of the conditions in Section 6 to ensure that the composition theorem can be used and that the composition is intuitively meaningful.

9 An Example

This section provides an example of using the framework provided in the previous sections to specify and verify properties of a composite system. The example system considered here is a much simplified version of an architecture for a secure computing system. The key to the architecture is that policy enforcement is separated from policy decisions. The components in the example are:

- A *kernel* which provides services to client processes.
- A *security server* which performs policy computations as requested by the kernel.

The example presented here is a much simplified version of the DTOS security architecture described in references [4] and [3]. The kernel is policy neutral in that it simply enforces policy decisions made by the security server. The kernel attaches labels called *security identifiers* (SIDs) to processes and system resources. At each enforcement point, the kernel asks the security server to make a policy decision based on the SID of the accessing process and the SID of the resource to be accessed. The security server responds with an *access vector* indicating the allowed access modes for the process to the resource. The security server computes this access vector based on a security database that is private to the security server. The kernel uses the returned access vector to determine whether to provide access to the resource. For simplicity, the only class of resource considered in the example is files,[5] and the only access mode controlled by the policy is write access. The overall system policy is that the kernel only provides services that are permitted by the policy defined in the security server.

One example of the type of policy that might be defined in the security server is a Multilevel Security (MLS) policy[5]. In this policy, sensitivity levels (such as UNCLASSIFIED, SECRET, and TOP SECRET) are associated with processes and resources and policy decisions are made based on the levels. For example, a process with level TOP SECRET is permitted to read (but not write) a file with level UNCLASSIFIED. This type of policy would be enforced by having the security server maintain a correspondence between SIDs and levels and perform the policy computations using the MLS rules. Upon receiving a request from the kernel, the security server would first determine the levels associated with the provided SIDs, next perform the computation, and finally return the access vector. Another example of the type of policy that might be defined in the security server is type enforcement[3]. In this policy, a process SID is mapped to

[5] The DTOS kernel does not actually provide files as a resource. Instead, files are provided by an operating system "personality" that runs on top of DTOS. We use files in the example here since we expect readers to have more familiarity with files than the resources in the actual DTOS kernel.

a *domain* and a resource SID is mapped to a *type* and policy computation rules are based on domains and types.

The intent is for the same kernel to be able to support sites with a variety of security policies. If a site desires a unique policy, it simply constructs a security server encoding that policy. A prototype DTOS kernel and security server have been constructed and some experiments have been performed to demonstrate that the policy can be changed without changing the kernel. However, there is still the question of how to perform the assurance for this system. If the system is viewed as a monolith, then each time a new security server is defined, the system specification must be viewed as having changed and there is the question of how much analysis must be redone.

It would be preferable for the assurance analysis to take advantage of the architecture of the system. The system is intentionally architected so that the kernel can be a reusable component. By using the composition framework, the kernel can be specified and verified as a separate component. Each new security server is separately specified and verified. Then, the composition theorem is used to combine the properties proved of the kernel and a given security server into an overall system security policy. Although this separation of the kernel and security server appears pointless in the simple example provided here, it is expected to be quite valuable for the actual DTOS system. The specifications that have been written for the DTOS kernel are a couple hundred pages long and there are over 100 separate access modes controlled in the system. Given the complexity of the kernel, it is valuable to be able to analyze it once and reuse the analysis each time a new security server is defined.

In the presentation that follows, we start with the overall system policy and use it to drive the specification of the components. This is an example of using the composition theorem to support system decomposition. We could just as well have used a bottom-up approach starting with the component specifications. Only the presentation would have been different.[6]

9.1 System

The system consists of the kernel as well as the security server. It provides service to the client as allowed by the policy defined in the security server. The only service provided by the kernel in this simple example is that of modifying a file.

[6] The PVS specifications are actually written in the bottom-up approach since PVS requires definition before use. Also note that the components are not quite independent since some definitions (for example, type definitions) are shared between the components.

9.1.1 State

The state of the system can be thought of as being comprised of the following classes of data:

- Data private to the kernel.
- Data private to the security server.
- Data representing the interface between the kernel and security server.
- Data representing the interface between the kernel and user processes.

Kernel Data
The data private to the kernel consists of:

- $process\_sid$ — a function mapping each process in the system to a SID
- $file\_sid$ — a function mapping each file in the system to a SID
- $file\_data$ — a function mapping each file in the system to its contents
- $active\_process$ — the user process currently executing on the processor[7]

Security Server Data
The only data private to the security server is:

- $ss\_write\_allowed$ — a function indicating when a process with a given SID is permitted to write to a file with a given SID

As discussed previously, the data private to a real security server would be more complex. A security server would typically have structures mapping SIDs to *security contexts* and structures associating access vectors with security context pairs. For simplicity, we abstract the security server here to simply the resulting function from a pair of SIDs to a Boolean.

Security Server Interface Data
The interface data through which the kernel and security server communicate consists of:

- $checking$ — a Boolean that when set to $true$ denotes that the kernel is waiting for a policy computation to be provided by the security server
- $psid$ — a SID indicating the process SID for a policy computation
- $fsid$ — a SID indicating the file SID for a policy computation

[7] For simplicity, we assume a uniprocessor system.

- $done$ — a Boolean that when set to $true$ indicates that the security server has completed a policy computation that the kernel has not yet "consumed"
- $write\_allowed$ — A Boolean indicating the result of a policy computation completed by the security server

User Process Interface Data
The interface data through which the kernel and user processes communicate consists of:

- $write\_requested$ — a function indicating whether a process has submitted a *write* request which has not been processed
- $write\_params$ — a function indicating the input parameters for a *write* request submitted by a process; $write\_params(p) = (f, d)$ means that process p submitted a request to write d to file f

9.1.2 Specification

The overall system is the composition of the kernel and the security server. The translators $ktokss$ and $stokss$ are used to translate states in the composition. The agents for both components as well as the overall system are of the same type, so the identity translation is used to map agents in the composition.

9.1.3 Properties

The property we prove of the overall system is:

> *(kss_ac_prop) Any transition that modifies the contents of a file has a starting state in which the security policy defined in the security server allows the current process to modify the file. In addition, the agent for any such transition is an agent of the kernel.*

The proof of this property is in terms of the following properties of the individual components:

- *(kss_write_preda) Any transition that modifies the contents of a file has a starting state in which:*
  - *psid is the active process' SID and fsid is the file's SID*
  - *checking, done, and write_allowed are set to true*

  In addition, the agent for such a transition is a kernel agent.
- *(kss_checking_pred) Every reachable state is such that whenever checking and done are both set to true, write_allowed has the same value as ss_write_allowed(psid,fsid).*

These properties are the translations of the properties *kernel_write_preda* (see Section 9.2.3) and *ss_checking_pred* (see Section 9.3.3) for the individual components. It is straightforward to prove that the kernel and security server components defined in the following sections satisfy the conditions of the composition theorem. Thus, to show that these properties hold for the composite system, it suffices to show that they hold for at least one of the components. Another way to view this is that the composition theorem implies that these requirements are sufficient conditions to allocate to the individual components of the system. We now describe the individual components and demonstrate that they ensure these properties hold.

9.2 Kernel

The kernel implements the service of modifying a file.

9.2.1 State

The kernel state consists of the data private to the kernel as well as the interface data. We define:

- $kernel\_view(st_1, st_2)$ to test whether the kernel state portions of two states are the same. This returns $true$ exactly when *process_sid*, *write_requested*, *write_params*, *file_sid*, *file_data*, *active_process*, *checking*, *psid*, *fsid*, *done*, and *write_allowed* have the same value in both states.
- $kernel\_priv(st_1, st_2)$ to test whether the kernel private portions of two states are the same. This returns $true$ exactly when *process_sid*,*file_sid*,*file_data*, and *active_process* have the same value in both states.
- $kernel\_ss\_int(st_1, st_2)$ to identify the non security server interface portions of the state. This returns $true$ exactly when $kernel\_priv$ returns $true$ and *write_requested* and *write_params* have the same value in both states.

  This says that the portions of the state hidden from the security server are the kernel private data and the fields *write_requested* and *write_params* which are the kernel's interface to user processes.
- $kernel\_p\_int(st_1, st_2, p)$ to identify the non user process interface portions of the state. This returns $true$ exactly when $kernel\_priv$ returns $true$ and:
  - *checking*, *psid*, *fsid*, *done*, and *write_allowed* have the same value in both states, and
  - For any p_1 other than p, $write\_requested(p_1)$ and $write\_params(p_1)$ have the same value in both states.

This says that the portions of the state hidden from a given user process are the kernel private data, the security server interface data, and the interface data for other user processes.

- $kernel\_hidd(ag)$ as the $hidd$ function for the kernel. The sets of states returned by it are as follows:
  - When $ag = kp$, the set of all pairs of states satisfying $kernel\_view$.
  - When $ag = sp$, the set of pairs of states for which $kernel\_ss\_int(st_1, st_2)$ returns $true$.
  - Otherwise, the set of pairs of states for which $kernel\_p\_int(st_1, st_2, ag)$ returns $true$.

  Here, kp and sp are distinguished, distinct agents used to represent, respectively, the kernel and security server.

The only restriction we place on the initial states for the kernel is that $checking$, $done$, and $write\_allowed$ are set to $false$. The function $kernel\_init$ tests a state to see whether these fields are set to $false$.
We define $kernel\_cags$ to be the set consisting of the single element, kp. This is the agent used to denote that a transition is caused by the kernel.

9.2.2 Specification

To specify the kernel as a component, a definition must be provided for each of the six pieces of a component. The $init$, $cags$, $view$, and $hidd$ components are as defined in the previous section.
The $guar$ piece defines the following transitions as being allowed by the kernel:

- *start_check(f,d)* — ask the security server whether f can be modified
  - $checking$ is set to $false$ in the starting state,
  - $write\_requested(active\_process)$ holds in the starting state,
  - $write\_params(active\_process) = (f, d)$ in the starting state,
  - $checking$ is set to $true$ and $done$ is set to $false$ in the final state, and
  - $psid$ and $fsid$ in the final state are set to the active process' SID and f's SID in the starting state
  - No other portions of the state that are visible to the kernel are changed.
- *write_file(f,d)* — if the policy allows f to be modified, then f is modified to have contents d
  - $write\_requested(active\_process)$ holds in the starting state,
  - $write\_params(active\_process) = (f, d)$ in the starting state,
  - $write\_allowed$, $done$, and $checking$ are set to $true$ in the starting state,
  - $fsid$ is set to $file\_sid(f)$ in the starting state,
  - the contents of f are changed to d, and $checking$ and $write\_requested(active\_process)$ are set to $false$ in the final state, and
  - No other portions of the state that are visible to the kernel are changed.
- $fail\_write(f, d)$ — if the policy does not allow f to be modified, then the policy computation is simply "consumed"
  - $write\_requested(active\_process)$ holds in the starting state,
  - $write\_params(active\_process) = (f, d)$ in the starting state,
  - $done$ and $checking$ are set to $true$ and $write\_allowed$ is set to $false$ in the starting state,
  - $checking$ and $write\_requested(active\_process)$ are set to $false$ in the final state, and
  - No other portions of the state that are visible to the kernel are changed.
- $schedule\_process$ — a new process is made the active process; this cannot occur in the middle of a policy computation
  - $checking$ is set to $false$ in the starting state, and
  - $active\_process$ is changed to a new process, and
  - No other portions of the state that are visible to the kernel are changed.
- $kernel\_stutter$ — no change is made to the kernel state.

We use two criteria in determining the transitions allowed in $rely$ for the kernel:

- Transitions allowed by the kernel's $rely$ must not modify private data or interface data for other agents.

- Transitions allowed by the kernel's $rely$ must not violate invariant properties desired for the kernel. The only such property we define here is that:

  Whenever checking is set to true, psid is set to the active process' SID.

  Since $process\_sid$ and $active\_process$ are contained in the kernel's private state, the only requirement that needs to be placed on the kernel's environment to ensure this property is that it does not modify $psid$ or $checking$.

So, we define $kernel\_rely$ to consist of all transitions that leave the $process\_sid$, $file\_sid$, $file\_data$, $active\_process$, $psid$, and $checking$ fields of the state unchanged as well as satisfying $kernel\_hidd$.
It is straightforward to prove that the given definition of the $kernel\_component$ satisfies the requirements for a component defined in Section 3.

9.2.3 Properties

It is straightforward to prove the following properties hold for the kernel:

- *(kernel_write_prop) Any transition that modifies the contents of a file has a starting state in which:*
  - *fsid is the file's SID*
  - *checking, done, and write_allowed are set to true*

  In addition, the agent for such a transition is a kernel agent.

  This follows by inspection of $kernel\_guar$ and $kernel\_rely$.

- *(kernel_checking_pred) Every reachable state is such that whenever checking is set to true, psid is set to the SID of the active process.*

  This follows by noting that it holds in any initial state and is held stable by $kernel\_guar$ and $kernel\_rely$.

- *(kernel_write_preda) Any transition that modifies the contents of a file has a starting state in which:*
  - *psid is the active process' SID and fsid is the file's SID,*
  - *checking, done, and write_allowed are set to true*

  In addition, the agent for such a transition is a kernel agent.

  This follows immediately from the first two properties demonstrated. The first property ensures everything other than the requirement on $psid$. The second property ensures the requirement on $psid$ since $checking$ holds in the starting state.

9.3 Security Server

9.3.1 State

The security server state consists of the data private to the security server as well as the interface data. We define:

- $ss\_view(st_1, st_2)$ to test whether the security server state portions of two states are the same. This returns $true$ exactly when *checking*, *psid*, *fsid*, *done*, *ss_write_allowed*, and *write_allowed* have the same value in both states.

- $ss\_priv(st_1, st_2)$ to test whether the security server private portions of two states are the same. This returns $true$ exactly when *ss_write_allowed* has the same value in both states.

- $ss\_hidd(ag)$ as the $hidd$ function for the security server. The sets of states returned by it are as follows:
  - When $ag = kp$, the set of all pairs of states satisfying ss_priv.
  - Otherwise, the set of pairs of states for which ss_view returns $true$.

The only restriction we place on the initial states for the security server is that $checking$, $done$, and $write\_allowed$ are set to $false$ and $ss\_write\_allowed$ is set to a constant, $policy\_database$. The function ss_init tests a state to see whether these requirements are satisfied.
We define ss_cags to be the set consisting of the single element, sp. This is the agent used to denote that a transition is caused by the security server.

9.3.2 Specification

To specify the security server as a component, a definition must be provided for each of the six pieces of a component. The $init$, $cags$, $view$, and $hidd$ components are as defined in the previous section.
The $guar$ piece defines the following transitions as being allowed by the security server:

- *compute_access* — determine whether an access is allowed
  - $checking$ is set to $true$ and $done$ is set to $false$ in the starting state,
  - $ss\_write\_allowed$, $checking$, $psid$, and $fsid$ are not changed,
  - $done$ is set to $true$ and $write\_allowed$ is set to $ss\_write\_allowed(psid, fsid)$ in the final state.

- $ss\_stutter$ — no change is made to the security server portions of the state.

We use two criteria in determining the transitions allowed in $rely$ for the security server:

- Transitions allowed by the security server's $rely$ must not change private portions of the security server state.
- Transitions allowed by the security server's $rely$ must not violate invariant properties desired for the security server. The only such property we define here is that:

  > *Whenever checking and done are both set to true, write_allowed is set to ss_write_allowed(psid,fsid).*

Thus, we require that $rely$ contains only transitions in which $ss\_write\_allowed$ is not altered and the following conditions hold whenever $checking$ and $done$ are set to $true$ in the final state for the transition:

- $checking$ and $done$ are set to $true$ in the starting state for the transition, and
- $write\_allowed$, $psid$, and $fsid$ are not changed.

It is straightforward to prove that the given definition of the $ss\_component$ satisfies the requirements for a component defined in Section 3.

9.3.3 Properties

It is straightforward to prove the following property holds for the security server:

- *(ss_checking_pred) Every reachable state is such that whenever checking and done are set to true, write_allowed is set to ss_write_allowed(psid,fsid).*

  This follows by noting that it holds in any initial state and is held stable by ss_guar and ss_rely.

10 Conclusion

In terms of writing specifications, the framework described here seems quite usable. The operations supported by each component can be specified in the "standard" manner and the framework can be used to combine the individual operations into a component specification. Although the use of translator functions in the composition is a little awkward, it is not that difficult and the specifications still seem readable. Requiring composition to be pairwise is somewhat clumsy especially for complex systems with a larger number of components. A possible enhancement would be to define a function which takes a sequence of components and composes them together using the pairwise composition. This might be more convenient for specification writers.
The approach used to accomplish the composition is a hybrid of the approaches advocated by Abadi-Lamport[1, 2] and Shankar[6]. This approach retains the following advantages of the individual approaches:

- Components must be proven to be appropriate for composition before reasoning about the composite.
- The introduction of new, private data structures in other components does not require updates to be made to the environmental assumptions of a component. That the component does not modify these data structures is implicit in the data structures being part of the private state of other components.
- The framework makes a clear distinction between the initial states, allowed transitions, and allowed environment transitions for each component. In addition, the framework forces the specification of the agents that are permitted to cause each transition.
- Most of the reasoning about a composite system can be reduced to reasoning about individual components.

An obvious disadvantage of using the modular specification approach rather than specifying the example as a monolithic entity is that is was necessary to specify how the individual components interacted. The shared components of the state ($checking$, $psid$, $fsid$, $done$, and $write\_allowed$) are used to model the communication protocol between the kernel and the security server. The specifications provided here are surprisingly long considering the complexity of the example. However, this is actually a trade-off between the accuracy of the model and the amount of effort required to analyze the model. By explicitly modeling the communication between the components, the correspondence between the model and the actual system is more obvious. It is also important to note that the modular specification approach has advantages from a maintenance standpoint. For example, suppose the security server were later replaced by a different security server that satisfied the same properties used in the correctness proof of the overall system. Then, the analysis of the system could be updated by simply reproving the security server properties. It would not be necessary to reprove properties of the kernel. Although more experience is required using the framework, we currently believe that the benefits resulting from the modular approach are worth the increased complexity resulting from specifying the communication between components. This is especially true since the whole intent of the DTOS system architecture is to allow for the replacement of the security server.

Note that we define a transition to be a state-state-agent triple rather than a state-state pair. Although the Abadi-Lamport work allows for transitions to be specified in this form, the examples they typically provide specify transitions as simply relations between a starting and final state. Furthermore, in some versions of the Abadi-Lamport work, they completely ignore agents. The Shankar work completely ignores agents, too. Our primary area of application is security, and we have found that specifying the agent for each transition is critical to security analysis. When only correctness is of concern, the component that performs a step is irrelevant as long as it is correctly performed. When security is a concern, who causes a transition is just as important as whether the transition is performed correctly.

In future releases of this report, we could start investigating liveness properties, but many properties of interest are safety properties and require no consideration of liveness. Our plan for how liveness properties will eventually be addressed is that two extra fields would be added to the definition of a component. One field would describe the Abadi-Lamport "weak fairness" conditions, while the other field would describe the Abadi-Lamport "strong fairness" conditions.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. Technical Report 118, Digital Equipment Corporation, Systems Research Center, Dec. 1993.

[2] J. de Bakker, W. de Roever, and G. Rosenberg. *Stepwise Refinement of Distributed Systems, LNCS 430*. Springer-Verlag, 1990.

[3] T. Fine and S. E. Minear. Assuring Distributed Trusted Mach. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 206–218, Oakland, CA, May 1993.

[4] S. E. Minear. Providing Policy Control Over Object Operations in a Mach Based System. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141–156, Salt Lake City, Utah, June 1995.

[5] NCSC. Trusted Computer Systems Evaluation Criteria. Standard, DOD 5200.28-STD, US National Computer Security Center, Fort George G. Meade, Maryland 20755-6000, Dec. 1985.

[6] N. Shankar. A lazy approach to compositional verification. Technical Report TSL-93-08, SRI International, Dec. 1993.