Verifying Hyperproperties With TLA

Leslie Lamport*

Fred B. Schneider†

***Abstract*—Hyperproperties generalize ordinary properties by expressing relations among multiple executions of a system. Self-composition has been used to reduce verifying that a system satisfies certain classes of hyperproperties to verifying that a derived system satisfies an ordinary property. By describing systems and their properties in the temporal logic TLA, we use self-composition to handle a larger class of hyperproperties that includes those we have seen that express security conditions. TLA tools are used to verify that high-level designs of industrial systems satisfy properties. Now, they can also verify that those systems satisfy these hyperproperties. No prior knowledge of hyperproperties or TLA is assumed.**

***Index Terms*—TLA, hyperproperties, verification**

I. Introduction

A *property* is a predicate on executions; it is true or false of an individual execution. Classical verification shows that a system satisfies a property. A *hyperproperty* is a predicate on sets of executions; it is true or false of a set of executions. New logics and tools have been developed to verify that systems satisfy certain classes of hyperproperties [3, 5, 6, 9, 12, 15, 18, 30]. We instead use TLA [19], a temporal logic supported by languages (TLA$^+$ and PlusCal) along with tools that have been developed and used in industry for two decades. A concluding discussion section compares our approach to prior work.

We show how to reduce verifying that systems satisfy a large class of hyperproperties—which we call *finitary* hyperproperties—to verifying TLA formulas. We start with a system described by a TLA formula P and a hyperproperty expressed by a formula $\mathcal{H}$ involving k behaviors. To assert that the system satisfies $\mathcal{H}$, we give a TLA formula $Q \Rightarrow R$ containing k copies of P. Formula Q describes a new system comprising multiple copies of the system running in parallel, so $Q \Rightarrow R$ asserts that this new system satisfies property R. Such an approach is called *self-composition* [6] and has been used before when R does not contain P. Because TLA is expressive enough to describe systems, we can allow R to contain copies of P and thereby handle a larger class of hyperproperties.

Such a reduction would be of little interest without a practical method to represent real systems and to verify the resulting formulas $Q \Rightarrow R$. TLA$^+$ [20] is a language based on TLA that is used in industry [28] to specify and verify high-level designs of complex concurrent and distributed software and hardware systems. Its tools include a model checker and a proof checker. Those tools were developed for verifying a TLA formula asserting that a system satisfies a property, including the case of a system implementing a higher-level system. We show that the tools can also verify a subclass of finitary hyperproperties called $\forall\exists$-hyperproperties, which includes specifications of system security and other examples that motivate hyperproperty verification in the literature. Accurately expressing these specifications uses a property of TLA called stuttering insensitivity in a new way.

In principle, TLA$^+$ can be used to verify descriptions of systems at any level of abstraction. In practice, TLA$^+$ and its tools are most useful for verifying high-level designs of systems—designs at the algorithm level rather than the code level. Such verification is especially important for concurrent systems, where it is easy to make algorithmic errors and difficult to find and correct those errors in the code. Having verified a high-level design, we would like to know that the property verified is preserved under refinement to an implementation. A simple condition ensures this to be the case for ordinary properties. It had already been observed in the context of security that hyperproperties need not be preserved under refinement [27]. We give a new mathematical analysis of when a refinement does preserve a $\forall\exists$-hyperproperty.

We assume no knowledge of hyperproperties or TLA. After some preliminaries, we describe a representation of hyperproperties in temporal logic. We then introduce RTLA, a temporal logic similar to TLA but lacking stuttering insensitivity. How hyperproperties are verified is illustrated with RTLA by verifying that a tiny system satisfies generalized noninterference (GNI)—a hyperproperty chosen to illustrate most of the issues that arise with our approach. Another small example shows that stuttering insensitivity is required to state GNI properly. We then introduce TLA and sketch a TLA verification that both small examples satisfy GNI; a TLA$^+$ formalization is available on the Web [22]. TLA$^+$ has already been used [4] to prove that a model of a commercial system satisfies a hyperproperty called observational determinism, but that proof required recording the execution history using an auxiliary variable. Section VII describes that work and shows how our method allows a direct proof that the system satisfies the hyperproperty, with no auxiliary variables. In Section VIII, we formulate other well-known security hyperproperties in TLA.

II. Preliminaries

An execution is often modeled as a sequence of states. Even in methods that describe executions in terms of events, a system is usually described by a state machine in which events are generated by state transitions—examples are Mealy machines, Büchi automata, and I/O automata [23]. Such a description

*Microsoft Research http://lamport.org

†Computer Science Department, Cornell University, Ithaca, New York. Email: fbs@cs.cornell.edu. Supported, in part, by AFOSR grant F9550-19-1-0264, and NSF grant 1642120.

corresponds to a state-based one, where events correspond to state changes.

A. Behaviors and State Machines

We call a sequence of states a *behavior*, and we call a pair of consecutive states in a behavior a *step*. Behaviors representing executions have usually been described by state machines, written in diverse ways such as Turing machines, Petri nets, and C++ programs. A state machine can be described by an *initial predicate* $\mathcal{I}$ on states and a *next-state predicate* $\mathcal{N}$ on pairs of states. (The set of states need not be finite.) The behaviors generated by the state machine are ones in which predicate $\mathcal{I}$ is true on the first state of the sequence and predicate $\mathcal{N}$ is true on every step.

A concurrent system can be described by a state machine in which each step represents operations performed by one or more processes at the same time. (Usually, when describing asynchronous systems, each step represents an operation of a single process.) The state machine representing an asynchronous system is generally nondeterministic, allowing a state to have multiple next states.

We consider state machines whose states are assignments of values to variables. For example, we can describe an hour-minute clock by a state containing variables hr and min that represent the hour and minute, respectively. We write this state machine's initial predicate as a formula containing the variables hr and min. For a 12-hour clock that reads 12:00 when first plugged in, the initial predicate is:

$$\mathcal{I}_{hm} \;\triangleq\; (min = 0) \wedge (hr = 12)$$

The clock's next-state predicate $\mathcal{N}$ is a formula containing unprimed and primed variables, where v represents the value of variable v in the first state and v' represents its value in the second state. For the hour-minute clock, the next-state predicate is:

$$\begin{aligned}\mathcal{N}_{hm} \;\triangleq\;\; & \;\; min' = (min + 1) \bmod 60 \\ & \wedge\; hr' = \text{IF } min = 59 \\ & \qquad\qquad \text{THEN IF } hr = 12 \text{ THEN } 1 \\ & \qquad\qquad\qquad\qquad\qquad\;\; \text{ELSE } hr + 1 \\ & \qquad\qquad \text{ELSE } hr\end{aligned}$$

B. Properties

Since we represent executions as behaviors, a *property* is a predicate on behaviors. We write $b \models P$ to mean that property P is true of behavior b. For a set S of behaviors, we let $S \models P$ mean that $b \models P$ is true for all $b \in S$. Verification traditionally establishes that all behaviors of a state machine that corresponds to a system satisfy some property P, which means verifying $S \models P$ where S is the set of all behaviors generated by the machine. For example, termination can be expressed as $S \models \mathit{Term}$, where $b \models \mathit{Term}$ is true iff (if and only if) b reaches a terminating state.

There is a natural correspondence between subsets of a set and predicates on the elements of that set. A predicate P on a set U corresponds to the subset of all elements of U for which P is true. Thus, a property corresponds to a set of behaviors.[1] We consider a property both to be a predicate on behaviors and the set of behaviors satisfying that predicate; each view is at times the more useful. Propositional logic operators on the predicates correspond to ordinary set operations—for example, $\vee$ corresponds to $\cup$ (set union), $\Rightarrow$ (implication) corresponds to $\subseteq$ (subset), $\equiv$ (equivalence) corresponds to $=$, and $\neg$ corresponds to set complement.

If we identify the property P with the set of behaviors satisfying P, then $S \models P$ means that $S \subseteq P$ is valid. If we regard the set S of behaviors to be a property, then $S \models P$ means that $b \models (S \Rightarrow P)$ is true for all behaviors b. For a property Q, let $\models Q$ mean that Q is true for all behaviors, so $S \models P$ is equivalent to $\models (S \Rightarrow P)$. Verification traditionally has been formulated as showing $S \models P$ rather than $\models (S \Rightarrow P)$ because state machines and properties were written and thought of in different ways.

C. Making State Machines Do Something

In our description of a state machine, the next-state predicate specifies only what steps are allowed. It says nothing about what steps must occur. This omission was deliberate. For reasons irrelevant to this paper, we want the initial predicate and next-state predicate to allow behaviors that end at any point—even though the next-state predicate allows further steps. To require that certain steps must occur, we add to the description a *supplementary* property that must also be satisfied by behaviors of the state machine. For example, the supplementary property of the state machine describing a concurrent system might require that steps representing operations performed by a non-terminated process keep occurring in the behavior. The supplementary property is generally a liveness property [2], and most often a fairness property [16]. However, here we make no assumption about supplementary properties.

III. Hyperproperties

A. Hyperproperties as Predicates on Sets of Behaviors

Properties cannot directly describe certain security conditions [27], so they were generalized to hyperproperties [10]. A *hyperproperty* is a predicate on sets of behaviors rather than on a single behavior, making it a predicate on properties. An example is the hyperproperty $\mathcal{H}$ where, for a property P, we define $\mathcal{H}(P)$ to be true iff:

> Any two terminating behaviors satisfying P that have different initial values of x have different terminal values of y.

Viewing a property P to be a set of behaviors, we define $\mathcal{H}$ to be a *finitary* hyperproperty iff $\mathcal{H}(P)$ can be written as a formula using propositional logic operators and quantification of the form $\forall\, b \in P$ with predicates $F(b_1, \ldots, b_n)$ that depend only

[1] We do not require states to form a set, so behaviors form a class—a collection that may be "too big" to be a set. (For example, the class of all sets is not a set.) We informally use the term *set* because it is more familiar than *class*.

on the behaviors b_i and not on P.[2] (Since negation is allowed, we can also write quantification of the form $\exists\, b \in P$.)

B. Hyperproperties as Predicates on Behaviors

By a standard result in predicate logic, it is always possible to "move all the quantifiers to the outside" (renaming bound variables, if necessary) in the definition of a finitary hyperproperty $\mathcal{H}$ and rewrite $\mathcal{H}(P)$ as

$$\forall\exists\, b_1 \in P \,:\, \ldots \forall\exists\, b_k \in P \,:\, J(b_1, \ldots, b_k) \tag{1}$$

where each $\forall\exists$ is either $\forall$ or $\exists$, and J does not depend on P. Verifying that a property P satisfies the hyperproperty $\mathcal{H}$ means verifying formula (1). Verifying that P satisfies a property is the special case:

$$\forall\, b \in P \,:\, J(b) \tag{2}$$

Methods developed over the past half century for verifying (2) (when P is described by a state machine) are not directly applicable to (1).

Our goal is to find a way to apply methods for verifying (2)—that is, verifying ordinary properties—to finitary hyperproperties. Self-composition has been used for the special case in which every quantifier $\forall\exists$ of (1) is the universal quantifier $\forall$ [6, 15, 30]. In that case, we let P^k be the state machine defined by running k copies of the state machine P in parallel, where a possible state of P^k is a k-tuple of possible states of P. We can then write (1) as

$$\forall\, b \in P^k \,:\, J(\pi_1(b), \ldots, \pi_k(b)) \tag{3}$$

where π_i is the element-by-element projection that maps from a sequence of k-tuples to the sequence of their i^{th} components. Formula (3) has the same form as (2).

We will generalize this approach to the class we call $\forall\exists$-hyperproperties—those with definitions of the form:

$$\begin{aligned} &\forall\, b_1 \in P \,:\, \ldots \forall\, b_j \in P \,:\\ &\quad K(b_1, \ldots, b_j) \;\Rightarrow\\ &\qquad \exists\, b_{j+1} \in P \,:\, \ldots \exists\, b_k \in P \,:\, L(b_1, \ldots, b_k) \end{aligned} \tag{4}$$

The methods we use might generalize further, but (4) is the most general form for which we know that a practical approach for verifying ordinary properties can be directly applied. Moreover, all published finitary hyperproperties that we have found are of this form.

Formula (4) views P as a set of behaviors. We now rewrite it with P viewed as a predicate on behaviors. We replace "$\forall\, b \in P :$" by "$\forall\, b : P(b) \Rightarrow$" and replace "$\exists\, b \in P :$" by "$\exists\, b : P(b) \wedge$". Doing that and applying a bit of predicate logic, (4) becomes:

$$\begin{aligned} &\forall\, b_1, \ldots, b_j \,:\, P(b_1) \wedge \ldots \wedge P(b_j) \wedge K(b_1, \ldots, b_j)\\ &\quad \Rightarrow\; \exists\, b_{j+1}, \ldots, b_k \,:\\ &\qquad\qquad P(b_{j+1}) \wedge \ldots \wedge P(b_k) \wedge L(b_1, \ldots, b_k) \end{aligned} \tag{5}$$

[2]The only non-finitary hyperproperties $\mathcal{H}$ we know for which it is interesting to verify that $\mathcal{H}(P)$ holds involve the probability of system P doing something. Those hyperproperties require a probability measure on P.

In this formula, P, K, and L are predicates on behaviors. We will write them in a state-based temporal logic. The value of a variable in such a logic describes part of the system state at some instant of time. Therefore, we must assume that the dependence of P, K, and L on any behavior b_i is formulated using only a finite number of variables that describe the system state. Such an assumption seems necessary for using a state-based logic to describe properties or hyperproperties.

C. Hyperproperties as Temporal Logic Formulas

We use a linear-time temporal logic, so the meaning of a formula is a predicate on behaviors. Temporal formulas are obtained from state predicates by applying temporal operators and the ordinary operators of predicate logic. For example, state predicate $x > y$ is true on a state iff the value of x in that state is greater than the value of y in that state. Interpreted as a temporal formula, it is true of a behavior iff it is true in the first state of that behavior. The temporal operator $\Box$ (read *always* or *henceforth*) is defined by letting $b \models \Box F$ be true iff $c \models F$ is true for c equal to b and all suffixes of b. Thus, $b \models \Box(x > y)$ is true iff $x > y$ is true for all states of b.

In temporal logic, variables can have different values in different states of a behavior,[3] just like variables in a programming language. We assume that our temporal logic has the usual temporal existential quantifier $\boldsymbol{\exists}$ over variables [24], where $b \models \boldsymbol{\exists}\, x : F$ asserts that there exists a behavior $\widehat{b}$ that is the same as b, except that the values of x in the states of $\widehat{b}$ and b may differ, such that $\widehat{b} \models F$ is true. Unlike formula $\exists\, x : F$ of ordinary predicate logic, which is true iff there exists a single value for x that makes F true, the temporal operator $\boldsymbol{\exists}\, x : F$ is true for a behavior iff there exists a sequence of values for x, one for each state of the behavior, that make F true. The quantifier $\boldsymbol{\exists}$ obeys all the rules that the quantifier $\exists$ of predicate logic does.

Formulas $\boldsymbol{\exists}\, x : F(x)$ and $\boldsymbol{\exists}\, y : F(y)$ say nothing about the values actually assumed by the variables x and y in a behavior. The symbols x and y in these formulas are called *bound* variables. It can be useful to think of $\boldsymbol{\exists}\, x : F(x)$ as the formula obtained by "hiding" variable x of $F(x)$, and we sometimes use the term *hidden* variables for bound variables. Unbound variables are called *free* variables.

We now rewrite (5) as a temporal logic formula. Formula (5) refers to k behaviors b_i. A temporal logic formula can refer only to a single behavior, which we call b. So, we encode the k behaviors b_i in b. As assumed above, (5) depends only on the values that the states of the behaviors b_i assign to some variables. Call those variables $v_1, \ldots, v_n$. We now also assume that formula P can then be written as a temporal logic formula $\widetilde{P}$ containing only those variables. To write (5) as a temporal formula about a single behavior b, we replace $P(b_i)$ in (5) with the formula obtained from $\widetilde{P}$ by substituting new variables for $v_1, \ldots, v_n$—a different set of variables for each i.

[3]What we call variables here are usually called *flexible variables*. Temporal logic also has *rigid variables* whose values are the same in all states of a behavior, but they will not concern us.

We need a notation for the formula obtained from $\widetilde{P}$ by substituting new variables $x_1, \ldots, x_n$ for $v_1, \ldots, v_n$. Existing notations for writing this formula are cumbersome. So, we introduce some new notation that is informal, but whose meaning should be clear. We write the formula produced by the substitution as $\widetilde{P}(x_1, \ldots, x_n)$. Moreover, we let $\mathbf{x}$ be an abbreviation for $x_1, \ldots, x_n$, so we can write the formula as $\widetilde{P}(\mathbf{x})$; and we do the same for other boldface identifiers. We also let $\mathbf{x_i}$ denote the list $x_{i,1}, \ldots, x_{i,n}$ of variables.

To write (5) as a temporal formula, which is a predicate on behaviors b, we replace each $P(b_i)$ by $\widetilde{P}(\mathbf{x_i})$. The values that b assigns to the variables $\mathbf{x_i}$ are thus interpreted as the values that the behavior b_i assigns to the variables $v_1, \ldots, v_n$. We also assume that $K(b_1, \ldots, b_j)$ and $L(b_1, \ldots, b_k)$ can be written as temporal logic formulas $\widetilde{K}(\mathbf{x_1}, \ldots, \mathbf{x_j})$ and $\widetilde{L}(\mathbf{x_1}, \ldots, \mathbf{x_k})$. We can then write (5) as

$$\begin{aligned} &\models \widetilde{P}(\mathbf{x_1}) \wedge \ldots \wedge \widetilde{P}(\mathbf{x_j}) \wedge \widetilde{K}(\mathbf{x_1}, \ldots, \mathbf{x_j}) \\ &\quad \Rightarrow \boldsymbol{\exists}\, \mathbf{x_{j+1}}, \ldots, \mathbf{x_k} : \\ &\qquad\qquad \widetilde{P}(\mathbf{x_{j+1}}) \wedge \ldots \wedge \widetilde{P}(\mathbf{x_k}) \wedge \widetilde{L}(\mathbf{x_1}, \ldots, \mathbf{x_k}) \end{aligned} \tag{6}$$

because $\models F$ asserts that F is true for all behaviors. For convenience, we drop the "$\sim$" and let P identify both the temporal formula $\widetilde{P}$ and the predicate on behaviors that it represents, and we do the same for K and L, so (6) becomes:

$$\begin{aligned} &\models P(\mathbf{x_1}) \wedge \ldots \wedge P(\mathbf{x_j}) \wedge K(\mathbf{x_1}, \ldots, \mathbf{x_j}) \\ &\quad \Rightarrow \boldsymbol{\exists}\, \mathbf{x_{j+1}}, \ldots, \mathbf{x_k} : \\ &\qquad\qquad P(\mathbf{x_{j+1}}) \wedge \ldots \wedge P(\mathbf{x_k}) \wedge L(\mathbf{x_1}, \ldots, \mathbf{x_k}) \end{aligned} \tag{7}$$

This formula asserts that the system described by the temporal logic formula P satisfies the hyperproperty defined by (5). Thus, if the predicates P, K, and L on behaviors can be written as temporal logic formulas, then the assertion (5) that a system satisfies a hyperproperty can also be written as a temporal logic formula.

D. RTLA

The introduction of temporal logic to verification provided a formalism for stating and verifying a rich class of properties. The original logic given by Amir Pnueli [29] had only the single temporal operator $\Box$, described above. The logic could not express many properties of interest, so additional temporal operators were subsequently proposed, including $\boldsymbol{\exists}$ (though it was not widely used). However, attempts to express (the sets of behaviors generated by) state machines as temporal logic properties still did not prove to be practical. So temporal logic verification consisted of proving formulas $S \models P$, where the property P was expressed in temporal logic and state machine S was expressed in some other way—usually as an automaton or in something like a programming language.

One way TLA differs from other temporal logics is by building its formulas not from state predicates, but from predicates on steps (pairs of states). We call these predicates *actions*. In TLA, an action is written as a formula containing primed and unprimed variables, the way we wrote the next-state predicate $\mathcal{N}_{hm}$ in Section II-A. A state predicate in TLA is just an action containing no primed variables, so it depends only on the first state of a state pair. The only temporal operators in TLA are $\Box$, $\boldsymbol{\exists}$, and operators defined in terms of them.

Instead of explaining TLA directly, we begin with the slightly simpler logic RTLA. It contains the operators $\Box$ and $\boldsymbol{\exists}$ defined above. An action, interpreted as an RTLA formula, is true of a behavior iff it is true of the first step of the behavior. So, the definition of $\Box$ implies that b is a possible behavior of the state machine described by the initial predicate $\mathcal{I}$ and the next-state predicate $\mathcal{N}$ iff $b \models \mathcal{I} \wedge \Box\mathcal{N}$ is true, since $b \models \mathcal{I}$ asserts that the first state of b satisfies $\mathcal{I}$, and $b \models \Box\mathcal{N}$ asserts that the first step of every suffix of b satisfies $\mathcal{N}$. (Every step of b is the first step of a suffix of b).

A supplementary property asserting that the state machine must generate some steps is expressed by an RTLA formula, but we will not explain how. The only supplementary property we need in this paper is one asserting that the behavior cannot end in a state in which an $\mathcal{N}$ step is possible. It is written WF($\mathcal{N}$).

E. Hiding

It would be impossible to express even rather simple temporal properties in RTLA without the operator $\boldsymbol{\exists}$. For example, consider the property that is true of a behavior iff the value of x cannot equal 1 unless it has previously equaled 42. Since this property depends only on the value of x, it can contain only the variable x; but it can't be expressed by a formula containing only x just by using the temporal operator $\Box$. However, it's easy to write that property as follows using a (Boolean-valued) hidden variable y:

$$\begin{aligned} \boldsymbol{\exists}\, y : \quad & (y = (x = 42)) \\ & \wedge\ \Box((\neg y \Rightarrow (x' \neq 1))\ \wedge\ (y' = (y \vee (x = 42)))) \end{aligned}$$

This property has the form $\mathcal{I} \wedge \Box\mathcal{N}$ of a state machine, but with a hidden variable y. The property is easy to express without a hidden variable using the temporal operators of most temporal logics. However, more complicated temporal properties are easier to understand when written as a state machine with hidden variables than when written in terms of those temporal operators.

F. Verification

Traditionally, verification has meant showing $S \models P$, for a state machine S and a property P. This can be written in temporal logic as $\models (S \Rightarrow P)$. Properties can be written with $\boldsymbol{\exists}$, so $\models (S \Rightarrow P)$ can have the form

$$\models (\boldsymbol{\exists}\, \mathbf{y} : S(\mathbf{x}, \mathbf{y})) \ \Rightarrow\ (\boldsymbol{\exists}\, \mathbf{z} : P(\mathbf{x}, \mathbf{z})) \tag{8}$$

where $\mathbf{x}$ are the free variables of S and P, and $\mathbf{y}$ and $\mathbf{z}$ are their respective hidden variables. By simple predicate logic, (8) is equivalent to

$$\models S(\mathbf{x}, \mathbf{y}) \ \Rightarrow\ (\boldsymbol{\exists}\, \mathbf{z} : P(\mathbf{x}, \mathbf{z})) \tag{9}$$

which asserts that, for any behavior satisfying $S(\mathbf{x}, \mathbf{y})$, we can find assignments of values to the variables $\mathbf{z}$ in each state of the behavior that makes $P(\mathbf{x}, \mathbf{z})$ true. The value assigned to $\mathbf{z}$ in any state of the behavior might depend on the values assigned to $\mathbf{x}$ and $\mathbf{y}$ in all the states of the behavior. Verification of (9) becomes much simpler if the assignment of values to $\mathbf{z}$ in any state depends only on the values of $\mathbf{x}$ and $\mathbf{y}$ in that state. Let a *state function* be any expression containing constants and unprimed variables (so a state predicate is a Boolean-valued state function). Letting n be such that $\mathbf{z}$ is $z_1, \ldots, z_n$, we verify (9) by finding state functions $f_1(\mathbf{x}, \mathbf{y}), \ldots, f_n(\mathbf{x}, \mathbf{y})$ that make this formula true:

$$\models S(\mathbf{x}, \mathbf{y}) \;\Rightarrow\; P(\mathbf{x}, \mathbf{f}(\mathbf{x}, \mathbf{y})) \tag{10}$$

where $\mathbf{f}$ is the list $f_1, \ldots, f_n$ of state functions and $P(\mathbf{x}, \mathbf{f}(\mathbf{x}, \mathbf{y}))$ is the formula obtained from $P(\mathbf{x}, \mathbf{z})$ by substituting $f_i(\mathbf{x}, \mathbf{y})$ for z_i, for each i. The formulas $f_i(\mathbf{x}, \mathbf{y})$ are called a *refinement mapping* [1].

In (10), we are substituting state functions $\mathbf{f}(\mathbf{x}, \mathbf{y})$ for the variables $\mathbf{z}$. Substituting a state function f for a variable v in an RTLA formula includes substituting f' for v', where the value of f' is the value of f in the next state, so the formula f' is obtained by priming all variables in f.

The validity of (9) does not imply that there exists a refinement mapping $\mathbf{f}$ satisfying (10). However, we can (in principle) always find such a refinement mapping if we replace S by an equivalent formula obtained by adding *auxiliary variables* to it [1]. Adding auxiliary variables $\mathbf{a}$ to $S(\mathbf{x}, \mathbf{y})$ means finding a formula $S^{\mathbf{a}}(\mathbf{x}, \mathbf{y}, \mathbf{a})$ that is equivalent to $S(\mathbf{x}, \mathbf{y})$ when the variables $\mathbf{a}$ are hidden—that is, where $\boldsymbol{\exists}\, \mathbf{a} : S^{\mathbf{a}}(\mathbf{x}, \mathbf{y}, \mathbf{a})$ is equivalent to $S(\mathbf{x}, \mathbf{y})$ [21]. We can then verify (9) by verifying:

$$\models S^{\mathbf{a}}(\mathbf{x}, \mathbf{y}, \mathbf{a}) \;\Rightarrow\; P(\mathbf{x}, \mathbf{f}(\mathbf{x}, \mathbf{y}, \mathbf{a}))$$

IV. GNI

Generalized noninterference (GNI) [26] is a hyperproperty that was proposed as a security condition for systems. We illustrate our method by showing that two example state machines satisfy GNI. Notable features of these verifications are: the refinement mappings for an existentially quantified copy of P in (7) and the use of stuttering insensitivity to express GNI in a state-based formalism. Whether GNI is useful is irrelevant.

GNI and other security conditions are usually stated in a model where an execution is described as a sequence of events rather than as a sequence of states. Events are classified as *public*, which are visible to all observers, or *secret*, which are visible only to privileged observers. GNI is a condition meant to ensure that a system's public events provide no information about its secret events. It asserts that for any two possible executions, there is a third possible execution having the public events of the first and the secret events of the second.

We first express GNI in RTLA and then describe a tiny example state machine that is easily be seen to satisfy GNI.

A. GNI in RTLA

In a state-based formulation of GNI, part of the state is public and part is secret. We take GNI to mean that observing public state reveals no information about secret state. Our state-based assertion that a system satisfies GNI can be written in the form (5) as follows:

$$\begin{aligned} \forall\, b_1, b_2 \,:\, P(b_1) \wedge P(b_2) \;\Rightarrow& \\ \exists\, b_3 \,:\, & P(b_3) \wedge L(b_1, b_2, b_3) \end{aligned} \tag{11}$$

where $P(b_i)$ asserts that b_i is a possible behavior of the system, and $L(b_1, b_2, b_3)$ asserts that the public state of b_3 is always the same as that of b_1 and the secret state of b_3 is always the same as that of b_2. We translate (11) to temporal logic the way we translated (5) to (7). To express $L(b_1, b_2, b_3)$ as a temporal logic formula, we assume that we are given state functions *public* and *secret* that characterize the public and secret state of the system. These state functions are parameters of the definition, just like P. The translation of (11) to temporal logic is then:

$$\models P(\mathbf{x_1}) \wedge P(\mathbf{x_2}) \Rightarrow \boldsymbol{\exists}\, \mathbf{x_3} \,:\, P(\mathbf{x_3}) \wedge L(\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}) \tag{12}$$

$$\begin{aligned} \text{where}\quad L(\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}) \;\triangleq&\\ \Box\,(\;\; & \mathit{public}(\mathbf{x_3}) = \mathit{public}(\mathbf{x_1}) \\ \wedge\;\; & \mathit{secret}(\mathbf{x_3}) = \mathit{secret}(\mathbf{x_2})\,) \end{aligned}$$

Remember that (12) asserts that a temporal logic formula, which is a predicate on behaviors, is true for every behavior b. In that formula, the values that b assigns to the variables $\mathbf{x_i}$ correspond to behavior b_i of (11).

B. *System* Tiny

System *Tiny* alternately produces a public output value and reads a secret input value, where values are elements of a set *Val*. The value of the variable *in* is the last input value read, and the value of the variable *out* is the last value output. The initial values of *in* and *out* are arbitrary. The value of the hidden variable *nin* determines whether the next step is a *Pub* step that produces a public output or a *Sec* step that reads a secret input. These steps can produce or read any value in *Val*.

System *Tiny* cannot satisfy GNI if a behavior could stop after taking an arbitrary numbers of steps. This is because *Tiny* produces one input value for every output value, so a behavior can have the public outputs of behavior b_1 and the secret inputs of b_2 only if the lengths of b_1 and b_2 differ by at most 1. We make *Tiny* satisfy GNI by requiring that its executions never stop, which we do by requiring it to satisfy the liveness condition $\mathrm{WF}(\mathcal{N})$.

RTLA formula P that describes the *Tiny* state machine is defined in Figure 1. Also defined there are the state functions *public* and *secret* for which we expect *Tiny* to satisfy GNI. Since *nin* is a hidden variable, it is not part of the actual state of *Tiny*, so it makes no sense to consider it either public or secret.

C. *Verifying That* Tiny *Satisfies GNI*

For j in $\{1, 2, 3\}$, let $\mathcal{I}_j$, …, secret_j be the formulas obtained from the formulas $\mathcal{I}$, …, *secret* defined in Figure 1

$$
\begin{aligned}
\mathcal{I} \;&\triangleq\; \quad in \in Val \\
&\quad \wedge\; out \in Val \\
&\quad \wedge\; nin = 0
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{N} \;&\triangleq\; Pub \vee Sec \\
&\text{where } Pub \triangleq \quad nin = 0 \wedge nin' = 1 \\
&\qquad\qquad\quad \wedge\; out' \in Val \\
&\qquad\qquad\quad \wedge\; in' = in \\
&\qquad\quad\; Sec \triangleq \quad nin = 1 \wedge nin' = 0 \\
&\qquad\qquad\quad \wedge\; in' \in Val \\
&\qquad\qquad\quad \wedge\; out' = out
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{L} \;&\triangleq\; \mathrm{WF}(\mathcal{N}) \\
Q \;&\triangleq\; \mathcal{I} \wedge \Box\mathcal{N} \wedge \mathcal{L} \\
P \;&\triangleq\; \exists\, nin : Q \\
public \;&\triangleq\; out \\
secret \;&\triangleq\; in
\end{aligned}
$$

Fig. 1. The RTLA Description of System *Tiny*.

by substituting new variables in_j, out_j, nin_j for the variables in, out, nin. With this notation, (12) becomes

$$
\begin{aligned}
&\models P_1 \wedge P_2 \;\Rightarrow \\
&\quad \exists\, in_3, out_3 : \\
&\qquad P_3 \wedge \Box((public_3 = public_1) \wedge (secret_3 = secret_2))
\end{aligned} \tag{13}
$$

The definition of P in Figure 1 and predicate logic reasoning shows that (13) is equivalent to

$$
\begin{aligned}
&\models Q_1 \wedge Q_2 \;\Rightarrow \\
&\quad \exists\, in_3, out_3, nin_3 : \\
&\qquad Q_3 \wedge \Box((public_3 = public_1) \wedge (secret_3 = secret_2))
\end{aligned} \tag{14}
$$

Expanding the definitions of Q and $\mathcal{L}$ and using the temporal logic tautology $\Box(F \wedge G) \equiv \Box F \wedge \Box G$, we see that $Q_1 \wedge Q_2$ is equivalent to

$$
(\mathcal{I}_1 \wedge \mathcal{I}_2) \;\wedge\; \Box(\mathcal{N}_1 \wedge \mathcal{N}_2) \;\wedge\; (\mathrm{WF}(\mathcal{N}_1) \wedge \mathrm{WF}(\mathcal{N}_2)) \tag{15}
$$

Formula (14) has the form of (9), and the equivalence of $Q_1 \wedge Q_2$ and (15) shows that the left-hand side of the implication is equivalent to the standard RTLA description of a state machine. Thus (13) has the form of (9), the kind of formula that arises in verifying that a state machine satisfies a temporal property.

As we observed above, we verify (9) by finding an appropriate refinement mapping $\mathbf{f}$ and verifying (10). The required refinement mapping should assign to each of the variables of P_3 the following functions of the variables of P_1 and P_2:

$$
in_3 \leftarrow in_2 \qquad out_3 \leftarrow out_1 \qquad nin_3 \leftarrow nin_2 \tag{16}
$$

Tiny is a tiny finite-state system, and it should be easy to verify (14) with a model checker. However, there are no tools for RTLA. We will see that it is easy to capture the meaning of (14) in a TLA formula, and that formula is easy to verify with TLC, the TLA model checker.

V. From RTLA to TLA

A. Stuttering Insensitivity

We have eliminated the distinction between state machines and properties by representing both with RTLA formulas. The assertion that a state machine S satisfies a property P is $\models (S \Rightarrow P)$. It would seem natural for implementation to be the same as satisfying a property, so for a state machine S_1 to implement a state machine S_2 would mean this formula is valid:

$$
\models S_1 \Rightarrow S_2 \tag{17}
$$

A description of an hour-minute clock should not imply that the clock has no display showing seconds—or no radio, or no alarm. So, (17) should be valid even if S_1 describes an hour-minute-second clock and S_2 describes an hour-minute clock. An hour-minute clock (that is allowed to stop) is described by this RTLA formula

$$
S_{hm} \;\triangleq\; \mathcal{I}_{hm} \wedge \Box\mathcal{N}_{hm} \tag{18}
$$

where $\mathcal{I}_{hm}$ and $\mathcal{N}_{hm}$ are defined in Section II-A. It is straightforward to modify S_{hm} by adding a variable scd, which represents seconds, to obtain an RTLA formula S_{hms} that describes an hour-minute-second clock. For these clock descriptions, (17) becomes

$$
\models S_{hms} \Rightarrow S_{hm} \tag{19}
$$

However, (19) is invalid. A behavior satisfying S_{hms} must take 59 steps that change only scd between steps that change min, but those scd-changing steps are not allowed by S_{hm}.

Formula (19) is invalid because S_{hm} describes an hour-minute clock only in a universe consisting just of the clock—or more precisely, a universe described by just the variables hr and min. Formula S_{hm} does not describe an hour-minute clock in a universe also containing the variable scd. For that universe, the description of an hour-minute clock must also allow steps that leave hr and min unchanged. We should write a description of the clock that is satisfied by every system that implements it. Moreover, that description should be appropriate for a universe containing other systems too—a universe for which a state consists of an assignment of values to scd and all other possible variables.

Having a potentially infinite number of variables might seem strange, but it's what math does. An equation like $x + y = 3$ is not about a universe containing only the variables x and y. There is no problem combining this equation with one containing the variable z. Every math formula is about a universe in which you can always talk about another variable. So a temporal logic formula containing only the variables hr and min should not rule out other variables; it should just make no explicit statement about their values.

The problem with RTLA formula S_{hm} is that it makes an implicit statement about every possible variable—namely, that the values of those variables change only when the value of min changes. In addition to steps satisfying next-state action $\mathcal{N}_{hm}$, formula S_{hm} should permit steps that allow other variables, including scd, to change but leave hr and min unchanged. Those additional steps satisfy $(hr' = hr) \wedge (min' = min)$. Since a tuple is left unchanged iff its components are left unchanged, we can write this formula as $\langle hr, min \rangle' = \langle hr, min \rangle$, where angle brackets $\langle\ \rangle$ enclose tuples. So, to obtain an RTLA formula that describes an hour-minute clock and does not constrain the rest of the universe, we redefine S_{hm}:

$$S_{hm} \ \triangleq\ \mathcal{I}_{hm} \wedge \Box(\mathcal{N}_{hm} \vee (\langle hr, min \rangle' = \langle hr, min \rangle)) \quad (20)$$

Formula S_{hm} defined by (20) is *stuttering insensitive* (SI), meaning that whether it is satisfied by a behavior is not affected by adding and/or removing from the behavior steps that leave its free variables (hr and min) unchanged.

We now define SI more precisely. Two sequences of values are *stuttering-equivalent* iff removing all repeated values from both produces identical sequences. For example, these two sequences of numbers are stuttering equivalent, since removing all repeated values from each produces the increasing sequence of all positive integers:

$$\begin{array}{l} 1,\ 1,\ 2,\ 2,\ 3,\ 3,\ 4,\ 4,\ 5,\ 5,\ 6,\ 6,\ 7,\ 7,\ 8,\ 8,\ \ldots \\ 1,\ 2,\ 2,\ 3,\ 3,\ 3,\ 4,\ 4,\ 4,\ 4,\ 5,\ 5,\ 5,\ 5,\ 5,\ 6,\ \ldots \end{array}$$

For any state function f and behavior b, define $b|_f$ to be the sequence of values obtained by evaluating f on the states of b. Define behaviors b_1 and b_2 to be f*-stuttering equivalent* iff $b_1|_f$ and $b_2|_f$ are stuttering equivalent. For S_{hm}, SI means that for any two behaviors b_1 and b_2 that are $\langle hr, min \rangle$-stuttering equivalent, $b_1 \models S_{hm}$ is true iff $b_2 \models S_{hm}$ is. In general, a temporal formula $F(\mathbf{x})$ with free variables $\mathbf{x}$ is SI iff, for any two behaviors b_1 and b_2 that are $\langle \mathbf{x} \rangle$-stuttering equivalent, $b_1 \models F(\mathbf{x})$ is equivalent to $b_2 \models F(\mathbf{x})$.

There are many ways to view SI. For our purposes, the best is to consider a behavior not as representing an execution of a system, but rather as being a movie film of an execution. Each frame of the film depicts a state, and the entire film is taken by a camera that can record at a varying speed, taking more or fewer frames. The only requirement for the film is that the state produced by each step during an execution of the system appears in at least one frame.

A formula is a predicate on behaviors, and we want it to be an assertion about executions—not about films of executions. If a system is described by the variables $\mathbf{x}$, then two behaviors b_1 and b_2 are films of the same execution iff they are $\langle \mathbf{x} \rangle$-stuttering equivalent. Therefore, a formula $F(\mathbf{x})$, which is a predicate on behaviors, is an assertion about executions and not just about particular films of executions iff $b_1 \models F(\mathbf{x})$ is equivalent to $b_2 \models F(\mathbf{x})$ for any $\langle \mathbf{x} \rangle$-stuttering equivalent behaviors b_1 and b_2—precisely the definition of what it means for $F(\mathbf{x})$ to be SI.

There is another way to express SI. We introduce a new temporal operator $\sim$ on state functions such that $b \models (f \sim g)$ is true for a behavior b iff $b|_f$ is stuttering equivalent to $b|_g$. For lists of variables $\mathbf{x}$ and $\mathbf{y}$, we abbreviate $\langle \mathbf{x} \rangle \sim \langle \mathbf{y} \rangle$ as $\mathbf{x} \sim \mathbf{y}$. A temporal formula $F(\mathbf{x})$ is SI iff

$$\models (\mathbf{x} \sim \mathbf{y}) \Rightarrow (F(\mathbf{x}) = F(\mathbf{y})) \quad (21)$$

for lists $\mathbf{x}$ and $\mathbf{y}$ of variables. The operator $\sim$ can be defined in TLA; it is used in expressing hyperproperties.

B. TLA

TLA is obtained by modifying RTLA so that every syntactically correct TLA formula is SI. This requires two changes to RTLA. Define $[A]_f$ to equal $A \vee (f' = f)$ for an action A and a state function f. The first change to RTLA is that, in TLA, primed variables may appear in a temporal formula only in an action A in a subformula $\Box[A]_f$, for some state function f. Thus (20) is written in TLA as

$$S_{hm} \ \triangleq\ \mathcal{I}_{hm} \ \wedge\ \Box[\mathcal{N}_{hm}]_{\langle hr, min \rangle} \quad (22)$$

The second change to RTLA to ensure SI is to the definition of $\exists$. In TLA, $b \models \exists\, y : F(\mathbf{x}, y)$ is defined to be true iff there exists a behavior $\widehat{b}$ that is $\langle \mathbf{x} \rangle$-stuttering equivalent to b such that $\widehat{b} \models F$ is true. (Note that $\widehat{b}$ can be obtained from a behavior $\overline{b}$ that is $\langle \mathbf{x}, y \rangle$-stuttering equivalent to b by changing the values that the states of $\overline{b}$ assign to y.) If S_{hms} is redefined to be the TLA formula describing an hour-minute-second clock with the definition analogous to (22), then $\exists\, scd : S_{hms}$ is equivalent to S_{hm}. With the RTLA definition of $\exists$, formula $\exists\, scd : S_{hms}$ would not be SI because it would be true only of behaviors that contain at least 59 steps (corresponding to steps changing scd required by S_{hms}) that leave hr and min unchanged between every step that changes min.

Formula S_{hm} defined in (22) allows behaviors ending with an infinite number of steps that leave hr and min unchanged. Such a behavior represents an execution in which the clock has stopped. (As explained in Section II-C, a supplementary property is needed to ensure that the clock doesn't stop.) Since termination can always be represented as a system's variables remaining forever unchanged, we do not need finite behaviors. So, for simplicity, we assume all behaviors are infinite.

VI. GNI Revisited

A TLA formula is also an RTLA formula; so (12), which defines what it means for P to satisfy GNI, is a TLA assertion if P and L are TLA formulas. Although (12) was written for systems P described in RTLA, we might expect it also to be suitable for systems described in TLA. It isn't. In particular, we would expect $Tiny$ to satisfy GNI, but we will show that its TLA description (given below) does not satisfy (12). We then describe a system $Little$ that should satisfy GNI, but even its RTLA description does not satisfy (12). That example leads us to replace (12) with a TLA formula that corresponds to the usual event-based definition of GNI.

A. Tiny *in TLA*

To describe $Tiny$ in TLA, we replace the definition of Q in Figure 1 by

$$Q \ \triangleq\ \mathcal{I} \ \wedge\ \Box[\mathcal{N}]_{\langle in, out, nin \rangle} \ \wedge\ \mathcal{L}$$

Also, the RTLA formula $\mathrm{WF}(\mathcal{N})$ is not SI and must be replaced in the definition of $\mathcal{L}$ by $\mathrm{WF}_{\langle in, out, nin \rangle}(\mathcal{N})$, whose definition can be found elsewhere [20].

We now show that the resulting TLA formula P does not satisfy (12), where each $\mathbf{x_i}$ is the list in_i, out_i of variables. Consider a behavior b in which the variables $\mathbf{x_1}$ and $\mathbf{x_2}$ assume sequences $s_1, s_2, \ldots$ and $t_1, t_2, \ldots$ of values that describe two behaviors b_1 and b_2 satisfying P. Suppose that these sequences begin as follows where, for example, $Pub(\mathbf{x_2})$ indicates that a step satisfies $Pub(\mathbf{x_2}, nin)$ for some values of nin and nin'.

$$\begin{array}{lccccccccc}
\mathbf{x_1}: & s_1 & \xrightarrow{Pub(\mathbf{x_1})} & s_2 & \xrightarrow{Sec(\mathbf{x_1})} & s_3 & \xrightarrow{Pub(\mathbf{x_1})} & s_4 & \ldots \\
\mathbf{x_2}: & t_1 & \xrightarrow{Pub(\mathbf{x_2})} & t_2 & & t_2 & \xrightarrow{Sec(\mathbf{x_2})} & t_3 & \ldots
\end{array}$$

The lists $\mathbf{x_1}$ and $\mathbf{x_2}$ of variables represent the values of the variables in and out in a behavior b that encodes behaviors b_1 and b_2. As allowed by the TLA formula P, the values t_2 of variables $\mathbf{x_2}$ do not change in the second step of behavior b. Let's also suppose that each of the Pub steps changes the value of out, and each of the Sec steps changes the value of in.

For behavior b to satisfy (12), there must exist values for $\mathbf{x_3}$ representing a behavior b_3 that satisfies P, where the public part of the state (the value for out) of $\mathbf{x_3}$ comes from $\mathbf{x_1}$ and the secret part (the value for in) comes from $\mathbf{x_2}$. But in the third step of the behavior, the variables of $\mathbf{x_3}$ that represent both in and out change, which is not allowed for a step of a behavior of $Tiny$. Therefore, no such $\mathbf{x_3}$ exists, and behavior b does not satisfy (12). So the RTLA definition (12) of P satisfying GNI is not satisfied for the TLA formula P that represents $Tiny$. Our TLA definition of GNI will be satisfied by the TLA formula P.

B. *System* Little

$Little$ is like $Tiny$, except instead of performing one Sec step between every two Pub steps, $Little$ can perform any number (including 0). The definition of the $Little$ state machine is obtained from the $Tiny$ specification of Figure 1 by letting the Pub action set nin to an arbitrary natural number, and letting the Sec action be enabled when $nin \neq 0$ and decrement nin by 1.

If we made just these changes, then there would be a problem in the resulting description of $Little$. A Pub step that output the same value as in the previous step (a step with $out' = out$) and set nin to 0 would be leaving all the variables unchanged. It would represent the system doing nothing—including producing no output—thus describing a system that is not allowed to produce the same output value twice in a row without performing a secret input. (This is not a problem for $Tiny$, in which the value of nin changes whenever an output is produced.) To allow successive Pub steps to output the same value, we include in out a bit that changes with each Pub step, so the value of out is a pair $\langle v, i \rangle$ with v in the set Val and i in $\{0, 1\}$. Such a change to the description of $Little$ is not needed for inputs, since every Sec step changes nin; but we make the same change to the value of in for consistency.

A TLA formula P that describes system $Little$ is defined in Figure 2, where $\oplus$ is the exclusive-or operator, $p[2]$ equals

$$\begin{array}{lcl}
\mathcal{I} & \triangleq & \quad in \in Val \times \{0\} \\
 & & \land \ out \in Val \times \{0\} \\
 & & \land \ nin = 0
\end{array}$$

$$\begin{array}{l}
\mathcal{N} \triangleq Pub \lor Sec \\
\quad \text{where } Pub \triangleq \quad nin = 0 \land nin' \in Nat \\
\qquad\qquad\qquad \land \ out' \in Val \times \{out[2] \oplus 1\} \\
\qquad\qquad\qquad \land \ in' = in \\
\qquad\qquad\ \ Sec \triangleq \quad nin \neq 0 \land nin' = nin - 1 \\
\qquad\qquad\qquad \land \ in' \in Val \times \{in[2] \oplus 1\} \\
\qquad\qquad\qquad \land \ out' = out
\end{array}$$

$$\begin{array}{lcl}
\mathcal{L} & \triangleq & \mathrm{WF}_{\langle in, out, nin \rangle}(\mathcal{N}) \\
Q & \triangleq & \mathcal{I} \ \land \ \Box[\mathcal{N}]_{\langle in, out, nin \rangle} \ \land \ \mathcal{L} \\
P & \triangleq & \exists\!\!\exists\, nin : Q
\end{array}$$

$$\begin{array}{lcl}
public & \triangleq & out \\
secret & \triangleq & in
\end{array}$$

Fig. 2. The TLA Description of System *Little*

the second element of an ordered pair p, and $Val \times \{i\}$ is the set of ordered pairs $\langle v, i \rangle$ with v in Val.

C. *GNI in TLA*

It is obvious how to convert Figure 2 to an RTLA description of *Little*, but the result would not satisfy the RTLA formula (12) for essentially the same reason that the TLA description of $Tiny$ doesn't. Choose the values of $\mathbf{x_1}$ and $\mathbf{x_2}$ representing behaviors b_1 and b_2 of $Little$ shown here:

$$\begin{array}{lccccccccc}
\mathbf{x_1}: & s_1 & \xrightarrow{Pub(\mathbf{x_1})} & s_2 & \xrightarrow{Pub(\mathbf{x_1})} & s_3 & \xrightarrow{Sec(\mathbf{x_1})} & s_4 & \ldots \\
\mathbf{x_2}: & t_1 & \xrightarrow{Pub(\mathbf{x_2})} & t_2 & \xrightarrow{Sec(\mathbf{x_2})} & t_3 & \xrightarrow{Pub(\mathbf{x_2})} & t_4 & \ldots
\end{array}$$

These behaviors are allowed by both the TLA and RTLA versions of $Little$. In the second step, the value of out represented by $\mathbf{x_1}$ and the value of in represented by $\mathbf{x_2}$ both change, so they both change for their values represented by $\mathbf{x_3}$. But, like $Tiny$, $Little$ allows no behavior in which a step changes both in and out, so the required values for $\mathbf{x_3}$, which must describe a behavior b_3 of $Little$, do not exist. Hence, the RTLA version of $Little$ does not satisfy (12), the RTLA version of a system satisfying GNI.

A description of *Little* should not satisfy an RTLA definition of GNI. Satisfying GNI should imply that observing a system's public events provides no information about its secret events. However, the RTLA specification implies that from behavior b_2 in our example, an observer can see that a secret input event occurred between the first two public output events, which is potentially useful information. This information is observable for the same reason RTLA does not consider a behavior described by an hour-minute-second clock with the seconds hidden to be a behavior of an hour-minute clock. That reason is the implicit assumption that a step is an observable event, even if the step changes the values of no variables (perhaps because any step takes an observable amount of time). With this assumption, from behavior b_2 in our example, the public Pub steps reveal the existence of the secret Sec step.

While it doesn't satisfy our RTLA definition of GNI, *Little* does satisfy the usual event-based definition of GNI. Given any behaviors b_1 and b_2 of *Little*, it's easy to find a third behavior b_3 that has the Pub events (changes to out) of b_1 and the Sec events (changes to in) of b_2. For example, suppose b_1 has an infinite number of Sec events. (The fairness condition $\mathcal{L}$ of P implies only that it must have infinitely many Pub events.) Let b_3 have a sequence of Pub steps that change out the same as the Pub steps of b_1 do, but set nin to 1, so each Pub step is followed by one Sec step. Let the Sec steps of b_3 perform the same changes to in as the Sec steps of b_2. Then b_3 is a behavior of *Little* having the same Pub events as b_1 and the same Sec events of b_2, as required to satisfy event-based GNI.

We now present a TLA formula defining GNI that is a state-based version that corresponds to the event-based one. We do so by modifying (12), which is a legal TLA formula if P and L are, but not the right one. Formula (12) states how behavior b_3 must be obtained by combining behaviors b_1 and b_2. But in TLA, a behavior represents a film of a system execution. GNI is about combining executions, not films.

Whatever we want to express in TLA about system executions must be stated in terms of films of executions—including how to construct a film b_3 from films b_1 and b_2. Formally, a film is a behavior. The way to make GNI be about combining executions is to construct behavior b_3 not by combining behaviors b_1 and b_2, but by combining behaviors $\widehat{b_1}$ and $\widehat{b_2}$ of our choice that describe the same executions as b_1 and b_2.

To translate this idea from behaviors to formulas, consider the formula $P(\mathbf{x_1})$ in (12). A behavior b satisfies this formula iff the values that b assigns to variables $\mathbf{x_1}$ constitute a behavior in which the system described by P is satisfied when its variables are renamed to $\mathbf{x_1}$. Moreover, values b assigns to other variables $\widehat{\mathbf{x_1}}$ constitute a behavior of the same execution as the values b assigns to variables $\mathbf{x_1}$ iff $b|_{\langle \mathbf{x_1} \rangle}$ equals $b|_{\langle \widehat{\mathbf{x_1}} \rangle}$, which is equivalent to the condition $b \models \mathbf{x_1} \sim \widehat{\mathbf{x_1}}$. To obtain b_3 from a behavior $\widehat{b_1}$ describing the same execution as b_1, we must replace $public(\mathbf{x_3}) = public(\mathbf{x_1})$ with $public(\mathbf{x_3}) = public(\widehat{\mathbf{x_1}})$ for some $\widehat{\mathbf{x_1}}$ satisfying $\mathbf{x_1} \sim \widehat{\mathbf{x_1}}$. Applying the same reasoning to $\mathbf{x_2}$, we get the following TLA definition of P satisfying GNI, where $\mathbf{x_1}$, $\mathbf{x_2}$, $\mathbf{x_3}$, $\widehat{\mathbf{x_1}}$, and $\widehat{\mathbf{x_2}}$ are all different variables.

$$
\begin{array}{l}
\models P(\mathbf{x_1}) \land P(\mathbf{x_2}) \;\Rightarrow \\
\quad \exists\, \widehat{\mathbf{x_1}}, \widehat{\mathbf{x_2}}, \mathbf{x_3} \;: \\
\qquad (\widehat{\mathbf{x_1}} \sim \mathbf{x_1}) \land (\widehat{\mathbf{x_2}} \sim \mathbf{x_2}) \land P(\mathbf{x_3}) \land L(\widehat{\mathbf{x_1}}, \widehat{\mathbf{x_2}}, \mathbf{x_3}) \\
\text{where } L(\widehat{\mathbf{x_1}}, \widehat{\mathbf{x_2}}, \mathbf{x_3}) \triangleq \Box(\; public(\mathbf{x_3}) = public(\widehat{\mathbf{x_1}}) \\
\qquad\qquad\qquad\qquad\qquad \land\, secret(\mathbf{x_3}) = secret(\widehat{\mathbf{x_2}}))
\end{array}
\tag{23}
$$

This formula is an assertion about behaviors b in which the values of the list of variables $\mathbf{x_i}$ describe behavior b_i (for $\mathbf{i}$ = 1,2,3), and the values of the list of variables $\widehat{\mathbf{x_i}}$ describe behavior $\widehat{b}_i$.

D. Aligning Films

Little does not satisfy the RTLA definition of GNI because there are films b_1 and b_2 in which a step of b_1 satisfying Pub and a step of b_2 satisfying Sec occur in corresponding frames. To show that *Little* satisfies the TLA definition, we construct films $\widehat{b_1}$ and $\widehat{b_2}$ (of the same executions as b_1 and b_2) in which every Pub step of $\widehat{b_1}$ occurs in the same frames as a Pub step of $\widehat{b_2}$. We can achieve this by adding extra frames—steps that leave the values of in, out, and nin unchanged. For example:

$$
\begin{array}{lccccccccc}
\mathbf{x_1} : s_1 & \xrightarrow{Pub(\mathbf{x_1})} & s_2 & \xrightarrow{Pub(\mathbf{x_1})} & s_3 & \xrightarrow{Sec(\mathbf{x_1})} & s_4 & & & \ldots \\
\mathbf{x_2} : t_1 & \xrightarrow{Pub(\mathbf{x_2})} & t_2 & \xrightarrow{Sec(\mathbf{x_2})} & t_3 & \xrightarrow{Pub(\mathbf{x_2})} & t_4 & & & \ldots \\
\widehat{\mathbf{x_1}} : s_1 & \xrightarrow{Pub(\widehat{\mathbf{x_1}})} & s_2 & & s_2 & \xrightarrow{Pub(\widehat{\mathbf{x_1}})} & s_3 & \xrightarrow{Sec(\widehat{\mathbf{x_1}})} & s_4 & \ldots \\
\widehat{\mathbf{x_2}} : t_1 & \xrightarrow{Pub(\widehat{\mathbf{x_2}})} & t_2 & \xrightarrow{Sec(\widehat{\mathbf{x_2}})} & t_3 & \xrightarrow{Pub(\widehat{\mathbf{x_2}})} & t_4 & & & \ldots
\end{array}
$$

To make Pub steps happen in corresponding frames of $\widehat{b_1}$ and $\widehat{b_2}$ by adding frames to b_1 and b_2, the same number of Pub steps must occur in both behaviors. Behaviors b_1 and b_2 do have the same number of Pub steps—namely, ∞—because the supplementary property $\mathcal{L}$ of *Little* implies that every behavior has infinitely many Pub steps.

The ability to match Pub steps in different films is an instance of a general matching rule: For actions A and B, behaviors b, and disjoint lists of variables $\mathbf{x}$ and $\mathbf{y}$, if there are the same number (possibly ∞) of $A(\mathbf{x})$ and $B(\mathbf{y})$ steps in b, then there exist values for $\widehat{\mathbf{x}}$ and $\widehat{\mathbf{y}}$ such that $\mathbf{x} \sim \widehat{\mathbf{x}}$, $\mathbf{y} \sim \widehat{\mathbf{y}}$, and a step is an $A(\widehat{\mathbf{x}})$ step iff it is a $B(\widehat{\mathbf{y}})$ step.

To state the rule precisely, we need a temporal operator $\overset{\#}{=}$, where $b \models A \overset{\#}{=} B$ is true for actions A and B iff there are the same number of A and B steps in b. However, $A \overset{\#}{=} B$ is not SI if A or B could be satisfied by a step that changes no variables, since adding such a step could change whether the behavior has the same number of A and B steps. Just as we apply $\Box$ only to actions of the form $[C]_v$ to ensure that TLA formulas are SI, we apply $\overset{\#}{=}$ only to actions of the form $\langle C \rangle_v$, an action defined to equal $C \land (v' \neq v)$. A $\langle C \rangle_v$ step is thus a C step that changes v. When applying the rule, v is usually a tuple of variables and at least one of them is changed by C, so C equals $\langle C \rangle_v$.

The general rule we are using can now be stated as validity of the following formula for all actions $A(\mathbf{x})$ and $B(\mathbf{y})$, where $\mathbf{x}$ and $\mathbf{y}$ are disjoint lists of variables.

$$
\begin{array}{l}
\models (\langle A(\mathbf{x}) \rangle_{\langle \mathbf{x} \rangle} \overset{\#}{=} \langle B(\mathbf{y}) \rangle_{\langle \mathbf{y} \rangle}) \;\Rightarrow \\
\quad \exists\, \widehat{\mathbf{x}}, \widehat{\mathbf{y}} \;:\; (\widehat{\mathbf{x}} \sim \mathbf{x}) \land (\widehat{\mathbf{y}} \sim \mathbf{y}) \\
\qquad\qquad \land\, \Box[\langle A(\widehat{\mathbf{x}}) \rangle_{\langle \widehat{\mathbf{x}} \rangle} \equiv \langle B(\widehat{\mathbf{y}}) \rangle_{\langle \widehat{\mathbf{y}} \rangle}]_{\langle \widehat{\mathbf{x}}, \widehat{\mathbf{y}} \rangle}
\end{array}
\tag{24}
$$

We use this rule in Section VI-E to verify that *Little* satisfies GNI.

E. Verifying That Little *Satisfies GNI*

To show that *Little* satisfies GNI, we must show that the TLA formula P that describes *Little* satisfies (23), where each of the variable lists $\mathbf{x_i}$ and $\widehat{\mathbf{x_i}}$ comprises two variables representing in and out. Let P_i equal $P(\mathbf{x_i})$, and for the other defined quantities like Q that also depend on nin, let Q_i equal $Q(\mathbf{x_i}, nin_i)$. Expanding the definitions of P_1 and P_2, (23) becomes

$$\begin{aligned}&\models (\exists\, nin_1 : Q_1) \wedge (\exists\, nin_2 : Q_2) \;\Rightarrow \\ &\quad \exists\, \widehat{\mathbf{x_1}}, \widehat{\mathbf{x_2}}, \mathbf{x_3} : \\ &\qquad (\widehat{\mathbf{x_1}} \sim \mathbf{x_1}) \wedge (\widehat{\mathbf{x_2}} \sim \mathbf{x_2}) \wedge P_3 \wedge L(\widehat{\mathbf{x_1}}, \widehat{\mathbf{x_2}}, \mathbf{x_3})\end{aligned} \tag{25}$$

$$\text{where}\quad L(\widehat{\mathbf{x_1}}, \widehat{\mathbf{x_2}}, \mathbf{x_3}) \;\triangleq\; \Box(\;\; public_3 = public(\widehat{\mathbf{x_1}}) \wedge secret_3 = secret(\widehat{\mathbf{x_2}})\,)$$

By predicate logic reasoning, (25) is equivalent to

$$\begin{aligned}&\models Q_1 \wedge Q_2 \;\Rightarrow \\ &\quad \exists\, \widehat{\mathbf{x_1}}, \widehat{\mathbf{x_2}}, \mathbf{x_3} : \\ &\qquad (\widehat{\mathbf{x_1}} \sim \mathbf{x_1}) \wedge (\widehat{\mathbf{x_2}} \sim \mathbf{x_2}) \wedge P_3 \wedge L(\widehat{\mathbf{x_1}}, \widehat{\mathbf{x_2}}, \mathbf{x_3})\end{aligned} \tag{26}$$

Instead of verifying (26), we will verify a condition that implies (26). The definition of $\sim$ implies that $\widehat{\mathbf{x_i}} \sim \mathbf{x_i}$ follows from $\widehat{\mathbf{x_i}}, \widehat{y} \sim \mathbf{x_i}, y$ for any variables y and $\widehat{y}$. Therefore, (26) is implied by:

$$\begin{aligned}&\models Q_1 \wedge Q_2 \;\Rightarrow \\ &\quad \exists\, \widehat{\mathbf{x_1}}, \widehat{nin_1}, \widehat{\mathbf{x_2}}, \widehat{nin_2}, \mathbf{x_3} : \\ &\qquad (\widehat{\mathbf{x_1}}, \widehat{nin_1} \sim \mathbf{x_1}, nin_1) \wedge (\widehat{\mathbf{x_2}}, \widehat{nin_2} \sim \mathbf{x_2}, nin_2) \\ &\qquad\quad \wedge P_3 \wedge L(\widehat{\mathbf{x_1}}, \widehat{\mathbf{x_2}}, \mathbf{x_3})\end{aligned} \tag{27}$$

Verifying (27) verifies (26), which verifies that *Little* satisfies GNI.

Recall that for every behavior b_1 and b_2, we must align the *Pub* steps. Observe that because every behavior satisfying Q has infinitely many *Pub* steps, $Q_1 \wedge Q_2$ implies $\langle Pub(\mathbf{x_1}, nin_1)\rangle_{\langle \mathbf{x_1}, nin_1\rangle} \stackrel{\#}{=} \langle Pub(\mathbf{x_2}, nin_2)\rangle_{\langle \mathbf{x_2}, nin_2\rangle}$. We can thus apply (24), substituting *Pub* for both A and B. A *Pub* step changes *out*, which implies $\langle Pub(\mathbf{x_i}, nin_i)\rangle_{\langle \mathbf{x_i}, nin_i\rangle}$ equals $Pub(\mathbf{x_i}, nin_i)$. Therefore, instantiating (24) yields:

$$\begin{aligned}&\models Q_1 \wedge Q_2 \;\Rightarrow \\ &\quad \exists\, \widehat{\mathbf{x_1}}, \widehat{nin_1}, \widehat{\mathbf{x_2}}, \widehat{nin_2} : \\ &\qquad (\widehat{\mathbf{x_1}}, \widehat{nin_1} \sim \mathbf{x_1}, nin_1) \wedge (\widehat{\mathbf{x_2}}, \widehat{nin_2} \sim \mathbf{x_2}, nin_2) \wedge \\ &\qquad\quad \Box[Pub(\widehat{\mathbf{x_1}}, \widehat{nin_1}) \equiv Pub(\widehat{\mathbf{x_2}}, \widehat{nin_2})]_{\langle \widehat{\mathbf{x_1}}, \widehat{nin_1}, \widehat{\mathbf{x_2}}, \widehat{nin_2}\rangle}\end{aligned} \tag{28}$$

By predicate logic reasoning, (28) implies that to verify (27), it suffices to verify:

$$\begin{aligned}&\models Q_1 \wedge Q_2 \wedge \\ &\quad (\widehat{\mathbf{x_1}}, \widehat{nin_1} \sim \mathbf{x_1}, nin_1) \wedge (\widehat{\mathbf{x_2}}, \widehat{nin_2} \sim \mathbf{x_2}, nin_2) \wedge \\ &\qquad \Box[Pub(\widehat{\mathbf{x_1}}, \widehat{nin_1}) \equiv Pub(\widehat{\mathbf{x_2}}, \widehat{nin_2})]_{\langle \widehat{\mathbf{x_1}}, \widehat{nin_1}, \widehat{\mathbf{x_2}}, \widehat{nin_2}\rangle} \\ &\quad \Rightarrow (\exists\, \mathbf{x_3} : P_3 \wedge L(\widehat{\mathbf{x_1}}, \widehat{\mathbf{x_2}}, \mathbf{x_3}))\end{aligned} \tag{29}$$

Because Q is SI, which is expressed in rule (21), $\widehat{\mathbf{x_i}}, \widehat{nin_i} \sim \mathbf{x_i}, nin_i$ implies that Q_i is equivalent to $Q(\widehat{\mathbf{x_i}}, \widehat{nin_i})$. So, we can verify (29) by verifying

$$\begin{aligned}&\models Q(\widehat{\mathbf{x_1}}, \widehat{nin_1}) \wedge Q(\widehat{\mathbf{x_2}}, \widehat{nin_2}) \wedge \\ &\quad \Box[Pub(\widehat{\mathbf{x_1}}, \widehat{nin_1}) \equiv Pub(\widehat{\mathbf{x_2}}, \widehat{nin_2})]_{\langle \widehat{\mathbf{x_1}}, \widehat{nin_1}, \widehat{\mathbf{x_2}}, \widehat{nin_2}\rangle} \\ &\qquad \Rightarrow (\exists\, \mathbf{x_3} : P(\mathbf{x_3}) \wedge L(\widehat{\mathbf{x_1}}, \widehat{\mathbf{x_2}}, \mathbf{x_3}))\end{aligned} \tag{30}$$

Comparing this with the RTLA version (14) of GNI, we see that we have used the freedom the TLA version provides to replace the films $\mathbf{x_1}$ and $\mathbf{x_2}$ with equivalent films $\widehat{\mathbf{x_1}}$ and $\widehat{\mathbf{x_2}}$ and used rule (24) to add the hypothesis $\Box[Pub(\widehat{\mathbf{x_1}}) \equiv Pub(\widehat{\mathbf{x_2}})]_{\langle \widehat{\mathbf{x_1}}, \widehat{\mathbf{x_2}}\rangle}$ that synchronizes the two films.

By substituting $\mathbf{x_i}$ and nin_i for $\widehat{\mathbf{x_i}}$ and $\widehat{nin_i}$, expanding the definition of P_3, and predicate logic, (30) becomes

$$\begin{aligned}&\models Q_1 \wedge Q_2 \wedge \Box[Pub_1 \equiv Pub_2]_{\langle \mathbf{x_1}, nin_1, \mathbf{x_2}, nin_2\rangle} \;\Rightarrow \\ &\qquad (\exists\, \mathbf{x_3}, nin_3 : Q_3 \wedge L(\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}))\end{aligned} \tag{31}$$

We show in Section VII that this method of reducing verification of (23) to verification of (31) by using rule (24) also works for other hyperproperties that, like GNI, assert for variables $\mathbf{x}$ the existence of values for variables $\widehat{\mathbf{x}}$ with $\widehat{\mathbf{x}} \sim \mathbf{x}$ that satisfy some condition.

We verify that *Little* satisfies (31) in the same way we verified that *Tiny* satisfies (14): We rewrite

$$Q_1 \wedge Q_2 \wedge \Box[Pub_1 \equiv Pub_2]_{\langle \mathbf{x_1}, nin_1, \mathbf{x_2}, nin_2\rangle} \tag{32}$$

in the form of a state machine description. This rewriting is more complicated than it was for *Tiny* because: (i) there is the additional third conjunct in (32), and (ii) the TLA definition of $Q(\mathbf{x_i}, nin_i)$ has the term $\Box[\mathcal{N}(\mathbf{x_i}, nin_i)]_{\langle \mathbf{x_i}, nin_i\rangle}$ instead of $\Box\mathcal{N}(\mathbf{x_i}, nin_i)$. Let v equal $\langle in, out, nin\rangle$. Expanding the definition of P and rearranging the terms, (32) becomes

$$\begin{aligned}&(\mathcal{I}_1 \wedge \mathcal{I}_2) \wedge (\Box[\mathcal{N}_1]_{v_1} \wedge \Box[\mathcal{N}_2]_{v_2} \wedge \Box[Pub_1 \equiv Pub_2]_{\langle v_1, v_2\rangle}) \\ &\quad \wedge (\mathcal{L}_1 \wedge \mathcal{L}_2)\end{aligned}$$

To transform this to a standard TLA state machine description, we write the shaded expression as $\Box[\mathcal{M}]_{\langle v_1, v_2\rangle}$ for a next-state action $\mathcal{M}$. Using the rule that $\Box$ distributes over $\wedge$ and remembering that $[A]_u$ equals $A \vee (u' = u)$, we see that we can let $\mathcal{M}$ equal

$$\begin{aligned}&((\mathcal{N}_1 \wedge \mathcal{N}_2) \vee (\mathcal{N}_1 \wedge (v_2' = v_2)) \vee (\mathcal{N}_2 \wedge (v_1' = v_1)) \\ &\quad \wedge (Pub_1 \equiv Pub_2)\end{aligned}$$

Expanding the definition of $\mathcal{N}$ and using the facts that *Pub* implies $(\neg Sec) \wedge (v' \neq v)$ and $Pub_1 \equiv Pub_2$, we can rewrite this formula as

$$\begin{aligned}&\quad (Pub_1 \wedge Pub_2) \\ &\vee\; (Sec_1 \wedge Sec_2) \vee (Sec_1 \wedge (v_2' = v_2)) \vee (Sec_2 \wedge (v_1' = v_1))\end{aligned}$$

which is the next-state action of a state machine with variables $\mathbf{x_1}$, nin_1, $\mathbf{x_2}$, and nin_2. Having rewritten (32) as a TLA description of a state machine, verifying (31) is a standard problem of verifying that a state machine satisfies a property. Its verification uses the same refinement mapping used for *Tiny*.

Here is a summary of what we have just done. Using (28), we reduced verifying that *Little* satisfies GNI to verifying (31). By rewriting (32) as a TLA description of a state machine, we reduced verifying (31) to a standard verification problem for which the TLA$^+$ tools were designed. TLC, the TLA$^+$ model checker, easily checks the rewritten version of (31) for models that substitute a small set of values for *Val* and bound the value of *nin* (by substituting a small set $\{0, \ldots, n\}$ for *Nat*). TLAPS, the TLA$^+$ proof checker, can easily check a proof of (31) without the liveness condition $\mathcal{L}_3$ of P_3; features needed to allow TLAPS to check liveness proofs are currently being implemented. For a complete verification, we should also check two more things: our rewriting of (32), which we did using TLAPS, and the two hypotheses we used to obtain (28). The first hypothesis, that every behavior of *Little* has infinitely many *Pub* steps, was checked by TLC on a small model. The second, that $\langle Pub \rangle_v$ equals Pub, is easily checked by TLAPS. The complete TLA$^+$ specifications, including proofs, are available on the Web [22].

F. Verifying That Tiny Satisfies GNI in TLA

Section IV-C explains how to verify that the RTLA description of *Tiny* satisfies the RTLA version of GNI. Essentially the same verification used there shows that the TLA description of *Tiny* satisfies the TLA version of GNI. For the TLA verification, we do exactly what we did for *Little*, except using action $\mathcal{N}$ instead of Pub. With the subscripting notation and definition of v as $\langle in, out, nin \rangle$ from Section VI-E, formula (32) then becomes:

$$Q_1 \,\wedge\, Q_2 \,\wedge\, \Box[\mathcal{N}_1 \equiv \mathcal{N}_2]_{\langle v_1, v_2 \rangle} \tag{33}$$

Expanding the definitions of Q and $\mathcal{L}$, temporal logic reasoning shows that (33) is equivalent to:

$$\begin{aligned} &(\mathcal{I}_1 \wedge \mathcal{I}_2) \;\wedge\; \Box[\mathcal{N}_1 \wedge \mathcal{N}_2]_{\langle v_1, v_2 \rangle} \\ &\quad \wedge\; (\mathrm{WF}_{v_1}(\mathcal{N}_1) \,\wedge\, \mathrm{WF}_{v_2}(\mathcal{N}_2)) \end{aligned} \tag{34}$$

This is the formula one would obtain from (15) by turning an RTLA description of a state machine into a TLA one. We then verify (23) using the same refinement mapping and essentially the same verification as for the RTLA version of *Tiny*. The TLA$^+$ formalizations are on the Web [22].

We obtained the TLA proof that *Tiny* satisfies GNI from its RTLA proof in Section IV-C by replacing $Q(\mathbf{x_1}) \wedge Q(\mathbf{x_2})$ with (33). This same transformation from an RTLA proof to a TLA proof works for any RTLA proof that a system satisfies the RTLA definition of GNI.

VII. PharOS and Observational Determinism

PharOS is a system in which multiple agents communicate by asynchronous message passing subject to real-time constraints on message-delivery time and on when actions may be performed. (It has been commercialized under the name Asterios®.) A goal of the system is *determinacy*—that the behavior of any agent is independent of the scheduling of agent actions. Azaiez et al. [4] proved that a high-level model of the system satisfies this goal. Their proof combined state-based and semantic behavioral reasoning, relating the two by adding an auxiliary variable to record the system's complete execution.

Determinacy in PharOS is an instance of a well-known security condition, *observational determinism* (OD). We show here how applying our approach can avoid the need for semantic behavioral reasoning, allowing a purely state-based proof.

A. Observational Determinism

Zdancewic and Myers [34] formulated OD as the assertion that any two system behaviors with the same initial value of $public$ are "equivalent". Equivalent would mean $public$-stuttering equivalent[4] if every behavior took the same number (possibly ∞) of public steps. That a system P satisfies OD would then be expressed by:

$$\begin{aligned} \models\; & P(\mathbf{x_1}) \,\wedge\, P(\mathbf{x_2}) \,\wedge\, (public(\mathbf{x_1}) = public(\mathbf{x_2})) \\ & \Rightarrow \boldsymbol{\exists}\, \widehat{\mathbf{x_1}}, \widehat{\mathbf{x_2}} \,:\, \mathbf{x_1} \sim \widehat{\mathbf{x_1}} \,\wedge\, \mathbf{x_2} \sim \widehat{\mathbf{x_2}} \,\wedge \\ & \qquad\qquad \Box(public(\widehat{\mathbf{x_1}}) = public(\widehat{\mathbf{x_2}})) \end{aligned} \tag{35}$$

Zdancewic and Myers consider only finite behaviors, for which they define equivalence to mean that the sequence of public steps of one of the behaviors is a prefix of the sequence of public steps of the other. The easiest way to express this condition in TLA is to posit a state predicate $term$ that is true iff the system has terminated—that is, iff the system can take no more state-changing steps. In that case, OD is obtained from (35) by replacing the shaded formula with

$$term(\widehat{\mathbf{x_1}}) \,\vee\, term(\widehat{\mathbf{x_2}}) \,\vee\, (public(\widehat{\mathbf{x_1}}) = public(\widehat{\mathbf{x_2}}))$$

B. The PharOS Proof

If we define $public$ to be the state $state[a]$ of an agent a, then determinacy for PharOS asserts that OD is satisfied for every agent a. Azaiez et al. proved this condition for an arbitrary agent a. They described PharOS in TLA$^+$ and checked their proof with TLAPS.

For their proof, they added to the TLA$^+$ system description an auxiliary variable whose value is the sequence of all previous system states. They proved that if b is the sequence of states of an arbitrary infinite PharOS behavior, then the values of $state[a]$ recorded in the auxiliary variable for a system behavior with the same initial state as b is always $state[a]$-stuttering equivalent to their values in some finite prefix of b.

In their proof of OD, b is a constant—a representation of a complete, infinite behavior. To define b, they wrote a constant formula (one containing no system variables) that captures the semantics of the system's TLA$^+$ specification. Theirs is thus a "hybrid" proof, combining TLA reasoning with semantic behavioral reasoning.

The description of PharOS allows terminating behaviors. We can handle terminating agents using $term$ as described above, but there's no need. The sequence of an agent's steps of a terminating behavior of PharOS is a prefix of its steps in a nonterminating behavior, so satisfying OD for nonterminating behaviors implies that OD is satisfied for terminating behaviors. We simplify the proof by assuming a fairness condition that requires agents never to terminate.

With this non-termination assumption, we can verify that PharOS satisfies OD the same way we verified that *Little* satisfies GNI. Neither semantic reasoning nor auxiliary variables are required. We verified that *Little* satisfies (23) by applying rule (24) to show that it suffices to verify (31). In the same

[4] Recall that $public$-stuttering equivalent is defined in Section V-A.

way, verifying that PharOS satisfies (35) can, by applying (24), be reduced to verifying

$$\models P(\mathbf{x_1}) \wedge P(\mathbf{x_2}) \wedge (public(\mathbf{x_1}) = public(\mathbf{x_2})) \\ \wedge \Box[Pub(\mathbf{x_1}) \equiv Pub(\mathbf{x_2})]_{\langle \mathbf{x_1}, \mathbf{x_2} \rangle} \\ \Rightarrow \Box(public(\mathbf{x_1}) = public(\mathbf{x_2})) \tag{36}$$

where P is the TLA$^+$ model of PharOS and Pub describes the steps taken by the given agent. Just as in the verification that $Little$ satisfies GNI, the left-hand side of (36) can be rewritten as a TLA description of a state machine. Verification then becomes the standard problem of verifying that a formula is an invariant of a state machine. This can be done without constructing a complete behavior or adding an auxiliary variable as in [4].

VIII. Some Other Hyperproperties

GNI and OD are just two of the security conditions discussed in the literature that are hyperproperties. We now consider how a few more security conditions and some other hyperproperties can be expressed in TLA. All other finitary hyperproperties we have seen can be handled in similar ways. Unlike GNI and OD, the examples considered here do not use the $\sim$ operator.

A. Nonin(ter)ference

GNI was preceded by a security policy called *noninterference* (NI) proposed by Goguen and Meseguer [17] as a condition on execution by two classes of users. NI was stated in terms of an automaton that executes commands, some of which belong to a set PC of public commands. The value of a state function we will call $public$ equals the output of the most recently executed public command. We formulate NI as a state machine with a fixed initial state and a state function cmd equal to the name of the most recent command.

NI asserts that executing any sequence of commands produces the same values of $public$ as executing the subsequence consisting of only the commands in PC. Goguen and Meseguer assumed commands are deterministic, meaning that any sequence of commands produces a unique execution. This assumption allows us to state NI as the following two equivalent conditions, where K is the assertion that behavior b_2 executes the subsequence of the commands executed by behavior b_1 consisting only of commands in PC:[5]

- Every pair of system behaviors b_1 and b_2 that satisfy K produce the same values of $public$.
- For every system behavior b_1 there exists a behavior b_2 satisfying K that produces the same values of $public$ as b_1.

These two conditions on behaviors yield different TLA formulations of what it means for a system P to satisfy NI:

$$\models P(\mathbf{x_1}) \wedge P(\mathbf{x_2}) \wedge K \Rightarrow \Box(public(\mathbf{x_1}) = public(\mathbf{x_2})) \tag{37}$$

[5] K is defined by:

$$\xi(\mathbf{x}) \triangleq \langle \mathbf{x}' \rangle \neq \langle \mathbf{x} \rangle$$
$$K \triangleq \Box[\ (\xi(\mathbf{x_1}) \wedge (cmd(\mathbf{x_1})' \in PC) \Rightarrow \xi(\mathbf{x_2})) \\ \wedge ((\xi(\mathbf{x_2}) \Rightarrow \xi(\mathbf{x_1}) \wedge (cmd(\mathbf{x_2})' = cmd(\mathbf{x_1})') \\ \wedge (cmd(\mathbf{x_2})' \in PC))\ ]_{\langle \mathbf{x_1}, \mathbf{x_2} \rangle}$$

$$\models P(\mathbf{x_1}) \Rightarrow \\ \boldsymbol{\exists}\, \mathbf{x_2} : P(\mathbf{x_2}) \wedge K \wedge \Box(public(\mathbf{x_1}) = public(\mathbf{x_2})) \tag{38}$$

They are equivalent under the assumption that commands are deterministic, but differ when commands are nondeterministic. Condition (37) more closely resembles Goguen and Meseguer's original formulation of NI, while (38) generalizes to handle nondeterministic commands.

Note that the $\sim$ operator is not needed in (37) because K asserts that the films $\mathbf{x_1}$ and $\mathbf{x_2}$ are properly aligned. It is not needed in (38) because K implies that $\mathbf{x_1}$ and $\mathbf{x_2}$ can be aligned by adding frames to $\mathbf{x_2}$, which is allowed by the $\boldsymbol{\exists}$ operator. The $\sim$ operator was needed in GNI (23) and OD (35) to allow replacing the "films" $\mathbf{x_1}$ and $\mathbf{x_2}$ with films $\widehat{\mathbf{x_1}}$ and $\widehat{\mathbf{x_2}}$ of the same executions, but properly aligned.

Noninference (NF) is a security condition that generalizes NI to allow nondeterministic commands. Mantel stated a version of NF in terms of event sequences [25]. His version can be represented in terms of states the way we represented GNI, where an event is represented by a state change. We add a state function $secret$ whose values are changed by executing commands not in PC. All commands in a behavior being commands in PC is then equivalent to $secret$ having the same value throughout the behavior. Mantel's version of NF is then described by (38) when K is the assertion that $secret(\mathbf{x_2})$ never changes, expressed in TLA as:

$$\Box[secret(\mathbf{x_2})' = secret(\mathbf{x_2})]_{\langle \mathbf{x_2} \rangle}$$

This version of (38) is satisfied by $Little$, but not by $Tiny$.

McClean [27] proposed a version of NF in terms of state sequences that can also be expressed in terms of the state function $secret$. It is obtained from (38) by replacing K with the assertion that $secret(\mathbf{x_2})$ always equals a fixed constant λ—an assertion expressed in TLA as $\Box(secret(\mathbf{x_2}) = \lambda)$.

B. Possibilistic Noninterference

Zdancewic and Myers [34] formulate a generalization of noninterference to handle non-deterministic commands; we call it *possibilistic noninterference* (PN). They expressed PN in a state-based model with a "public state" described by a state function $public$. PN is satisfied by a system iff, for every possible system behaviors b_1 and b_2 such that $public$ has the same value in the initial states of b_1 and b_2, there is a system behavior b_3 having the same initial state as b_2 and the same values of $public$ as b_1 in all states.

Zdancewic and Myers's definition of PN is based on a model in which a state sequence represents an execution rather than a film of an execution. For a clock in which $public$ is the value of the hour and minute, in this model observing only $public$ reveals that the clock is also counting seconds because that same value of $public$ appears in multiple successive states. Even though this definition is based on a model that is not SI, we can write a (SI) TLA formula asserting that a system satisfies it by restricting how the system is described.

The restriction is that for a system step to be considered observable, it must change the value of some state function. For PN, this means that the sequences of values for $public$ can differ in two behaviors because of changes only to a variable that doesn't affect the value of *public*. We can then represent the definition of PN with behaviors b_1, b_2, and b_3 that represent films by adding the requirement that b_1 and b_3 are aligned so that their states change at the same time. (There is no need to align b_2 with b_1 and b_3 because only the initial state of b_2 is mentioned in the definition, so no further alignment is required and the $\sim$ operator is not needed.) The assertion that the system P satisfies PN is then:

$$\begin{aligned}\models P(\mathbf{x_1}) \land P(\mathbf{x_2}) \land (\mathit{public}(\mathbf{x_1}) = \mathit{public}(\mathbf{x_2})) \\ \Rightarrow \exists\, \mathbf{x_3} : P(\mathbf{x_3}) \land K \land (\langle \mathbf{x_2} \rangle = \langle \mathbf{x_3} \rangle) \\ \land \Box(\mathit{public}(\mathbf{x_1}) = \mathit{public}(\mathbf{x_3}))\end{aligned} \tag{39}$$

where the alignment condition K is defined by

$$K \triangleq \Box[(\langle \mathbf{x_1} \rangle' \neq \langle \mathbf{x_1} \rangle) \equiv (\langle \mathbf{x_3} \rangle' \neq \langle \mathbf{x_3} \rangle)]_{\langle \mathbf{x_1}, \mathbf{x_3} \rangle}$$

It is not hard to see that *Tiny* and *Little* both satisfy (39). Given behaviors b_1 and b_2 of either system, the behavior b_3 obtained by simply replacing the first state of b_1 with the first state of b_2 is also a behavior of that system. To verify (39) for these two systems, we expand the definitions of P and verify:

$$\begin{aligned}\models Q(\mathbf{x_1}, \mathit{nin}_1) \land Q(\mathbf{x_2}, \mathit{nin}_2) \\ \land (\mathit{public}(\mathbf{x_1}) = \mathit{public}(\mathbf{x_2})) \\ \Rightarrow \exists\, \mathbf{x_3} : P(\mathbf{x_3}) \land K \land (\langle \mathbf{x_2} \rangle = \langle \mathbf{x_3} \rangle) \\ \land \Box(\mathit{public}(\mathbf{x_1}) = \mathit{public}(\mathbf{x_3}))\end{aligned} \tag{40}$$

We verify this by adding an auxiliary variable h to $Q(\mathbf{x_1}, \mathit{nin}_1)$ to obtain Q^h such that $Q(\mathbf{x_1}, \mathit{nin}_1)$ is equivalent to $\exists\, h : Q^h(\mathbf{x_1}, \mathit{nin}_1, h)$ and then verifying:

$$\begin{aligned}\models Q^h(\mathbf{x_1}, \mathit{nin}_1, h) \land Q(\mathbf{x_2}, \mathit{nin}_2) \\ \land (\mathit{public}(\mathbf{x_1}) = \mathit{public}(\mathbf{x_2})) \\ \Rightarrow \exists\, \mathbf{x_3} : P(\mathbf{x_3}) \land K \land (\langle \mathbf{x_2} \rangle = \langle \mathbf{x_3} \rangle) \\ \land \Box(\mathit{public}(\mathbf{x_1}) = \mathit{public}(\mathbf{x_3}))\end{aligned}$$

We can let h equal 1 in the initial state and be set to 0 by the next-state action of Q^h. The refinement mapping is defined so that the values of variables $\mathbf{x_3}$ equal the values of $\mathbf{x_2}$ if $h = 1$ and the values of $\mathbf{x_1}$ if $h = 0$.

Tiny satisfies (39), but that doesn't mean it satisfies PN. Formula (39) represents PN only under the assumption that every observable step changes the system's state, and *Tiny* allows steps we consider observable that change only *nin*—steps that represent input or output of the same value twice in a row—and *nin* is a hidden variable, not part of the system state. What satisfying (39) means in this case does not concern us.

C. Input/Output Hyperproperties

Besides describing security conditions, hyperproperties have been used to express relations between the input and output of a system that starts with an input, produces an output, and halts. For example, monotonicity is a hyperproperty asserting that if the input of behavior b_1 is less than the input of b_2, then the output of b_1 is less than that of b_2.

In state-based representations of systems, such input/output relations can be expressed in terms of state functions *inp* and *outp*, where the input is the value of *inp* in the initial state and the output is the value of *outp* in the final state. Letting *term* be a state predicate that is true iff the system has terminated, monotonicity for a system P is expressed as:

$$\begin{aligned}\models P(\mathbf{x_1}) \land P(\mathbf{x_2}) \land (\mathit{inp}(\mathbf{x_1}) < \mathit{inp}(\mathbf{x_2})) \Rightarrow \\ \Box(\mathit{term}(\mathbf{x_1}) \land \mathit{term}(\mathbf{x_2}) \Rightarrow (\mathit{outp}(\mathbf{x_1}) < \mathit{outp}(\mathbf{x_2})))\end{aligned}$$

TLA provides a good way for verifying such a condition, especially if the system P involves concurrency. The $\sim$ operator does not appear because this condition involves only initial and terminal states, so no alignment of the films is required.

D. Some Problematic Security Conditions

Most of the examples of hyperproperties we have examined concern security. We know of only one class of security conditions for which the TLA formulation is significantly more complicated than the ones described here. The conditions in that class stipulate that adding one or more events to the middle of a system execution produces a possible system execution. One example is the *perfect security property* (PSP) defined by Zakinthinos and Lee [33]. Expressing such a condition by replacing events with command executions, as in NI, is not hard. A TLA statement of PSP asserts the existence of a variable whose value indicates when the extra commands are being added. However, it might be easier to state and verify the condition by using auxiliary variables, as was done in the original PharOS verification.

IX. Preservation Under Refinement

If we verify that a system P satisfies a hyperproperty and P is refined by another system S, then we would like S also to satisfy that hyperproperty. When that is the case for all S and P, we say that the hyperproperty is *preserved under refinement*.

Thus far, the systems and the properties they satisfy have been expressed in terms of the same (free) variables. This makes refinement the same as implementation: A system S *refines* a system P iff S implies P, which means the set of behaviors allowed by S is a subset of the set allowed by P. Whether a hyperproperty is preserved under refinement can be seen from its definition. A hyperproperty described in the form of (1) is preserved under refinement if every $\forall\exists$ is $\forall$. When described as in (7), that means $k = j$, so P does not appear to the right of the $\Rightarrow$. This is the case handled by previous work using self-composition. The special case $k = j = 1$ implies that ordinary properties are preserved under refinement, since P satisfying property L means $\models P(\mathbf{x}) \Rightarrow L(\mathbf{x})$. When $k > j$, the most we can say is that P satisfying (7) implies that S also satisfies (7) if S and P are equivalent.

In practice, we often want to show that a system P is refined by a system S described at a lower level of abstraction, so P and S can have different free variables. For example, P might describe characters displayed on a screen, and S might describe the screen as an array of pixels. It makes no sense to say that a statement about pixels refines a statement about characters. What does make sense is to say that S refines P

under a given correspondence between pixels and characters. A more complex example is if P describes a system in which processes communicate by sending messages over point-to-point channels, while S splits those messages into packets that are sent over a packet-switching network.

The idea that a system S refines a system P described at a higher level of abstraction is expressed formally using an *interface refinement*, which is a property relating the (free) variables of S and those of P. We define S refines P under the interface refinement I to mean

$$\models S(\mathbf{w}) \wedge I(\mathbf{w},\mathbf{x}) \;\Rightarrow\; P(\mathbf{x}) \tag{41}$$

where I must satisfy:

$$\models S(\mathbf{w}) \;\Rightarrow\; \boldsymbol{\exists}\,\mathbf{x} : I(\mathbf{w},\mathbf{x}) \tag{42}$$

Condition (42) asserts that every behavior of S corresponds under I to some behavior, and (41) asserts that it is a behavior of P.[6]

In general, I may be written as a state machine. As we have seen, the conjunction of two state machines can be written as a state machine, so verifying (41) reduces to the problem of one state machine implying another. A simple instance is when $I(\mathbf{w},\mathbf{x})$ is $\Box(\mathbf{x} = \mathbf{g}(\mathbf{w}))$, in which case (41) is equivalent to

$$\models S(\mathbf{w}) \Rightarrow P(\mathbf{g}(\mathbf{w})) \tag{43}$$

and we say S refines P under interface refinement mapping $\mathbf{g}$.

Mathematically, (43) is the same condition that arises if the variables of P are regarded as hidden and we are given the refinement mapping $\mathbf{g}$ under which $S(\mathbf{w})$ must imply $\boldsymbol{\exists}\,\mathbf{x} : P(\mathbf{x})$. This form of I handles the example of refining a screen that displays characters with one displaying pixels, where $\mathbf{g}(\mathbf{w})$ specifies the screen of characters that corresponds to the screen of pixels described by $\mathbf{w}$. However, I would probably have to be a state machine for the example of refining messages by packets.

It would be nice if all hyperproperties were preserved under interface refinement. If P satisfies (7), we would like (41) and (42) to imply that S does too. However, since S and P may have different free variables, we can't use the same formulas K and L in (7) for S as for P. We have to specify the formulas K_S and L_S for which S should satisfy (7).

For refinement under an interface refinement mapping $\mathbf{g}$, there are natural candidates for K_S and L_S:

$$\begin{aligned} K_S(\mathbf{w_1},\ldots,\mathbf{w_j}) &\;\triangleq\; K(\mathbf{g}(\mathbf{w_1}),\ldots,\mathbf{g}(\mathbf{w_j})) \\ L_S(\mathbf{w_1},\ldots,\mathbf{w_k}) &\;\triangleq\; L(\mathbf{g}(\mathbf{w_1}),\ldots,\mathbf{g}(\mathbf{w_k})) \end{aligned}$$

When $k = j$, if P satisfies (7) then (43) implies that S satisfies (7) with these definitions of K_S and L_S. However, these natural definitions of K_S and L_S might not be useful definitions. For example, P satisfying GNI says something useful about a system's security only if the values of *secret* and *public* together specify the values of all the free variables of P. However, the values of $secret(\mathbf{x})$ and $public(\mathbf{x})$ can specify the values of the free variables $\mathbf{x}$ of $P(\mathbf{x})$ without $secret(\mathbf{g}(\mathbf{w}))$ and $public(\mathbf{g}(\mathbf{w})))$, which appear in K_S and L_S, specifying the values of the free variables $\mathbf{w}$ of $S(\mathbf{w})$.

For arbitrary K_S and L_S, the assumptions needed to conclude from P, K, L satisfying (7) that S, K_S, and L_S satisfy it are (42) and:

$$\models I(\mathbf{w},\mathbf{x}) \;\Rightarrow\; (S(\mathbf{w}) \equiv P(\mathbf{x}))$$

$$\models P(\mathbf{x}) \;\Rightarrow\; \boldsymbol{\exists}\,\mathbf{w} : I(\mathbf{w},\mathbf{x})$$

$$\begin{aligned} \models\; & S(\mathbf{w_1}),\ldots,S(\mathbf{w_j}) \wedge K_S(\mathbf{w_1},\ldots,\mathbf{w_j}) \\ & \wedge I(\mathbf{w},\mathbf{x_1}) \wedge \ldots \wedge I(\mathbf{w},\mathbf{x_j}) \;\Rightarrow\; K(\mathbf{x_1},\ldots,\mathbf{x_j}) \end{aligned}$$

$$\begin{aligned} \models\; & P(\mathbf{x_1}) \wedge \ldots \wedge P(\mathbf{x_k}) \wedge K(\mathbf{x_1},\ldots,\mathbf{x_j}) \wedge L(\mathbf{x_1},\ldots,\mathbf{x_k}) \\ & \wedge I(\mathbf{w_1},\mathbf{x_1}) \wedge \ldots \wedge I(\mathbf{w_j},\mathbf{x_k}) \;\Rightarrow\; L_S(\mathbf{w_1},\ldots,\mathbf{w_k}) \end{aligned}$$

For the special case of (7) with $j = k$, we can replace the shaded conditions with (41).

X. Discussion

A. Prior Work

Prior work has used temporal logic to verify that systems satisfy security conditions without expressing the conditions as hyperproperties. Huisman et al. [18] formulated observational determinism in both CTL* and the polyadic modal μ-calculus. They experimented with model checkers for both logics. Alur et al. [3] defined a class of trees that are suitable for capturing observational indistinguishability. Information flow properties can be described using temporal logics, including CTL and the μ-calculus, interpreted on these trees. Algorithms for model checking formulas in these logics were also given. Finkbeiner et al. [12] defined new logics by adding a modal operator to characterize certain information flows. They explored the complexity of model checking these logics and developed a fragment of one logic that is both expressive enough to describe non-interference and observational determinism and for which model checking is efficient. Balliu [5] used a linear time temporal epistemic logic with a past operator to express information flow properties, including GNI. TLA has also been used to verify that a system satisfies a particular hyperproperty. PharOS (Section VII) was one example; Wayne [32] also independently used TLA in this way.

Clarkson et al. [9] were the first to introduce a temporal logic for describing a general class of hyperproperties. Their linear-time logic, HyperLTL, expresses finitary hyperproperties, as described by (1). They built a prototype model checker based on nondeterministic Büchi automata for a subset of HyperLTL formulas that includes $\forall\exists$-hyperproperties. It was improved using alternating Büchi automata by Finkbeiner et al. [14] with the MCHyper model checker. These model checkers for HyperLTL are completely automatic, but the inherent complexity of handling temporal existential quantification means that they are not practical for hyperproperties described by instances of (7) actually containing an $\boldsymbol{\exists}$ (i.e., when $k > j$). Coenen et al. [11] enhanced MCHyper to handle $\forall\exists$-hyperproperties more efficiently, based on a game-theoretic metaphor. In effect, they partially automated construction of the

[6]Broy [8] proposed an equivalent formalization of interface refinement in terms of event streams.

refinement mappings used by TLA; complete automation was also possible in some cases. More efficient model checkers can also be built to handle specialized classes of hyperproperties efficiently. For example, Finkbeiner et al. [13] built one for a particular class called quantitative hyperproperties.

B. Contributions

Prior work on verifying hyperproperties using self-composition handled hyperproperties of the form (3). One of our contributions is using self-composition to handle arbitrary finitary hyperproperties. This is feasible because TLA can easily describe a system as a formula. Given a refinement mapping for each $\exists$, we can verify the TLA formula expressing that a system satisfies an arbitrary finitary hyperproperty. Moreover, we know that refinement mappings can be found for instances of (7) that seem to arise in industry. We have no experience with the TLA formulas that arise for other classes of finitary hyperproperties, and we haven't seen any realistic examples of such hyperproperties.

Another contribution is the observation that stuttering insensitivity (SI) facilitates the treatment of security conditions in a state-based formalism. It has long been known that SI simplifies verifying implementation, so an hour-minute-second clock naturally implements an hour-minute clock. For that purpose, SI could have been avoided by requiring systems to allow explicitly described stuttering steps and considering those additional behaviors to be additional executions. But formulating event-based definitions of GNI and some other security conditions in terms of states led us to define the temporal operator $\sim$, and we could write a simple rule for reasoning about $\sim$ only because TLA satisfies SI. This provides further evidence for the value of SI in formalisms for describing systems.

Perhaps our most important contribution is showing how tools that have been developed through two decades of industrial experience can be used to verify that systems satisfy a large class of hyperproperties. TLA$^+$ and its tools have been used in the design and verification (mainly by model checking) of systems ranging from multi-core processor chip caches [7] to real-time operating systems [31] to large-scale cloud infrastructure [28]. This provides reason to hope that the approach we have described can work for real systems.

Acknowledgments

Michael Clarkson and Bernd Finkbeiner provided helpful explanations of aspects of HyperLTL and its model checkers. Comments by the CSF reviewers helped us improve the presentation. Stephan Merz helped us understand the PharOS proof; he also pointed out small errors and statements that needed clarification throughout the paper. This work resulted from the authors' attendance at SATIS '18 in Sommarøy, Norway.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[3] Rajeev Alur, Pavol Cerný, and Swarat Chaudhuri. Model checking on trees with path equivalences. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 664–678. Springer, 2007.

[4] Selma Azaiez, Damien Doligez, Matthieu Lemerre, Tomer Libal, and Stephan Merz. Proving determinacy of the PharOS real-time operating system. In Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklós Biró, editors, *5th Intl. Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2016)*, volume 9675 of *LNCS*, pages 70–85. Springer, 2016.

[5] Musard Balliu. A logic for information flow analysis of distributed programs. In Hanne Riis Nielson and Dieter Gollmann, editors, *Secure IT Systems—18th Nordic Conference, NordSec*, volume 8208 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2013.

[6] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17), 28-30 June 2004, Pacific Grove, CA, USA*, pages 100–114.

[7] Brannon Batson and Leslie Lamport. High-level specifications: Lessons from industry. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, Leiden, The Netherlands, Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*, pages 242–261. Springer, 2002.

[8] Manfred Broy. Compositional refinement of interactive systems. *Journal of the ACM*, 44(6):850–891, November 1997.

[9] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust*, pages 265–284. Springer Berlin Heidelberg, 2014.

[10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[11] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. *CoRR*, abs/2005.07425, 2020. https://arxiv.org/abs/2005.07425.

[12] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation—13th International Conference, VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 169–185. Springer, 2012.

[13] Bernd Finkbeiner, Christopher Hahn, and Hazem Torfah. Model checking quantitative hyperproperties. *CoRR*, abs/1905.13514, 2019. http://arxiv.org/abs/1905.13514.

[14] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification—27th International Conference, CAV*, volume 9206 of *Lecture Notes in Computer Science*, pages 30–48. Springer, 2015.

[15] Nissim Francez. Product properties and their direct verification. *Acta Informatica*, 20:329–344, 1983.

[16] Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1986.

[17] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[18] Marieke Huisman, Pratik Worah, and Kim Sunesen. A temporal logic characterisation of observational determinism. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*, page 3. IEEE Computer Society, 2006. https://doi.org/10.1109/CSFW.2006.6.

[19] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[20] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003. Also available on the Web via a link at http://lamport.org.

[21] Leslie Lamport and Stephan Merz. Prophecy made simple. http://lamport.azurewebsites.net/pubs/simple.pdf.

[22] Leslie Lamport and Fred B. Schneider. Specifications for Verifying Hyperproperties with TLA. Web page. https://lamport.azurewebsites.net/tla/hyperproperties/hyper.html.

[23] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Symposium on the*

Principles of Distributed Computing, pages 137–151. ACM, August 1987.

[24] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems—safety*. Springer, 1995.

[25] Heiko Mantel. Possibilistic definitions of security—An assembly kit. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3-5, 2000*, pages 185–199. IEEE Computer Society.

[26] Daryl McCullough. Noninterference and the composability of security properties. In *Proceedings of the 1988 IEEE Conference on Security and Privacy*, SP'88, pages 177–186.

[27] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *1994 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, May 16-18, 1994*, pages 79–93.

[28] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, April 2015.

[29] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.

[30] Marcelo Sousa and Isil Dillig. Cartesian Hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 57–69, 2016.

[31] Eric Verhulst, Raymond T. Boute, José Miguel Sampaio Faria, Bernard H. C. Sputh, and Vitaliy Mezhuyev. *Formal Development of a Network-Centric RTOS*. Springer, New York, 2011.

[32] Hillel Wayne. Hypermodeling hyperproperties. Web page: https://www.hillelwayne.com/post/hyperproperties/.

[33] Aris Zakinthinos and E. S. Lee. A general theory of security properties. In *Proceedings 1997 IEEE Symposium on Security and Privacy*, pages 94–102.

[34] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW-16)*, pages 29–43, 2003.