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 \( H \) involving \( k \) behaviors. To assert that the system satisfies \( H \), 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: TLA@cs.cornell.edu. Supported, in part, by AFSOR 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 $I$ on states and a next-state predicate $N$ on pairs of states. (The set of states need not be finite.) The behaviors generated by the state machine are ones in which predicate $I$ is true on the first state of the sequence and predicate $N$ 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:

$$I_{hm} \triangleq (min = 0) \land (hr = 12)$$

The clock’s next-state predicate $N$ 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:

$$N_{hm} \triangleq \begin{cases} 
\min' = (\min + 1) \mod 60 \\
\land hr' = \text{IF } \min = 59 \\
\text{THEN IF } hr = 12 \text{ THEN } 1 \\
\text{ELSE } hr + 1 
\end{cases}$$

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 $S \models \text{Term}$, where $b \models \text{Term}$ is true 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.

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, $\lor$ 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, and most often a fairness property. 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, so they were generalized to hyperproperties. 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 $H$ where, for a property $P$, we define $H(P)$ 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 $H$ to be a finitary hyperproperty iff $H(P)$ 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

---

1We 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

\[
\forall \exists b_1 \in P : \ldots \forall \exists b_k \in P : J(b_1, \ldots, b_k)
\]

(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:

\[
\forall b \in P : J(b)
\]

(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 \( \forall \).\(^6\) 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

\[
\forall b \in P^k : J(\pi_1(b), \ldots, \pi_k(b))
\]

(3)

where \( \pi_i \) 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:

\[
\forall b_1 \in P : \ldots \forall b_j \in P : K(b_1, \ldots, b_j) \Rightarrow \exists b_{j+1} \in P : \ldots \exists b_k \in P : L(b_1, \ldots, b_k)
\]

(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 “\( \exists b : P(b) \land \)” . Doing that and applying a bit of predicate logic, (4) becomes:

\[
\forall b_1, \ldots, b_j : P(b_1) \land \ldots \land P(b_j) \land K(b_1, \ldots, b_j) \Rightarrow \exists b_{j+1}, \ldots, b_k : P(b_{j+1}) \land \ldots \land P(b_k) \land L(b_1, \ldots, b_k)
\]

(5)

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 \( \square \) (read always or henceforth) is defined by letting \( b \models \square F \) be true iff \( c \models F \) is true for \( c \) equal to \( b \) and all prefixes of \( b \). Thus, \( b \models \square(x > y) \) 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 \( \exists \) over variables [24], where \( b \models \exists x : F \) asserts that there exists a behavior \( \hat{b} \) that is the same as \( b \), except that the values of \( x \) in the states of \( \hat{b} \) and \( b \) may differ, such that \( \hat{b} \models F \) 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 \( \exists x : F \) 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 \( \exists \) obeys all the rules that the quantifier \( \forall \) of predicate logic does.

Formulas \( \exists x : F(x) \) and \( \exists y : F(y) \) 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 \( \exists x : F(x) \) 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 \( \hat{P} \) 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 \( \hat{P} \) by substituting new variables for \( v_1, \ldots, v_n \) — a different set of variables for each \( i \).

\(^2\)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 \( P \).

\(^3\)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 \( \tilde{P} \) 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 \( \tilde{P}(x_1, \ldots, x_n) \). Moreover, we let \( x \) be an abbreviation for \( x_1, \ldots, x_n \), so we can write the formula as \( \tilde{P}(x) \); and we do the same for other boldface identifiers. We also let \( x_i \) 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 \( \tilde{P}(x_i) \). The values that \( b \) assigns to the variables \( x_i \) 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 \( \bar{K}(x_1, \ldots, x_j) \) and \( \bar{L}(x_1, \ldots, x_k) \). We can then write (5) as

\[
\models \tilde{P}(x_1) \land \ldots \land \tilde{P}(x_j) \land \bar{K}(x_1, \ldots, x_j)
\]

(6)

\[
\Rightarrow \exists x_{j+1}, \ldots, x_k : \tilde{P}(x_{j+1}) \land \ldots \land \tilde{P}(x_k) \land \bar{L}(x_1, \ldots, x_k)
\]

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

\[
\models P(x_1) \land \ldots \land P(x_j) \land K(x_1, \ldots, x_j)
\]

(7)

\[
\Rightarrow \exists x_{j+1}, \ldots, x_k : P(x_{j+1}) \land \ldots \land P(x_k) \land L(x_1, \ldots, x_k)
\]

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 \( \square \), described above. The logic could not express many properties of interest, so additional temporal operators were subsequently proposed, including \( \exists \) (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 \( S \models \diamond P \), 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 \( N_{hm} \) 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 \( \square, \exists \), and operators defined in terms of them.

Instead of explaining TLA directly, we begin with the slightly simpler logic RTLA. It contains the operators \( \square \) and \( \exists \) defined above. An action, interpreted as an RTLA formula, is true of a behavior if and only if it is true of the first step of the behavior. So, the definition of \( \square \) implies that \( b \) is a possible behavior of the state machine described by the initial predicate \( I \) and the next-state predicate \( N' \) if \( b \models I \land \square N' \) is true, since \( b \models I \) asserts that the first state of \( b \) satisfies \( I \), and \( b \models \square N' \) asserts that the first step of every suffix of \( b \) satisfies \( N' \). (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 \( N' \) step is possible. It is written \( WF(N') \).

E. Hiding

It would be impossible to express even rather simple temporal properties in RTLA without the operator \( \exists \). For example, consider the property that is true of a behavior if and only if 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 \( \square \). However, it’s easy to write that property as follows using a (Boolean-valued) hidden variable \( y \):

\[
\exists y : \ (y = (x = 42)) \land \square((\neg y \Rightarrow (x' \neq 1)) \land (y' = (y \lor (x = 42))))
\]

This property has the form \( I \land \square N' \) 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 \( \exists \), so \( \models (S \Rightarrow P) \) can have the form

\[
(\exists y : S(x, y)) \Rightarrow (\exists z : P(x, z))
\]

(8)

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

\[
S(x, y) \Rightarrow (\exists z : P(x, z))
\]

(9)

which asserts that, for any behavior satisfying \( S(x, y) \), we can find assignments of values to the variables \( z \) in each state of the behavior that makes \( P(x, z) \) true. The value assigned to \( z \) in
any state of the behavior might depend on the values assigned to \( x \) and \( y \) in all the states of the behavior. Verification of (9) becomes much simpler if the assignment of values to \( z \) in any state depends only on the values of \( x \) and \( y \) 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 \( z \) is \( z_1, \ldots, z_n \), we verify (9) by finding state functions \( f_1(x, y), \ldots, f_n(x, y) \) that make this formula true:

\[
\models S(x, y) \Rightarrow P(x, f(x, y)) \tag{10}
\]

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

In (10), we are substituting state functions \( f(x, y) \) for the variables \( z \). 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 \( f \) 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 \( a \) to \( S(x, y) \) means finding a formula \( S^a(x, y, a) \) that is equivalent to \( S(x, y) \) when the variables \( a \) are hidden—that is, where \( \exists a : S^a(x, y, a) \) is equivalent to \( S(x, y) \) [21]. We can then verify (9) by verifying:

\[
\models S^a(x, y, a) \Rightarrow P(x, f(x, y, a))
\]

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:

\[
\forall b_1, b_2 : P(b_1) \land P(b_2) \Rightarrow P(b_3) \land L(b_1, b_2, b_3) \tag{11}
\]

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:

\[
\models P(x_1) \land P(x_2) \Rightarrow \exists x_3 : P(x_3) \land L(x_1, x_2, x_3) \tag{12}
\]

where \( L(x_1, x_2, x_3) \triangleq \Box (\text{public}(x_3) = \text{public}(x_1) \land \text{secret}(x_3) = \text{secret}(x_2) \)

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 \( x_i \) 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 WF(\( N \)).

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 \( I_j, \ldots, secret_j \) be the formulas obtained from the formulas \( I, \ldots, secret \) defined in Figure 1
Tiny is a tiny finite-state system, and it should be easy to verify with TLC, the TLA model checker. 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.

\[ I \triangleq \ \text{in} \in \text{Val} \]
\[ \land \ \text{out} \in \text{Val} \]
\[ \land \ \text{nin} = 0 \]
\[ N \triangleq \text{Pub} \lor \text{Sec} \]
where \[ \text{Pub} \triangleq \ \text{nin} = 0 \land \text{nin'} = 1 \]
\[ \land \ \text{out} \in \text{Val} \]
\[ \land \ \text{in'} = \text{in} \]
\[ \text{Sec} \triangleq \ \text{nin} = 1 \land \text{nin'} = 0 \]
\[ \land \ \text{in'} \in \text{Val} \]
\[ \land \ \text{out} = \text{out} \]
\[ L \triangleq \text{WF}(N) \]
\[ Q \triangleq I \land \Box N \land L \]
\[ P \triangleq \exists \text{nin} : Q \]
\[ \text{public} \triangleq \text{out} \]
\[ \text{secret} \triangleq \text{in} \]

Fig. 1. The RTLA Description of System Tiny.

by substituting new variables \( in_j, out_j, \text{nin}_j \) for the variables \( in, out, \text{nin} \). With this notation, (12) becomes

\[ \models P_1 \land P_2 \Rightarrow \exists \text{in}_3, \text{out}_3 : \]
\[ P_3 \land \Box ((\text{public}_3 = \text{public}_1) \land (\text{secret}_3 = \text{secret}_2)) \]

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

\[ \models Q_1 \land Q_2 \Rightarrow \exists \text{in}_3, \text{out}_3, \text{nin}_3 : \]
\[ Q_3 \land \Box ((\text{public}_3 = \text{public}_1) \land (\text{secret}_3 = \text{secret}_2)) \]

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

\[ (I_1 \land I_2) \land \Box(N_1 \lor N_2) \land (\text{WF}(N_1) \land \text{WF}(N_2)) \]

(15)

Formula (14) has the form of (9), and the equivalence of \( Q_1 \land Q_2 \) 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 \( \Phi \) 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 \( P_2 \):

\[ \text{in}_3 \leftarrow \text{in}_2 \quad \text{out}_3 \leftarrow \text{out}_1 \quad \text{nin}_3 \leftarrow \text{nin}_2 \]

(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:

\[ \models S_1 \Rightarrow S_2 \]  

(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

\[ S_{hm} \triangleq I_{hm} \land \Box N_{hm} \]  

(18)

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

\[ \models S_{hms} \Rightarrow S_{hm} \]  

(19)

However, (19) is invalid. A behavior satisfying \( S_{hms} \) must take 59 steps that change only \( \text{scd} \) between steps that change \( \text{min} \), but those \( \text{scd} \)-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 \( \text{scd} \). 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 \( \text{scd} \) 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 \( N_{hm} \), formula \( S_{hm} \) should permit steps that allow other variables, including \( \text{scd} \), to change but leave \( hr \) and \( min \) unchanged. Those
additional steps satisfy \((hr' = hr) \land (min' = min)\). 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 \(S_{hm}\):

\[
S_{hm} \triangleq \mathcal{I}_{hm} \land \Box (N_{hm} \lor \Box (\langle hr, min \rangle = \langle hr, min \rangle)) \tag{20}
\]

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 \((hr \text{ and } min)\) 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:

\[
1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, \ldots
\]

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 \(F(x)\) with free variables \(x\) is SI iff, for any two behaviors \(b_1\) and \(b_2\) that are \(\langle x \rangle\)-stuttering equivalent, \(b_1 \models F(x)\) is equivalent to \(b_2 \models F(x)\).

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 \(x\), then two behaviors \(b_1\) and \(b_2\) are films of the same execution iff they are \(\langle x \rangle\)-stuttering equivalent. Therefore, a formula \(F(x)\), which is a predicate on behaviors, is an assertion about executions and not just about particular films of executions iff \(b_1 \models F(x)\) is equivalent to \(b_2 \models F(x)\) for any \(\langle x \rangle\)-stuttering equivalent behaviors \(b_1\) and \(b_2\)—precisely the definition of what it means for \(F(x)\) 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 \(x\) and \(y\), we abbreviate \(\langle x \rangle \sim \langle y \rangle\) as \(x \sim y\). A temporal formula \(F(x)\) is SI iff

\[
\models (x \sim y) \Rightarrow (F(x) = F(y)) \tag{21}
\]

for lists \(x\) and \(y\) 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 \(A \lor (f = f)\) 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

\[
S_{hm} \triangleq \mathcal{I}_{hm} \land \Box [N_{hm} \lor [(hr, min) = (hr, min)]] \tag{22}
\]

The second change to RTLA to ensure SI is to the definition of \(\exists\). In TLA, \(b \models \exists y : F(x, y)\) is defined to be true iff there exists a behavior \(\hat{b}\) that is \(\langle x \rangle\)-stuttering equivalent to \(b\) such that \(\hat{b} \models F\) is true. (Note that \(\hat{b}\) can be obtained from a behavior \(\hat{b}\) that is \(\langle x, y \rangle\)-stuttering equivalent to \(b\) by changing the values of the states of \(\hat{b}\) assign to \(y\).) If \(S_{hm}^\prime\) is redefined to be the TLA formula describing an hour-minute-second clock with the definition analogous to (22), then \(\exists scd : S_{hm}^\prime\) is equivalent to \(S_{hm}\). With the RTLA definition of \(\exists\), formula \(\exists scd : S_{hm}^\prime\) 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 \(S_{hm}^\prime\)) that leave \(hr\) and \(min\) unchanged 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

\[
Q \triangleq \mathcal{I} \land \Box [N]_{(in, out, min)} \land L
\]

Also, the RTLA formula \(WF(N)\) is not SI and must be replaced in the definition of \(L\) by \(WF_{(in, out, min)}(N)\), whose definition can be found elsewhere [20].
We now show that the resulting TLA formula \( P \) does not satisfy (12), where each \( x_i \) is the list \( in_i, out_i \) of variables. Consider a behavior \( b \) in which the variables \( x_1 \) and \( x_2 \) 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, \( Pub(x_3) \) indicates that a step satisfies \( Pub(x_2, \text{nin}) \) for some values of \( n in \) and \( n in' \).

\[
\begin{align*}
\mathbf{x}_1 & : s_1 \quad Pub(x_1) \quad s_2 \quad Sec(x_1) \quad s_3 \quad Pub(x_1) \quad s_4 \quad \ldots \\
\mathbf{x}_2 & : t_1 \quad Pub(x_2) \quad t_2 \quad Sec(x_2) \quad t_2 \quad \ldots 
\end{align*}
\]

The lists \( x_1 \) and \( x_2 \) 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 \( x_2 \) 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 \( x_3 \) representing a behavior \( b_3 \) that satisfies \( P \), where the public part of the state (the value for \( out \)) of \( x_3 \) comes from \( x_1 \) and the secret part (the value for \( in \)) comes from \( x_2 \). But in the third step of the behavior, the variables of \( x_3 \) that represent both \( in \) and \( out \) change, which is not allowed for a step of a behavior of \( Tiny \). Therefore, no such \( x_3 \) 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 \( n in \) to an arbitrary natural number, and letting the \( Sec \) action be enabled when \( n in \neq 0 \) and decrement \( n in \) 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 \( n in \) 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 \( n in \) 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 \( n in \); 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 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 \( x_1 \) and \( x_2 \) representing behaviors \( b_1 \) and \( b_2 \) of \( Little \) shown here:

\[
\begin{align*}
\mathbf{x}_1 & : s_1 \quad Pub(x_1) \quad s_2 \quad Pub(x_1) \quad s_3 \quad Sec(x_1) \quad s_4 \quad \ldots \\
\mathbf{x}_2 & : t_1 \quad Pub(x_2) \quad t_2 \quad Sec(x_2) \quad t_3 \quad Pub(x_2) \quad t_4 \quad \ldots 
\end{align*}
\]

These behaviors are allowed by both the TLA and RTLA versions of \( Little \). In the second step, the value of \( out \) represented by \( x_1 \) and the value of \( in \) represented by \( x_2 \) both change, so they both change for their values represented by \( x_3 \). But, like \( Tiny \), \( Little \) allows no behavior in which a step changes both \( in \) and \( out \), so the required values for \( x_3 \), 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.
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 \text{out} of \( b_1 \) and the \( Sec \) events (changes to \text{in} of \( b_2 \). For example, suppose \( b_1 \) has an infinite number of \( Sec \) events. (The fairness condition \( L \) of \( P \) implies only that it must have infinitely many \( Pub \) events.) Let \( b_3 \) have a sequence of \( Pub \) steps that change \text{out} the same as the \( Pub \) steps of \( b_1 \) do, but set \text{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 \text{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 film 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 \( b_1 \) and \( b_2 \) 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 \( P(x_1) \) in (12). A behavior \( b \) satisfies this formula iff the values that \( b \) assigns to variables \( x_1 \) constitute a behavior in which the system described by \( P \) is satisfied when its variables are renamed to \( x_1 \). Moreover, values \( b \) assigns to other variables \( x_2 \) constitute a behavior of the same execution as the values \( b \) assigns to variables \( x_1 \) iff \( b(x_1) = b(x_1) \), which is equivalent to the condition that \( b \) provides \( x_1 \) as \( x_1 \). To obtain \( b_3 \) from a behavior \( b \), describing the same execution as \( b_1 \), we must replace \( public(x_3) = public(x_1) \) with \( public(x_3) = public(x_1) \) for some \( x_1 \) satisfying \( x_1 = x_1 \). Applying the same reasoning to \( x_3 \), we get the following TLA definition of \( P \) satisfying GNI, where \( x_1, x_2, x_3, x_1, \) and \( x_2 \) are all different variables.

\[
\models P(x_1) \land P(x_2) \Rightarrow \\
\exists x_1, x_2, x_3 : \\
(x_1 \sim x_1) \land (x_2 \sim x_2) \land P(x_3) \land L(x_1, x_2, x_3)
\]

where \( L(x_1, x_2, x_3) = \Delta \square (public(x_3) = public(x_1) \land secret(x_3) = secret(x_2)) \)

This formula is an assertion about behaviors \( b \) in which the values of the list of variables \( x_1 \) describe behavior \( b \), (for \( i = 1,2,3 \), and the values of the list of variables \( x_1 \) describe behavior \( b \).

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 \( b_1 \) and \( b_2 \) (of the same executions as \( b_1 \) and \( b_2 \) in which every \( Pub \) step of \( b_1 \) occurs in the same frames as a \( Pub \) step of \( b_2 \). We can achieve this by adding extra frames—steps that leave the values of \text{in}, \text{out}, and \text{nin} unchanged. For example:

\[
\begin{align*}
&x_1 : s_1 \quad Pub(x_1) \quad s_2 \quad Pub(x_1) \quad s_3 \quad Sec(x_1) \quad s_4 \quad \ldots \\
&x_2 : t_1 \quad Pub(x_2) \quad t_2 \quad Sec(x_2) \quad t_3 \quad Pub(x_2) \quad t_4 \quad \ldots \\
&x_3 : s_1 \quad Pub(x_3) \quad s_2 \quad Pub(x_3) \quad s_3 \quad Sec(x_3) \quad s_4 \ldots \\
&x_4 : s_1 \quad Pub(x_4) \quad t_2 \quad Sec(x_4) \quad t_3 \quad Pub(x_4) \quad t_4 \quad \ldots
\end{align*}
\]

To make \( Pub \) steps happen in corresponding frames of \( b_1 \) and \( b_2 \) 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, \( \infty \)—because the supplementary property \( L \) 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 \( x \) and \( y \), if there are the same number (possibly \( \infty \)) of \( A(x) \) and \( B(y) \) steps in \( b \), then there exist values for \( x \sim x \) and \( y \sim y \) such that \( x \sim x \) and \( y \sim y \), and a step is an \( A(x) \) step iff it is a \( B(y) \) step.

To state the rule precisely, we need a temporal operator \( \# \), where \( b \models A \# B \) is true for actions \( A \) and \( B \) iff there are the same number of \( A \) and \( B \) steps in \( b \). However, \( A \# B \) 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 \( \square \) only to actions of the form \( C \) to ensure that TLA formulas are SI, we apply \( \# \) only to actions of the form \( C \) to an action defined to equal \( C \lor v \neq v \). A \( \langle C \rangle \) 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 \( \langle C \rangle \).

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

\[
\models (\langle A(x) \rangle(x) \# \langle B(y) \rangle(y)) \Rightarrow \\
\exists x,y : (x \sim x) \land (y \sim y) \land \square (\langle A(x) \rangle(x) \# \langle B(y) \rangle(y))
\]

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 \( x_1 \) and \( x_3 \) comprises two variables representing \text{in} and \text{out}. Let \( P_i \) equal \( P(x_i) \), and for the other
defined quantities like $Q$ that also depend on $\text{nin}$, let $Q_i$ equal $Q(x_i, \text{nin}_i)$. Expanding the definitions of $P_1$ and $P_2$, (23) becomes
\[ \vdash (\exists \text{nin}_1 : Q_1) \land (\exists \text{nin}_2 : Q_2) \Rightarrow \exists \hat{x}_1, \hat{x}_2, x_3 : (\hat{x}_1 \sim x_1) \land (\hat{x}_2 \sim x_2) \land P_3 \land L(\hat{x}_1, \hat{x}_2, x_3) \]
where $L(\hat{x}_1, \hat{x}_2, x_3) \triangleq \square (\text{public}_3 = \text{public}(\hat{x}_1) \land \text{secret}_3 = \text{secret}(\hat{x}_2))$

By predicate logic reasoning, (25) is equivalent to
\[ \vdash Q_1 \land Q_2 \Rightarrow \exists \hat{x}_1, \hat{x}_2, x_3 : (\hat{x}_1 \sim x_1) \land (\hat{x}_2 \sim x_2) \land P_3 \land L(\hat{x}_1, \hat{x}_2, x_3) \]
Instead of verifying (26), we will verify a condition that implies (26). The definition of $\sim$ implies that $\hat{x}_i \sim x_i$ follows from $\hat{x}_i, \hat{y} \sim x_i, y$ for any variables $y$ and $\hat{y}$. Therefore, (26) is implied by:
\[ \vdash Q_1 \land Q_2 \Rightarrow \exists \hat{x}_1, \text{nin}_1, \hat{x}_2, \text{nin}_2, x_3 : (\hat{x}_1, \text{nin}_1) \sim x_1, \text{nin}_1 \land (\hat{x}_2, \text{nin}_2) \sim x_2, \text{nin}_2 \land P_3 \land L(\hat{x}_1, \hat{x}_2, x_3) \]
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, $Q_1 \land Q_2$ implies $\langle \text{Pub}(x_1, \text{nin}_1) \rangle_{(x_1, \text{nin}_1)} \neq \langle \text{Pub}(x_2, \text{nin}_2) \rangle_{(x_2, \text{nin}_2)}$. We can thus apply (24), substituting Pub for both $A$ and $B$. A Pub step changes out, which implies $\langle \text{Pub}(x_1, \text{nin}_1) \rangle_{(x_1, \text{nin}_1)}$ equals Pub$(x_1, \text{nin}_1)$. Therefore, instantiating (24) yields:
\[ \vdash Q_1 \land Q_2 \Rightarrow \exists \hat{x}_1, \text{nin}_1, \hat{x}_2, \text{nin}_2 : (\hat{x}_1, \text{nin}_1) \sim x_1, \text{nin}_1 \land (\hat{x}_2, \text{nin}_2) \sim x_2, \text{nin}_2 \land \square [\text{Pub}(\hat{x}_1, \text{nin}_1) \equiv \text{Pub}(\hat{x}_2, \text{nin}_2)]_{(\hat{x}_1, \text{nin}_1, \hat{x}_2, \text{nin}_2)} \]
By predicate logic reasoning, (28) implies that to verify (27), it suffices to verify:
\[ \vdash Q_1 \land Q_2 \land (\exists x_3 : P_3 \land L(x_3, x_3)) \]
Because $Q$ is SI, which is expressed in rule (21), $\hat{x}_1, \text{nin}_1 \sim x_1, \text{nin}_1$ implies that $Q_i$ is equivalent to $Q(x_i, \text{nin}_i)$. So, we can verify (29) by verifying
\[ \vdash Q(\hat{x}_1, \text{nin}_1) \land Q(\hat{x}_2, \text{nin}_2) \land \square [\text{Pub}(\hat{x}_1, \text{nin}_1) \equiv \text{Pub}(\hat{x}_2, \text{nin}_2)]_{(\hat{x}_1, \text{nin}_1, \hat{x}_2, \text{nin}_2)} \Rightarrow (\exists x_3 : P_3 \land L(x_3, x_3)) \]
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 $x_1$ and $x_2$ with equivalent films $\hat{x}_1$ and $\hat{x}_2$ and used rule (24) to add the hypothesis $\square [\text{Pub}(\hat{x}_1) \equiv \text{Pub}(\hat{x}_2)]_{(\hat{x}_1, \hat{x}_2)}$ that synchronizes the two films.

By substituting $x_1$ and $\text{nin}_1$ for $\hat{x}_1$ and $\text{nin}_1$, expanding the definition of $P_3$, and predicate logic, (30) becomes
\[ \vdash Q_1 \land Q_2 \land \square [\text{Pub}_1 \equiv \text{Pub}_2](x_1, \text{nin}_1, x_2, \text{nin}_2) \Rightarrow (\exists \text{x}_3, \text{nin}_3 : P_3 \land L(x_1, x_2, x_3)) \]
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 $x$ the existence of values for variables $\hat{x}$ with $\hat{x} \sim x$ that satisfy some condition.

We verify that Little satisfies (31) in the same way we verified that Tiny satisfies (14): We rewrite
\[ Q_1 \land Q_2 \land \square [\text{Pub}_1 \equiv \text{Pub}_2](x_1, \text{nin}_1, x_2, \text{nin}_2) \]
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 $Q(x_i, \text{nin}_i)$ has the term $\square [\text{A}[x_i, \text{nin}_i]](x_i, \text{nin}_i)$ instead of $\square \text{A}[x_i, \text{nin}_i]$. Let $u$ equal $(\text{in}, \text{out}, \text{nin})$. Expanding the definition of $P$ and rearranging the terms, (32) becomes
\[ (\mathcal{I}_{1} \land \mathcal{I}_{2}) \lor (\square [\mathcal{N}_{1}]_{(v_{1}, u_{v}}) \land \square [\mathcal{N}_{2}]_{(v_{2}, u_{v}}) \lor \square [\text{Pub}_1 \equiv \text{Pub}_2]_{(v_{1}, v_{2})}) \]
\[ \land (\mathcal{L}_{1} \land \mathcal{L}_{2}) \]
To transform this to a standard TLA state machine description, we write the shaded expression as $\square [\mathcal{M}]_{(v_{1}, v_{2})}$ for a next-state action $\mathcal{M}$. Using the rule that $\square$ distributes over $\land$ and remembering that $[A]_u$ equals $A \lor (u' = u)$, we see that we can let $\mathcal{M}$ equal
\[ ((\mathcal{N}_{1} \land \mathcal{N}_{2}) \lor (\mathcal{N}_{1} \land (v_{2}' = v_{2})) \lor (\mathcal{N}_{2} \land (v_{1}' = v_{1})) \]
Expanding the definition of $\mathcal{N}$ and using the facts that $\text{Pub}$ implies $\langle \sim \text{Sec} \rangle \land (v' \neq v)$ and $\text{Pub}_1 \equiv \text{Pub}_2$, we can rewrite this formula as
\[ (\text{Pub}_1 \land \text{Pub}_2) \lor (\text{Sec}_1 \land \text{Sec}_2) \lor (\text{Sec}_1 \land (v_{2}' = v_{2})) \lor (\text{Sec}_2 \land (v_{1}' = v_{1})) \]
which is the next-state action of a state machine with variables $x_1, \text{nin}_1, x_2$, and $\text{nin}_2$. 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 $\text{Val}$ and bound the value of $\text{nin}$ by substituting a small set $\{0, \ldots, n\}$ for $\text{Nat}$. 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 satisfies the TLA version of GNI. For the TLA verification, we verify its RTLA proof in Section IV-C with Little proof works for any RTLA proof that a system satisfies TLA. This is the formula one would obtain from TLAPS. The complete TLA formalizations are 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 TLA 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 \( N \) instead of \( Pub \). With the subscripting notation and definition of \( v \) as \( \langle \text{in}, \text{out}, \text{nin} \rangle \) from Section VI-E, formula (32) then becomes:

\[
Q_1 \land Q_2 \land \square[N_1 \equiv N_2]_{\langle v_1, v_2 \rangle}
\]

(33)

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

\[
(Q_1 \land Q_2) \land \square[N_1 \land N_2]_{\langle v_1, v_2 \rangle} \land (WF_{v_1}(N_1) \land WF_{v_2}(N_2))
\]

(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 \( Q(x_1) \land Q(x_2) \) 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

Zdancewicz 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 if every behavior took the same number (possibly \( \infty \)) of public steps. That a system \( P \) satisfies OD would then be expressed by:

\[
\begin{aligned}
\models P(x_1) \land P(x_2) \land (public(x_1) = public(x_2)) \\
\Rightarrow \exists \bar{x}_1, \bar{x}_2 : x_1 \sim x_1 \land x_2 \sim \bar{x}_2 \land \\
\square(public(x_1) = public(x_2))
\end{aligned}
\]

(35)

Zdancewicz 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

\[
\text{term}(x_1) \lor \text{term}(x_2) \lor (public(x_1) = public(x_2))
\]

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

---

\[ ^4 \text{Recall that public-stuttering equivalent is defined in Section V-A.} \]
way, verifying that PharOS satisfies (35) can, by applying (24), be reduced to verifying

\[ \models P(x_1) \land P(x_2) \land (\text{public}(x_1) = \text{public}(x_2)) \land (\Box \text{Public}(x_1) \equiv \text{Public}(x_2)_{[x_1, x_2]}) \rightarrow \Box (\text{public}(x_1) = \text{public}(x_2)) \] (36)

where \( P \) is the TLA\(^+\) model of PharOS and \( \text{Pub} \) describes the steps taken by the given agent. Just as in the verification that \( \text{Little} \) 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. Noninterference

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 \( \text{public} \) 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 \( \text{cmd} \) equal to the name of the most recent command.

NI asserts that executing any sequence of commands produces the same values of \( \text{public} \) 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 \( PC \):

1. Every pair of system behaviors \( b_1 \) and \( b_2 \) that satisfy \( K \) produce the same values of \( \text{public} \).
2. For every system behavior \( b_1 \) there exists a behavior \( b_2 \) satisfying \( K \) that produces the same values of \( \text{public} \) as \( b_1 \).

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

\[ \models P(x_1) \land P(x_2) \land K \rightarrow \Box (\text{public}(x_1) = \text{public}(x_2)) \] (37)

\[ \models P(x_1) \rightarrow \exists x_2 : P(x_2) \land K \land \Box (\text{public}(x_1) = \text{public}(x_2)) \] (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 \( x_1 \) and \( x_2 \) are properly aligned. It is not needed in (38) because \( K \) implies that \( x_1 \) and \( x_2 \) can be aligned by adding frames to \( x_2 \), which is allowed by the \( \exists \) operator. The \( \sim \) operator was needed in GNI (23) and OD (35) to allow replacing the “films” \( x_1 \) and \( x_2 \) with films \( \hat{x}_1 \) and \( \hat{x}_2 \) of the same executions, but properly aligned.

Noninterference (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 \( \text{secret} \) whose values are changed by executing commands not in \( PC \). All commands in a behavior being commands in \( PC \) is then equivalent to \( \text{secret} \) having the same value throughout the behavior. Mantel’s version of NF is then described by (38) when \( K \) is the assertion that \( \text{secret}(x_2) \) never changes, expressed in TLA as:

\[ \Box (\text{secret}(x_2) = \text{secret}(x_2)) \] (38)

This version of (38) is satisfied by \( \text{Little} \), but not by \( \text{Tiny} \).

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

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 \( \text{public} \). PN is satisfied by a system iff, for every possible system behaviors \( b_1 \) and \( b_2 \) such that \( \text{public} \) 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 \( \text{public} \) 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 \( \text{public} \) is the value of the hour and minute, in this model observing only \( \text{public} \) reveals that the clock is also counting seconds because that same value of \( \text{public} \) 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 \( \text{public} \) 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 b1, b2, and b3 that represent films by adding the requirement that b1 and b2 are aligned so that their states change at the same time. (There is no need to align b2 with b1 and b3 because only the initial state of b2 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:

\[
\models P(x_1) \land P(x_2) \land (\text{public}(x_1) = \text{public}(x_2)) \quad (39)
\]

\[
\exists x_3 : P(x_3) \land K \land ((x_2) = (x_3)) \land \Box(\text{public}(x_1) = \text{public}(x_3))
\]

where the alignment condition \( K \) is defined by

\[
K \equiv \Box(((x_1)' \neq (x_1)) \equiv ((x_3)' \neq (x_3)))_{(x_1, x_3)}
\]

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:

\[
\models Q(x_1, n_{in}) \land Q(x_2, n_{in}) \land (\text{public}(x_1) = \text{public}(x_2)) \quad (40)
\]

\[
\exists x_3 : P(x_3) \land K \land ((x_2) = (x_3)) \land \Box(\text{public}(x_1) = \text{public}(x_3))
\]

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

\[
\models Q^h(x_1, n_{in}, h) \land Q(x_2, n_{in}) \land (\text{public}(x_1) = \text{public}(x_2)) \land (\text{public}(x_1) = \text{public}(x_3)) \land \Box(\text{public}(x_1) = \text{public}(x_3))
\]

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 \( x_3 \) equal the values of \( x_2 \) if \( h = 1 \) and the values of \( x_1 \) 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 \( n_{in} \)—steps that represent input or output of the same value twice in a row—and \( n_{in} \) 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:

\[
\models P(x_1) \land P(x_2) \land (\text{inp}(x_1) < \text{inp}(x_2)) \Rightarrow \Box(\text{term}(x_1) \land \text{term}(x_2) \Rightarrow (\text{outp}(x_1) < \text{outp}(x_2)))
\]

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 \( P \) if \( S \) 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 \( \models P(x) \Rightarrow L(x) \). 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 makes 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

\[
\models S(w) \land I(w, x) \Rightarrow P(x) \tag{41}
\]

where \( I \) must satisfy:

\[
\models S(w) \Rightarrow \exists x : I(w, x) \tag{42}
\]

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

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 \( I(w, x) \) is \( \Box(x = g(w)) \), in which case (41) is equivalent to

\[
\models S(w) \Rightarrow P(g(w)) \tag{43}
\]

and we say \( S \) refines \( P \) under interface refinement mapping \( g \).

Mathematically, (43) is the same condition that arises if the variables of \( P \) are regarded as hidden and we are given the refinement mapping \( g \) under which \( S(w) \) must imply \( \exists x : P(x) \). This form of \( I \) handles the example of refining a screen that displays characters with one displaying pixels, where \( g(w) \) specifies the screen of characters that corresponds to the screen of pixels described by \( w \). 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 \( g \), there are natural candidates for \( K_S \) and \( L_S \):

\[
K_S(w_1, \ldots, w_j) \triangleq K(g(w_1), \ldots, g(w_j))
\]

\[
L_S(w_1, \ldots, w_k) \triangleq L(g(w_1), \ldots, g(w_k))
\]

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 \textit{secret} and \textit{public} together specify the values of all the free variables of \( P \). However, the values of \textit{secret}(x) and \textit{public}(x) can specify the values of the free variables \( x \) of \( P(x) \) without \textit{secret}(\( g(w) \)) and \textit{public}(\( g(w) \)), which appear in \( K_S \) and \( L_S \), specifying the values of the free variables \( w \) of \( S(w) \).

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

\[
\models I(w, x) \Rightarrow (S(w) \equiv P(x))
\]

\[
\models P(x) \Rightarrow \exists w : I(w, x)
\]

\[
\models S(w_1), \ldots, S(w_j) \land K_S(w_1, \ldots, w_j) \land I(w_1, x_1) \land \ldots \land I(w_j, x_j) \Rightarrow K(x_1, \ldots, x_j)
\]

\[
\models P(x_1) \land \ldots \land P(x_k) \land K(x_1, \ldots, x_j) \land L(x_1, \ldots, x_k) \land I(w_1, x_1) \land \ldots \land I(w_j, x_j) \Rightarrow L_S(w_1, \ldots, w_k)
\]

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 ∀∃-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 \( \exists \) (i.e., when \( k > j \)). Coenen et al. [11] enhanced MCHyper to handle ∀∃-hyperproperties more efficiently, based on a game-theoretic metaphor. In effect, they partially automated construction of the
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 chips [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


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


