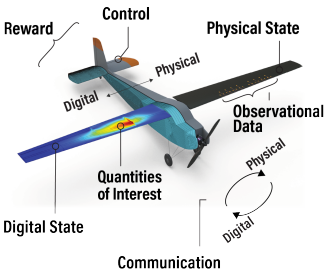
Graphical Abstract

Formal Verification of Digital Twins with the Temporal Logic of Actions

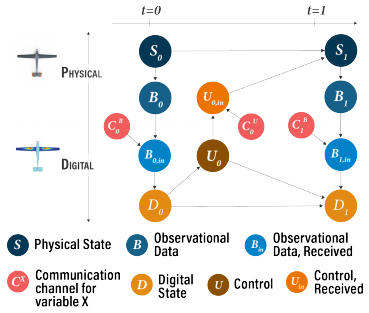
Luwen Huang, Ufuk Topcu, Lav R. Varshney, Karen Willcox

A Formal Approach to Digital Twin Verification

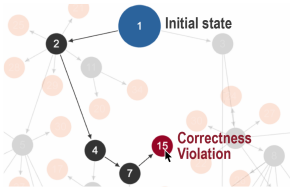
The Seven Fundamental Elements of a Digital Twin



The Digital Twin Probabilistic Graphical Model



The Digital Twin Finite State Machine



Deriving the digital twin finite state machine from its probabilistic graphical model to uncover property violations over all possible behaviors throughout the digital twin's lifespan.

Highlights

Formal Verification of Digital Twins with the Temporal Logic of Actions

Luwen Huang, Ufuk Topcu, Lav R. Varshney, Karen Willcox

- A methodology for formal verification of digital twins that transforms a probabilistic graphical model of the digital twin into a finite state machine, equipped with a formal representation of concurrency and an explicit model of communication uncertainty.
- A methodology to identify digital twin verification goals from the digital twin finite state machine.
- The Temporal Logic of Actions models asynchronous evolution and uncertain communication in digital twins.
- Formal specification clarifies key decisions in structuring an unmanned aerial vehicle digital twin, including which assumptions are essential and how different components are represented in the specification.

Formal Verification of Digital Twins with the Temporal Logic of Actions

Luwen Huang[a], Ufuk Topcu[b,d], Lav R. Varshney[c], Karen Willcox[b,d,e,*]

[a]*Department of Computer Science, University of Texas at Austin,*
[b]*Department of Aerospace Engineering and Engineering Mechanics, University of Texas at Austin,*
[c]*AI Innovation Institute, Stony Brook University,*
[d]*Oden Institute for Computational Engineering and Sciences, University of Texas at Austin,*
[e]*Santa Fe Institute,*

Abstract

Verification of digital twins, especially in the presence of distributed components and asynchronous communication, remains a fundamental challenge. Uncertainty arises not only from the physical environment and virtual representations, but also from the bidirectional flow of data between them. This paper presents a formal verification methodology that captures these dynamics by translating a digital twin's probabilistic elements into a finite state machine, augmented with explicit models of message delay and concurrent evolution. Using the Temporal Logic of Actions (TLA), we specify and verify temporal properties that ensure system-level correctness despite communication disorder and component-level asynchrony. We demonstrate this approach on a digital twin of an unmanned aerial vehicle, showing how to verify synchronization properties that govern end-to-end coherence between the physical and virtual components. Our case study highlights how formal specification clarifies system modeling choices, including which guarantees must hold, how system behaviors should be represented, and what assumptions are essential.

Keywords: Digital twins, TLA, formal verification, model checking

*Corresponding Author

Email addresses: luwen@cs.utexas.edu (Luwen Huang), utopcu@utexas.edu (Ufuk Topcu), lav.varshney@stonybrook.edu (Lav R. Varshney), kwillcox@oden.utexas.edu (Karen Willcox)

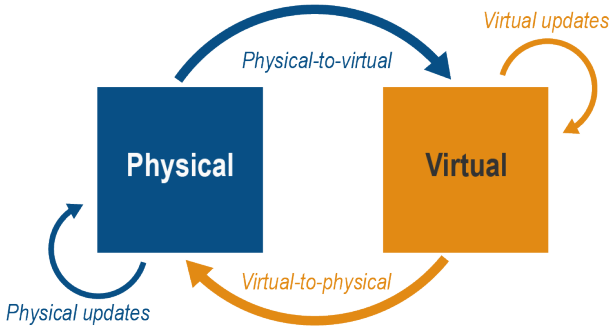


Figure 1: In a digital twin system, the physical and virtual realms each evolve dynamically and also evolve as a result of bidirectional data flow.

1. Introduction

Verification, together with validation and uncertainty quantification, plays a critical role in establishing trust in digital twins [1]. We adopt the National Academies' definition of a digital twin as "a set of virtual information constructs that mimics the structure, context, and behavior of a natural, engineered, or social system (or system-of-systems), is dynamically updated with data from its physical twin, has a predictive capability, and informs decisions that realize value" [1]. As illustrated in Fig. 1, a digital twin system consists of a physical system that evolves in the real world and a virtual representation that operates in a virtual space. Bidirectional interaction between the two is not just a feature but a defining element of a digital twin. As emphasized by the National Academies, "the bidirectional interaction between the virtual and the physical is central to the digital twin" [1]. This bidirectional interaction is also a central consideration for digital twin verification. Not only do both the physical system and the virtual representation each evolve dynamically, but they also interact through asynchronous, bidirectional communication. Observational data flows from the physical to the virtual, while digital twin outputs, such as predictions, assessments, or decisions, flow back to the physical. This paper presents a formal verification methodology that specifies a digital twin system, including an explicit model of the properties of the bidirectional communication, and verifies its correctness using the Temporal Logic of Actions (TLA).

Formal verification is a well-established area of research [2, 3, 4], with

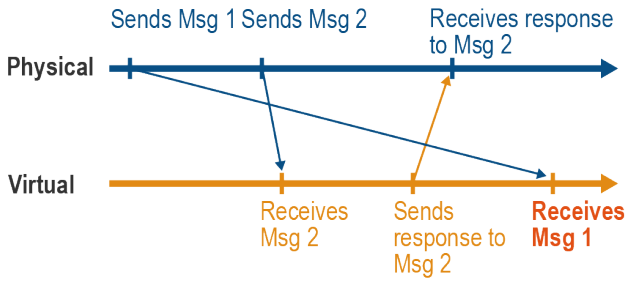


Figure 2: A simple example of the kinds of questions digital twin verification needs to address: Message 1 arrives after Message 2. How should this be handled?

active work on safety, autonomy, and control [5, 6, 7, 8, 9]. For example, [10] extends the Lingua Franca coordination language to handle network failures, while others propose methods to achieve deterministic timing in distributed control executions [11, 12, 13, 14]. When it comes to digital twins, the complexity of the virtual representation and its dynamic evolution makes the verification task challenging. Furthermore, ensuring that each component of the digital twin system behaves correctly in isolation does not guarantee correct orchestration across the full system. Our approach shifts focus toward system-level digital twin verification in the context of asynchronous evolution and bidirectional communication. Our work recognizes the challenge that in many real-world digital twin implementations, the interactions between physical and virtual will be distributed, asynchronous, and error-prone. Fig. 2 illustrates one simple scenario that underlines the importance of formal digital twin verification: the physical system sends two messages in order, but they are received out of order by the digital twin. How should the digital twin handle the stale message when it finally arrives? How should the physical twin interpret a delayed response?

In the field of distributed computing, TLA has been widely used for specifying and verifying correctness in systems with concurrent processes [15, 16, 17, 18], including high-profile industrial applications such as distributed resource management at Amazon Web Services [19]. Digital twin systems pose distinct challenges that extend beyond conventional distributed software systems. First, digital twins tightly couple heterogeneous physical and computational elements, requiring reasoning over sensors, actuators, communication channels, and predictive models. Second, digital twins often incorporate probabilistic models whose outputs evolve with real-time data,

introducing further complexity. Third, digital twins must maintain ongoing synchronization between physical and virtual processes through bidirectional communication. These requirements differentiate digital twins from traditional distributed systems and demand specialized modeling techniques. We address this need by developing a TLA-based approach that formally specifies and verifies temporal properties under asynchronous evolution and uncertain communication. Our approach provides a foundation for system-level verification in the emerging digital twin domain, complementing lower-level control and communication guarantees by explicitly modeling how such components interact in the full system context.

This paper focuses on the challenge of how to formally specify and verify the correctness of digital twin systems in the presence of asynchronous, distributed behavior and uncertain communication. We present a methodology that transforms a probabilistic graphical model (PGM) of the digital twin into a finite state machine (FSM), equipped with a formal representation of concurrency and an explicit model of communication uncertainty. Using TLA, we express and verify temporal properties that capture both the asynchronous evolution of the physical and virtual components and the effects of message channel uncertainty. Our results show that correctness properties can be guaranteed, even when components evolve independently and communicate via an imperfect message channel.

We introduce the following novel contributions:

1. **Formal digital twin system specification**: A method for constructing formally verifiable specifications of digital twins in TLA, derived from a probabilistic graphical model and expressed as a finite state machine that formally captures concurrency and asynchronous behavior.
2. **Model of digital twin bidirectional communication**: An explicit model of digital twin bidirectional communication achieved via an augmentation of the PGM framework to explicitly model channel uncertainty in distributed communication flows.

The remainder of this paper is organized as follows. Section 2 introduces our digital twin formal verification framework, with three parts: we first introduce our abstraction of the digital twin as a PGM, second describe our approach of mathematically deriving the FSM abstraction from the PGM representation, and third describe how to formally model communication in the digital twin FSM. Next, Section 3 identifies the verification goals of the

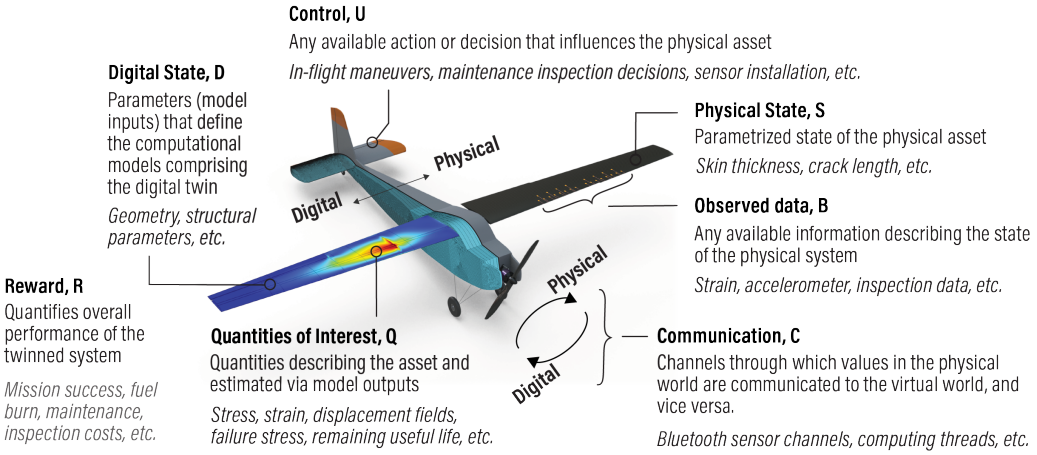


Figure 3: Example instantiation of the seven digital twin elements for an unmanned aerial vehicle (UAV). These elements apply across varied digital twin application domains such as aerospace, medicine, manufacturing, etc. Figure modified from [20].

digital twin system. Section 4 demonstrates the approach through an illustrative application to a structural health monitoring digital twin of an unmanned aerial vehicle, including specification, abstraction, and verification. Section 5 concludes the paper.

2. A Mathematical Formulation for Verifiable Digital Twins

This section introduces the integral elements of a digital twin along with its probabilistic graphical model (PGM) formulation. We then present the digital twin finite state machine (FSM), deriving its structure with abstractions for the initial state, state transitions, termination conditions, and fairness assumptions.

2.1. The Digital Twin Probabilistic Graphical Model

Prior work by [20] identified six key elements that characterize a digital twin, forming a generalized digital twin mathematical formulation that has been used across varied application domains such as aerospace [21], civil infrastructure [22], and healthcare [23]. These six elements, which may be scalar or vector-valued depending on the application, are as follows:

1. The **physical state $\boldsymbol{S}$** consists of parameters that describe the physical system.

2. The **observational data** $\boldsymbol{B}$ consist of any available information describing the state of the physical system.
3. The **control inputs** $\boldsymbol{U}$ represent any action or decision that influences the physical system.
4. The **digital state** $\boldsymbol{D}$ consists of the parameters that define the computational models underlying the digital twin.
5. The **quantities of interest** $\boldsymbol{Q}$ are the quantities describing the physical system, estimated via model outputs.
6. The **reward** $\boldsymbol{R}$ quantifies performance of the twinned system with respect to mission goals or operational objectives, such as safety, cost, efficiency, or success criteria.

In this work, we introduce a seventh core element: **communication channel** $\boldsymbol{C}$. While bidirectional communication between the physical and digital systems is integral to a digital twin [1], it is seldom modeled as an explicit component of the digital twin framework. However, in real-world deployments, such communication occurs over imperfect channels that may introduce latency, message reordering, or data loss. For instance, sensor data may be transmitted from the physical system to a remotely hosted digital twin, while control inputs must traverse the reverse path to physical actuators or human operators; both directions are subject to delay and error. Communication induces system-wide concurrency: sensing, transmission, computation, and actuation in different components proceed independently and interleave nondeterministically. For instance, a control command may be mid-computation when fresh sensor data arrives, leading to race conditions or inconsistent state updates. Such behaviors can arise even when each component is correct in isolation, because correctness does not necessarily compose when components interact over imperfect, concurrent channels. We refer to this phenomenon as *distributed communication*. In such settings, component-level correctness does not guarantee system-level correctness. Accordingly, in our approach, we treat distributed communication as a first-class modeling element, on equal footing with the six previously established components.

The digital twin PGM framework, first introduced in [20], provides a formal representation of how the elements in a digital twin evolve over time. In a PGM, each node corresponds to a random variable, and each edge indicates a conditional dependence between variables. Figure 4 illustrates a PGM with two digital twin elements across two timesteps: the digital state at the initial timestep, D_0, the control decision computed at that timestep, U_0, and the

Algorithm 1 Augmentation of the digital twin PGM to model distributed communication

Require: $\mathcal{G} = (\mathcal{V}, E)$: PGM with vertex set $\mathcal{V}$ and edge set E
Ensure: Augmented PGM $\mathcal{G}' = (\mathcal{V}', E')$ that explicitly models communication

Initialize $\mathcal{V}' \leftarrow \mathcal{V}$, $E' \leftarrow E$
for all $(X, Y) \in E$ **such that** `DistributedChannel`$(X \to Y) =$ `True` **do**
    Node Augmentation:
        Add node X_{in} to $\mathcal{V}'$
        Add node C^X to $\mathcal{V}'$
    Edge Restructuring:
        Remove edge $X \to Y$ from E
        Add edges: $\{X \to X_{\text{in}}, C^X \to X_{\text{in}}, X_{\text{in}} \to Y\}$ to E'
    if channel state C^X depends on transmitted value X **then**
        Add edge $X \to C^X$ to E'
    end if
end for
return $\mathcal{G}' = (\mathcal{V}', E')$

digital state at the next timestep, D_1. The subscripts indicate the temporal

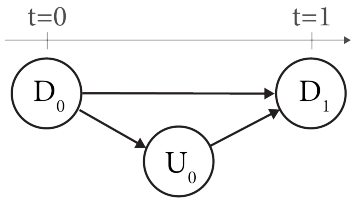


Figure 4: A probabilistic graphical model (PGM) with random variables D_0, U_0, and D_1 across two timesteps. Edges indicate conditional dependencies, e.g., $P(U_0 \mid D_0)$.

index of each variable. The directed edge from D_0 to U_0, denoted $D_0 \to U_0$, captures the dependency of control decisions on the digital state. The edges $D_0 \to D_1$ and $U_0 \to D_1$ encode how the digital state at the next timestep evolves as a function of both the prior state and the previously computed control.

We introduce an augmentation to the digital twin PGM framework that incorporates communication as a formally modeled process. Specifically, for every variable X that is communicated via a distributed channel, we begin by identifying three random variables: (1) the transmitted value X (already existing), (2) a new random variable X_{in} representing the received value of

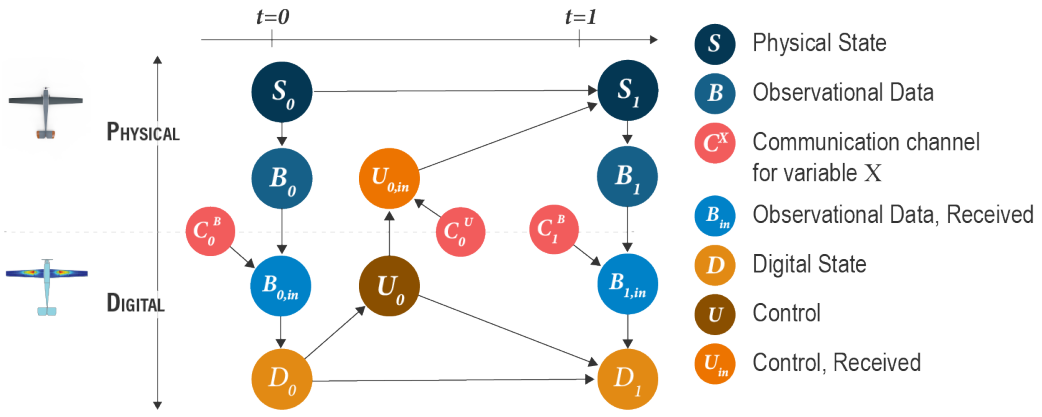


Figure 5: Illustrative digital twin probabilistic graphical model (PGM) spanning two timesteps encoding dependencies among core elements of this digital twin: physical state (S), observational data (B), digital state (D), control (U), and communication (C^X for each variable X communicated over a distributed channel). Directed edges indicate probabilistic dependencies, e.g., $S \to B$.

X, and (3) a new random variable C^X representing the reliability of the communication channel (e.g., latency, loss, reordering). We then introduce two new edges: $X \to X_{\text{in}}$, indicating that the received value depends on the transmitted one, and $C^X \to X_{\text{in}}$, indicating that channel conditions influence the received value. To maintain causal integrity, we replace all downstream edges originating from X with edges from X_{in} — reflecting the fact that only received data influences subsequent variables, consistent with prior analyses of message errors in graphical models [24]. Optionally, we may include an edge $X \to C^X$ if the transmitted value influences the state of the channel. This dependency arises in scenarios where large or frequent transmissions alter channel behavior, such as buffer overflow or congestion in bandwidth-constrained systems. Algorithm 1 details the augmentation procedure. We write $\mathcal{G} = (\mathcal{V}, E)$ for the PGM prior to augmentation, where $\mathcal{V}$ is the vertex set and E is the edge set, and initialize the augmented graph as a copy $\mathcal{G}' = (\mathcal{V}', E')$ with $\mathcal{V}' \leftarrow \mathcal{V}$ and $E' \leftarrow E$, where $\leftarrow$ denotes assignment. The predicate $\texttt{DistributedChannel}(X \to Y)$ is `True` if the dependency $X \to Y$ is carried over a distributed channel. Figure 5 illustrates the augmented structure. In this augmented PGM, variables include physical state S, emitted observational data B, received data B_{in}, emitted control U, received control U_{in}, digital state D, and communication channels C^B and C^U.

Throughout this work, we use the digital twin PGM shown in Figure 5 as

a running example to guide exposition. For detailed guidance on constructing the digital twin PGM, we refer readers to the original work [20].

2.2. The Digital Twin Finite State Machine

We next formulate an FSM representation of the digital twin, which provides the mathematical foundation for digital twin verification. An FSM is a formal model consisting of a set of *states*, a set of *transitions* between those states, and rules that determine which transitions are possible from each state. In the FSM context, a *state* is a complete assignment of values to all variables in the system at a given instant, and a *transition* encodes the logical conditions under which those variables may change.

We express the digital twin FSM in the *Temporal Logic of Actions* (TLA) [25], a temporal logic for specifying and reasoning about discrete systems. Unlike other linear-time logics that describe properties of states [2], TLA represents state changes as explicit *actions*, making it possible to specify not just what properties hold, but also how the system evolves from one state to the next. This action-oriented view is particularly valuable for specifying digital twins, where understanding and abstracting the behavior of individual transitions is as important as ensuring that desired properties are satisfied. In this paper, we follow Lamport's usage of the term "specification" [25, 26], referring to both the formal FSM model and its properties expressed in TLA, whereas other conventions, e.g., in [27, 3, 4], refer to "specification" only as the properties to be satisfied by an underlying system.

We define the digital twin FSM as:

$$\text{Digital Twin} \coloneqq \mathcal{I} \wedge \mathcal{N} \wedge \mathcal{F} \tag{1}$$

This representation comprises three components: the initial predicate $\mathcal{I}$, which specifies valid starting configurations; the next-state predicate $\mathcal{N}$, which encodes how the system can legally evolve; and the fairness assumptions $\mathcal{F}$, which constrain long-run behaviors by determining which transitions must eventually occur. A *predicate* is a logical condition: given a candidate state (or state transition), it evaluates to *true* if that state satisfies the intended rule, and *false* if it violates it. For example, a predicate requiring "the battery level must be non-negative" is true in all states with a non-negative battery value, and false otherwise.

The conjunction operator $\wedge$ requires all three predicates to hold simultaneously: a valid execution must start in a state satisfying $\mathcal{I}$, evolve according

to $\mathcal{N}$, and respect $\mathcal{F}$. Writing the specification in this form allows us to use *model checking* — an automated procedure that systematically explores every state and transition to determine whether all required predicates (including properties defined in Section 3) are always true [3]. This exhaustive search allows us to give formal guarantees about the system's behavior. The computational procedure for this analysis is described in Section 4.

On notation. Following formal methods conventions, we use lowercase symbols to denote variables. Hence, random variable X in the digital twin PGM is denoted in the digital twin FSM as x. We also adopt the standard convention that x denotes the value of a variable before a transition, and x' denotes its value after the transition occurs.

To construct the variables in the digital twin FSM, we must first abstract the real-world quantities into a form amenable to symbolic verification. This means we cannot, for instance, directly model real-valued sensor readings like 3.223411. Instead, for continuous quantities, we must construct discrete representations. The abstraction principles below guide this construction process and define how real-valued, continuous-time, and stochastic elements from the PGM are transformed into symbolic representations in the FSM.

Abstraction Guideline I: Discretization of continuous values. Discretizing continuous values into integers is a common formal methods technique that facilitates verification [28]. Let X be a continuous-valued variable in the PGM such that $X \in [a, b]$. In the FSM abstraction, we discretize X as $x \in a', a'+1, \ldots, b'$, where $a' = \lfloor a \rfloor$ and $b' = \lceil b \rceil$. While integer discretization is common, FSM variables may in general range over other finite symbolic domains, including booleans, strings, tuples, or bounded vectors, as appropriate for the verification objective.

Abstraction Guideline II: Discretization of time. Physical systems evolve over continuous time, but the FSM abstraction advances in discrete steps indexed by $n \in \mathbb{N}$. Each transition corresponds to a time step of size Δt, such that continuous time t is sampled at $t_n = n \cdot \Delta t$. The choice of Δt is determined by the verification objectives and sets the temporal resolution of the abstraction. Within this framework, a transition is treated as *atomic* — all changes specified by the transition occur instantaneously within a single discrete time step.

Abstraction Guideline III: Modeling communication channels as queues. In practice, communication uncertainty arises from latent factors such as latency, packet loss, and noise, which can be modeled as random parameters $\boldsymbol{\alpha}$ governing the channel distribution $C^X \sim P(\boldsymbol{\alpha})$. In the FSM abstraction, we adopt a formal methods convention by modeling communication channels as message queues [29]. Let c^X denote the queue abstraction representing channel C^X. Sending a message x appends it to the queue:

$$\texttt{Append}(c^X, x) \coloneqq\ c^{X\prime} = c^X \cup \{x\} \tag{2}$$

Let $k \in 0, \ldots, \texttt{len}(c^X) - 1$ be a nondeterministically chosen index and $c^X[k]$ be the $(k+1)$th element in queue c^X. Receiving a message removes $c^X[k]$ from the queue and assigns it to variable x_{in}:

$$\begin{aligned} \texttt{Receive}\ x_{\text{in}} \coloneqq\ & x'_{\text{in}} = c^X[k] \\ & \wedge\ \texttt{Remove}(c^X, k) \end{aligned} \tag{3}$$

$$\text{where } \texttt{Remove}(c^X, k) \coloneqq c^{X\prime} = c^X \setminus \{c^X[k]\}$$

Table 1 summarizes the key variables in the digital twin FSM abstraction, grouped by physical system, virtual representation, and communication channels. The left column describes each variable as it appears in the PGM, reflecting how quantities are modeled computationally through statistical dependencies and behaviors governed by physical laws. The right column presents how each variable is abstracted in the FSM. In implementation (see Section 4), these FSM abstraction variables carry auxiliary metadata — such as timestamps, message indices, or source labels — which are omitted here for clarity.

PGM Representation	**FSM Abstraction**
Physical System Variables	
Physical state $S \in \mathbb{R}$: physical state parameters evolving via dynamics and control.	**Abstraction** $s \in \mathbb{Z}$: discretized physical parameters.
Observational data $B^m \in \mathbb{R}$: output of sensor m. Index m ranges over $\{1, \ldots, M\}$ sensors.	**Abstraction** $b^m \in \mathbb{Z}$: discretized sensor output for sensor m.

Control received $U_{\text{in}} \in \mathcal{U}$: control message received by the physical system, subject to channel uncertainty. $\mathcal{U}$ is the set of all valid control messages (continuous, discrete, or mixed).	**Abstraction** $u_{\text{in}} \in \mathbb{Z}$: integer-encoded control category, obtained by mapping each $U \in \mathcal{U}$ to a finite set of codes.
Digital Twin Variables	
Observational data received $B_{\text{in}}^m \in \mathbb{R}$: sensor m data received by the virtual representation, subject to channel uncertainty.	**Abstraction** $b_{\text{in}}^m \in \mathbb{Z}$: discretized received sensor data for sensor m.
Digital state $D \in \mathbb{R}$: parameters representing the virtual representation's estimate of the physical state.	**Abstraction** $d \in \mathbb{Z}$: discretized digital state.
Control $U \in \mathcal{U}$: control message computed by the virtual representation.	**Abstraction** $u \in \mathbb{Z}$: integer-encoded control category.
Communication Channel Variables	
Observational data channel $C^{B,m}$: channel model for transmitting sensor m data.	**Abstraction** $c^{B,m} = [\,]$ as queue: append message on send, retrieve on receive; message at time t is $c^{B,m}[t]$.
Control channel C^U: channel model for transmitting control data.	**Abstraction** $c^U = [\,]$ as queue: append message on send, retrieve on receive; message at time t is $c^U[t]$.

Table 1: Key variables in the digital twin system. The left column describes each variable's behavior in the PGM. The right column shows how each variable is abstracted in the FSM using discrete representations.

2.3. Deriving Next State Transitions

The next state predicate defines the allowable ways the system may evolve from one state to the next. In a single timestep, the digital twin advances by executing exactly one transition ω — chosen from the entire set of FSM transitions Ω — or by transitioning to a termination condition $\mathcal{T}$:

$$\mathcal{N} \coloneqq (\omega \in \Omega) \vee \mathcal{T} \tag{4}$$

The logical disjunction ($\vee$) indicates that at each discrete step, only one transition ω or $\mathcal{T}$ is executed.

We introduce our methodology for deriving the set of transitions Ω from the structure of the corresponding digital twin PGM. The key insight is that

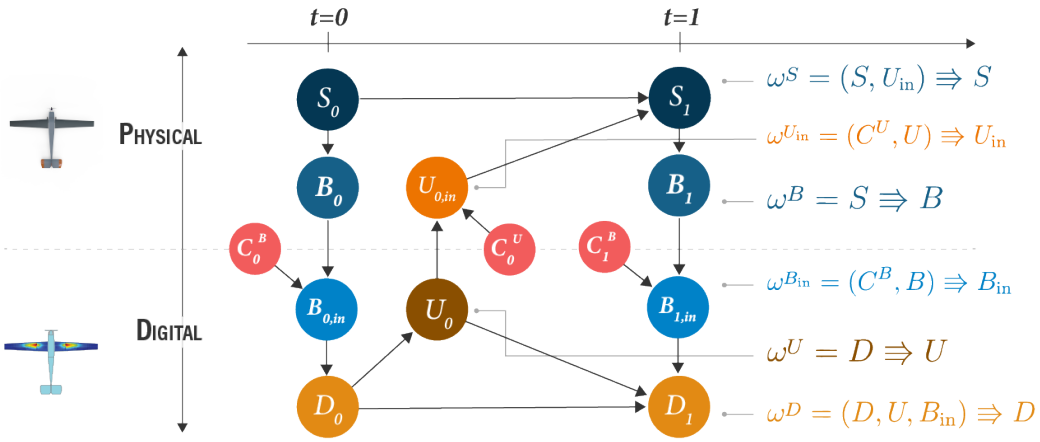


Figure 6: Each FSM transition ω^X corresponds to the update of a variable $X \in \mathcal{X}$, where $\mathcal{X}$ is the set of non-time-indexed variables in the PGM.

an FSM transition ω^X, which governs the update of a particular variable X, corresponds to the conditional dependencies encoded in the PGM. Specifically, since the PGM defines X as conditionally dependent on its parent variables $\text{Pa}(X)$, the FSM transition ω^X must take those same parents as inputs. The transition then produces x', the updated value of x in the next FSM timestep. Formally, let $\mathcal{X}$ denote the set of all non-time-indexed random variables in the PGM. For each $X \in \mathcal{X}$, we identify its associated transition as:

$$\omega^X : \text{Pa}(X) \Rrightarrow X \tag{5}$$

where $\text{Pa}(X)$ denotes the parent variables of X, and the notation $\Rrightarrow$ denotes the set of directed edges from the variables $\text{Pa}(X)$ to X. Figure 6 illustrates the set of transitions obtained, $\Omega = \{\omega^S, \omega^{U_{\text{in}}}, \omega^B, \omega^{B_{\text{in}}}, \omega^U, \omega^D\}$, by applying Eqn. (5) to our digital twin PGM. For example, the transition $\omega^D : (D, U, B_{\text{in}}) \Rrightarrow D$ captures that the digital state at the next step is determined by the digital state current value, the current computed control, and the most recently received observational data. As another example, the transition $\omega^{B_{\text{in}}} : (B, C^B) \Rrightarrow B_{\text{in}}$ illustrates the effect of our explicit model of digital twin communication: the value of the received observational data at the next step depends on both the transmitted value B and the current state of the communication channel C^B.

There is a well-established theoretical connection between PGMs and FSMs [30, 31, 32]. Both serve as automata for modeling system behavior, but

they differ in their treatment of uncertainty. PGMs encode distributions over variable dependencies, allowing probabilistic reasoning. FSMs, in contrast, describe possible sequences of state transitions without assigning explicit probabilities [33]. In this symbolic formulation, we aim not to estimate the *probability*, but rather the *possibility* whether a system can reach undesirable states. Since all core variables of interest are present in the digital twin PGM, the resulting digital twin FSM is complete with respect to modeling state evolution — every essential system variable is represented. While auxiliary variables (e.g., counters, timers, freshness flags) may be introduced during implementation to support model checking, the FSM's structure remains grounded in the dependencies encoded in the PGM, ensuring consistency between both probabilistic and logical representations.

We now define abstractions for the transition actions of system variables. Several variables of the FSM depend on functions that capture the output of complex stochastic or continuous processes in the PGM. To construct transition actions, we introduce two additional abstraction guidelines to abstract how nondeterministic and continuous-time processes are transformed into symbolic transition actions in the FSM.

Abstraction Guideline IV: Abstraction of nondeterministic functions. To abstract a nondeterministic process, we first discretize the input and output domains into $\hat{I}$ and $\hat{\mathcal{O}}$, respectively. We then identify a set of *guard conditions* $\{G_i\}_{i=1}^{K}$, where each G_i is a logical predicate (e.g., a threshold condition such as $d < \texttt{Threshold}$) that evaluates to either true or false. For each guard G_i, we assign a corresponding set of possible outputs $O_i = \{o_{i1}, o_{i2}, \ldots\} \subseteq \hat{\mathcal{O}}$. The abstraction then nondeterministically selects an output $o \in O_i$, denoted as:

$$
\begin{aligned}
x' = & \texttt{IF } G_1(\cdot) \\
& \quad \vee_{o \in O_1} o \\
& \texttt{ELSE IF } G_2(\cdot) \\
& \quad \vee_{o \in O_2} o \\
& \ldots \\
& \texttt{ELSE} \\
& \quad \vee_{o \in O_K} o
\end{aligned}
\tag{6}
$$

Here, the notation $\vee_{o \in O_i} o$ denotes choosing a single element o from the set of possible outputs O_i.

Abstraction Guideline V: Atomicity of actions. In the digital twin FSM, we model all actions defined within a *single* transition as occurring within a single discretized time step (see Abstraction Guideline II). It is sometimes desirable to treat *multiple* transitions as if they occur within the same atomic time step, for example, to simplify the model or to capture application-specific semantics. Let ω^X and ω^Y denote transitions that update variables x and y in the FSM, respectively. We say that ω^X and ω^Y are *mutually atomic* if they always execute together within the same discrete time step. Formally, we denote this by:

$$\text{Mutually atomic}(\omega^X, \omega^Y) \coloneqq \omega^X \ \wedge \ \omega^Y \tag{7}$$

where the conjunction $\wedge$ joins both transitions as occurring simultaneously, since a transition is indivisible. For example, although sensor data is physically generated first and then transmitted via Bluetooth circuitry after some delay, a digital twin FSM specification may abstract away this granularity and treat generation and transmission as mutually atomic. Whether to impose mutual atomicity is ultimately a modeling decision, guided by verification goals and the application context.

Table 2 details actions for each transition ω^X. The actions shown are illustrative examples: the specific guard conditions and the scope of non-deterministic choices are application-dependent, and may include additional guards or alternative choices beyond those depicted here.

Transition Name	**Transition Abstraction**
Physical System Transitions	
Physical state $\omega^S \colon (S, U_{\text{in}}) \Rrightarrow S$	$\omega^S \coloneqq s' =$ `IF` $(u = 3) \wedge (s <= 50)$ $(s-1) \vee (s-5)$ `ELSE` $s \vee (s-1)$ Here, $(u = 3) \wedge (s \leq 50)$ illustrates an example of a guard condition, while $s-1$ and $s-5$ represent possible outputs associated with the two input regions defined by that guard, per Eqn. (6).

Observational data
$\omega^{B^m} : S \Rrightarrow B$

$$\omega^{B^m} \coloneqq \; b^{m\prime} = s + (\epsilon_1 \vee \epsilon_2 \vee \cdots) \\ \wedge\, c^{B,m\prime} = \texttt{Append}(c^{B,m}, b^{m\prime})$$

where $\epsilon_i \in \mathbb{Z}$ are possible noise values for sensor m. Generation and queue insertion are modeled as mutually atomic.

Control received
$\omega^{U_{\text{in}}} : (C^U, U) \Rrightarrow U_{\text{in}}$

$$\omega^{U_{\text{in}}} \coloneqq \; u'_{\text{in}} = c^U[k] \\ \wedge\, \texttt{Remove}(c^U, k)$$

for some nondeterministically chosen $k \in K$, where $K = \{0, \ldots, \texttt{len}(c^U) - 1\}$. This is an instance of the general message-receive rule in Eqn. (3).

Digital Twin Transitions

Observational data received
$\omega^{B^m_{\text{in}}} : (C^B, B) \Rrightarrow B^m_{\text{in}}$

$$\omega^{B^m_{\text{in}}} \coloneqq \; b^{m\prime}_{\text{in}} = c^{B,m}[k] \\ \wedge\, \texttt{Remove}(c^{B,m}, k)$$

for each $m \in \{1, \ldots, M\}$ and some nondeterministically chosen $k \in K_m$, where $K_m = \{0, \ldots, \texttt{len}(c^{B,m}) - 1\}$. This is an instance of the general message-receive rule in Eqn. (3).

Digital state
$\omega^D : (D, U, B_{\text{in}}) \Rrightarrow D$

$$\begin{aligned} \omega^D \coloneqq d' = \texttt{IF}\;\; & (d \le 50) \wedge (u = 3) \wedge (\forall m \colon b^m <= 55) \\ & (d-5) \vee \ldots \vee (d-10) \\ \texttt{ELSEIF}\;\; & \forall m \colon (b^m \ge 35) \wedge (b^m \le 75) \\ & (d-1) \vee (d-2) \\ \texttt{ELSE}\;\; & \\ & d \vee (d-1) \end{aligned}$$

Here, the two example guard conditions $\big((d \le 50) \wedge (u = 3) \wedge (\forall m \colon b^m <= 55)\big)$ and $\big(\forall m \colon (b^m \ge 35) \wedge (b^m \le 75)\big)$ partition the input space into three regions, each with a possible set of outputs.

Control $\omega^U : D \Rrightarrow U$	$\omega^U \coloneqq u' = \wedge\ \texttt{IF}\ \ d \geq 55$ $u = 3$ $\texttt{ELSE}$ $u = 2 \vee 3$ $\wedge\ c^{U\prime} = \texttt{Append}(c^U, u')$ Here, $d \geq 55$ serves as a guard condition: if it holds, the transition deterministically outputs $u = 3$; otherwise, the output is chosen nondeterministically between $u = 2$ and $u = 3$. Generation and queue insertion are modeled as mutually atomic.

Table 2: FSM transition actions for each transition ω^X. The transitions shown are illustrative examples: the specific guard conditions and set of nondeterministic choices are application-dependent.

2.4. Initial and Termination States in the Digital Twin FSM

The initial state predicate defines the set of allowed starting configurations for the FSM. In other words, it specifies the values that each variable must take at the beginning of execution. We write the initial state predicate $\mathcal{I}$ as:

$$\begin{aligned} \mathcal{I} \coloneqq \quad & (s = s_0) \wedge (d = d_0) \wedge (u = u_0) \\ & \wedge (u_{\text{in}} = u_{\text{in},0}) \wedge (c^U = c_0^U) \\ & \wedge \big[\forall m : \\ & \qquad (b^m = b_0^m) \wedge (b_{\text{in}}^m = b_{\text{in},0}^m) \wedge (c^{B,m} = c_0^{B,m})\big] \end{aligned} \tag{8}$$

Each clause of the form $(x = x_0)$ specifies the initial value of a variable x at time $t = 0$. Thus, Eqn. (8) states that a valid initial state is one in which every variable of the FSM — physical state, digital state, control variables, and for each sensor its data and channel contents — begins at its specified initial value.

The digital twin FSM is said to reach *termination* when the system satisfies a symbolic predicate $\mathcal{T}$, which defines the allowable terminal states of the system. Unlike the initial predicate $\mathcal{I}$, which uses fixed values, the termination predicate may include thresholds, inequality guards, or logical stopping conditions. We express it abstractly as:

$$\mathcal{T} \coloneqq \texttt{TerminationGuard}(s, d, b, b_{\text{in}}, u, u_{\text{in}}, c^u, c^b) \tag{9}$$

The guard $\texttt{TerminationGuard}(\cdot)$ is a predicate evaluating to true or false that may include conditions such as $s \leq 10$, $\forall m \colon b^m < 5$, or completion criteria based on elapsed time, number of iterations, or convergence thresholds.

2.5. Fairness Assumptions in the Digital Twin FSM

Fairness specifies assumptions about how the system selects among multiple possible transitions at each step. In models with nondeterminism, it is often possible for some transitions to be repeatedly enabled but never taken. A transition is said to be enabled if its guard condition is satisfied in the current state — that is, it is eligible to occur. Fairness assumptions prevent pathological stalling, such as repeatedly ignoring a control computation or a data reception step that should logically proceed. We define three types of fairness labels for transitions in the digital twin FSM:

1. **Weak fairness** $\mathrm{WF}(\omega)$, defined as:

$$\mathrm{WF}(\omega) \coloneqq \Box\, \texttt{enabled}(\omega) \Rightarrow \Diamond\, \texttt{executed}(\omega) \tag{10}$$

   This states that if ω is always enabled, it must eventually be executed. Here, $\Box$ is the "always" operator and $\Diamond$ is the "eventually" operator.

2. **Strong fairness** $\mathrm{SF}(\omega)$, defined as:

$$\mathrm{SF}(\omega) \coloneqq \Box\Diamond\, \texttt{enabled}(\omega) \Rightarrow \Box\Diamond\, \texttt{executed}(\omega) \tag{11}$$

   Here, $\Box\Diamond\texttt{enabled}(\omega)$ expresses that ω is intermittently enabled over the system execution. Eqn. (11) states that if ω is enabled infinitely often, then it must also be executed infinitely often.

3. **Unfair** $\mathrm{UF}(\omega)$, defined as:

$$\mathrm{UF}(\omega) \coloneqq \neg\, \texttt{WF}(\omega) \wedge \neg\, \texttt{SF}(\omega) \tag{12}$$

   This represents transitions for which no fairness guarantees are assumed. The negation operator $\neg$ means "not"; thus, $\texttt{UF}(\omega)$ holds when ω is neither weakly fair nor strongly fair. Such transitions may be enabled, and yet may never be executed.

In the digital twin FSM, we impose fairness conditions by considering which behaviors are plausible in the physical and digital world, and selecting

FSM Transition	Fairness Assumption
Physical System Fairness Assumptions	
Physical state ω^S	$\mathrm{WF}(\omega^S)$: Weak fairness ensures evolution occurs when the physical system is persistently active.
Observational data ω^{B^m}	$\mathrm{WF}(\omega^{B^m})$: Sensor emits data if functioning. Weak fairness allows skipping emission when the sensor is offline or faulty.
Control received $\omega^{U_{\mathrm{in}}}$	$\mathrm{SF}(\omega^{U_{\mathrm{in}}})$: Control messages may be delayed due to intermittent channel unavailability. Strong fairness ensures controls are eventually received when the channel and receiver become enabled.
Digital Twin Fairness Assumptions	
Observational data received $\omega^{B^m_{\mathrm{in}}}$	$\mathrm{SF}(\omega^{B^m})$: Strong fairness ensures sensor data are eventually received when the channel and receiver become enabled.
Digital state ω^D	$\mathrm{SF}(\omega^D)$: Digital state must eventually update when repeatedly able. Strong fairness prevents indefinite stalling despite available inputs.
Control ω^U	$\mathrm{SF}(\omega^U)$: Control is eventually computed when repeatedly enabled. Strong fairness ensures computations are not indefinitely deferred.

Table 3: Fairness assumptions for transitions in the digital twin FSM. These labels specify whether a transition must eventually occur when enabled, and prevent unrealistic stalling in the presence of nondeterminism.

the fairness assumption that best reflects those behaviors. Table 3 summarizes the fairness assumptions assigned to each transition. These constraints determine whether a transition must eventually occur when enabled, and help rule out unrealistic stalling. The complete fairness predicate $\mathcal{F}$ for the digital twin FSM is defined as:

$$\mathcal{F} = \mathrm{WF}(\omega^S) \wedge \mathrm{WF}(\omega^{B^m}) \wedge \mathrm{SF}(\omega^{U_{\mathrm{in}}}) \wedge \mathrm{SF}(\omega^{B^m_{\mathrm{in}}}) \wedge \mathrm{SF}(\omega^D) \wedge \mathrm{SF}(\omega^U) \quad (13)$$

3. Deriving Verification Goals from the Digital Twin FSM

We define the digital twin verification problem as the task of proving that a formal model of the system satisfies a set of correctness properties $\mathcal{P} =$

$\{P_1, P_2, \dots\}$, under a specified initial condition $\mathcal{I}$, next state predicate $\mathcal{N}$, and fairness assumptions $\mathcal{F}$. These correctness properties fall into two major categories: (1) *Safety properties*, which assert that the system never enters an incorrect state (e.g., "a valve is never open when pressure exceeds threshold"); and (2) *Liveness properties*, which assert that a desired condition eventually occurs (e.g., "every control message is eventually executed"). We propose that all digital twins must satisfy a canonical liveness property reflecting their core function: the digital state must eventually synchronize with the physical state. Formally, we define the digital twin canonical liveness property as:

$$P_1 \coloneqq \Diamond \left(\text{dist}(D, S) \leq \epsilon\right) \tag{14}$$

where dist is an application-defined distance function and ϵ is an acceptable error margin. This property asserts that the digital twin must, at some point, converge to a sufficiently accurate representation of the physical system.

However, verifying P_1 directly is often infeasible due to its broad scope and the limited observability of the physical state S. To address this, we introduce a structured decomposition strategy that derives P_1 as a logical consequence of more granular and verifiable sub-properties. The decomposition begins with two intermediate properties that isolate the virtual and physical sources of synchronization error:

$$P_1 \Leftarrow P_2 \wedge P_3 \tag{15}$$

Here, P_2 states that the digital state must reliably track the physical state, while P_3 asserts that the physical system must respond accurately to the digital twin's issued control commands. Although P_2 and P_3 remain high-level, this split reflects a natural partition between the virtual and physical realms. Note that although the physical state S evolves in reality according to natural dynamics, in the digital twin FSM we represent that evolution as a transition predicate ω^S (see Table 2). This does not mean we verify nature, but rather that we verify our model of the physical evolution. This model is necessary because the FSM is a complete specification of the digital twin system: both physical and digital elements are captured in the same formalism. Verification then ensures that our abstraction of S evolves consistently, respects its intended discretization, and is based on an appropriate underlying physical model.

To refine P_2 and P_3 further, we apply a transition-based decomposition strategy grounded in the structure of the FSM. Recall from Eqn. (5) that

each FSM transition for variable x is written as:

$$\omega^X : \mathrm{Pa}(X) \Rrightarrow X$$

For each transition, we define two verification obligations:

1. **Input Consistency**: The predicate $\texttt{Consist}(\mathrm{Pa}(x))$ asserts that the inputs to ω^X are both individually valid and mutually consistent. For example, in the transition ω^D for updating the digital state, the inputs u, d, and b_{in} must not only be valid individually, but also correspond to approximately the same point in the system's execution. This ensures that updates are computed from coherent, non-stale inputs rather than from mismatched or invalid data.

2. **Correctness of the Update Rule**: The predicate $\texttt{Corr}(\omega^X)$ asserts that the transition logic for updating x faithfully implements intended behavior. This correctness obligation can be further decomposed into two parts:

   (a) **Abstraction Logic Correctness**, denoted $\texttt{CorrAbst}(\omega^X)$: The FSM transition logic correctly encodes the intended behavior at the level of symbolic abstraction.

   (b) **Model Correctness**, denoted $\texttt{CorrModel}(\mathcal{F}^X)$: The underlying function $\mathcal{F}$ (see Abstraction Guideline IV), which the transition ω^X abstracts, must faithfully represent the relevant real-world dynamics. If $\mathcal{F}$ is mismodeled, the resulting FSM will yield behaviors that systematically violate desired properties.

The full verification condition for the transition ω^X is:

$$\texttt{Verify}(\omega^X) \coloneqq \texttt{Consist}(\mathrm{Pa}(X)) \wedge \texttt{CorrAbst}(\omega^X) \wedge \texttt{CorrModel}(\mathcal{F}^X) \quad (16)$$

In many cases, $\texttt{CorrModel}(\mathcal{F}^X)$ is not verified directly within our framework but instead treated as an external assumption. In cases where no underlying model is abstracted, the $\texttt{CorrModel}(\mathcal{F}^X)$ term may be vacuously true or omitted. Our strategy enables formal guarantees about system-level behavior in the presence of bidirectional digital-physical interaction, while incorporating correctness assurances from existing domain-specific work. For instance, our verification strategy can incorporate guarantees from certified control synthesis [8, 6] or verified machine learning methods [34, 35].

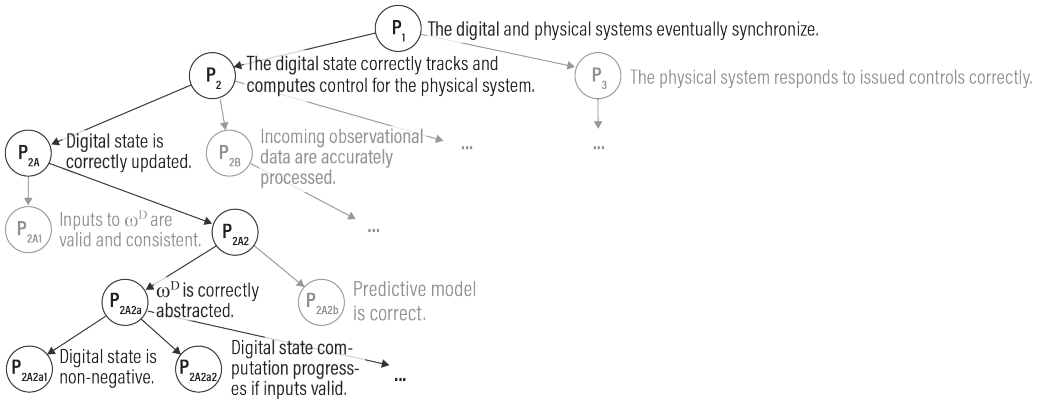


Figure 7: Directed acyclic graph of digital twin verification obligations. The figure highlights o ne refinement path for verifying P_{2A}, illustrating how obligations such as $\texttt{Corr}(\cdot)$ can be decomposed into increasingly granular sub-properties.

Under this verification decomposition framework, the resulting property set $\mathcal{P} = \{P_1, \ldots, P_N\}$ induces a directed acyclic graph $\mathbb{G}$ of logical dependencies, where N is the total number of properties (nodes) in $\mathbb{G}$. Properties with the same topological rank in $\mathbb{G}$ share consistent abstraction levels and natural language interpretations. Each property in $\mathcal{P}$ is either (1) verified internally via the digital twin's FSM model or (2) assumed to hold based on external guarantees. Figure 7 illustrates a representative subgraph of $\mathbb{G}$, where externally verified properties — such as P_{2A2b} — are explicitly marked. Table 4 lists the verification obligations derived from each FSM transition. Each transition ω^X is decomposed into consistency and correctness properties. The $\texttt{CorrAbst}(\omega^X)$ obligations often encode detailed transition-specific constraints (e.g., guard conditions, bounds, or monotonicity rules), which are too granular to display here; their full formalizations are given in Appendix Appendix A.

FSM Transition	**Verification Goals**
ω^D: $(D, U, B_{\text{in}}) \Rrightarrow D$	P_{2A}: Digital state is correctly updated. P_{2A1}: $\texttt{Consist}(d, u, b_{\text{in}})$: Input values of d, u and b_{in} do not differ by more than Δ_t time window. P_{2A2}: $\texttt{Corr}(\omega^D)$ P_{2A2a}: $\texttt{CorrAbst}(\omega^D)$: ω^D is correctly abstracted. P_{2A2b}: $\texttt{CorrModel}(\mathcal{F}^D)$: Predictive model $\mathcal{F}^D$ is correct.

$\omega^{B^m_{\text{in}}}$: $(C^{B,m}, B^m) \Rrightarrow B^m_{\text{in}}$	P_{2B}: Incoming observational data are correctly processed. P_{2B1}: $\texttt{Consist}(c^{B,m}, b^m_{\text{in}})$: Data b^m is received from channel $c^{B,m}$. P_{2B2}: $\texttt{Corr}(\omega^{B^m_{\text{in}}})$ P_{2B2a}: $\texttt{CorrAbst}(\omega^{B^m_{\text{in}}})$: b^m_{in} steps correctly. P_{2B2b}: $\texttt{CorrModel}(\mathcal{F}^{B^m_{\text{in}}})$: Bluetooth receiver is functional.
ω^U: $D \Rrightarrow U$	P_{2C}: Control is correctly computed. P_{2C1}: $\texttt{Consist}(d)$: Digital state input d is valid. P_{2C2}: $\texttt{Corr}(\omega^U)$ P_{2C2a}: $\texttt{CorrAbst}(\omega^U)$: u steps correctly. P_{2C2b}: $\texttt{CorrModel}(\mathcal{F}^U)$: Control logic $\mathcal{F}^U$ is correct.
ω^S: $(S, U) \Rrightarrow S$	P_{3A}: Modeled physical state correctly evolves. P_{3A1}: $\texttt{Consist}(s, u)$: Values for S and U do not differ by more than Δ_t time window. P_{3A2}: $\texttt{Corr}(\omega^S)$ P_{3A2a}: $\texttt{CorrAbst}(\omega^S)$: s steps correctly. P_{3A2b}: $\texttt{CorrModel}(\mathcal{F}^S)$: Underlying physics model $\mathcal{F}^S$ reflects real-world dynamics.
$\omega^{U_{\text{in}}}$: $(C^U, U) \Rrightarrow U_{\text{in}}$	P_{3B}: Incoming control commands are correctly processed. P_{3B1}: $\texttt{Consist}(c^U, u)$: Control message u is received from channel c^U. P_{3B2}: $\texttt{Corr}(\omega^{U_{\text{in}}})$ P_{3B2a}: $\texttt{CorrAbst}(\omega^{U_{\text{in}}})$: u_{in} steps correctly. P_{3B2b}: $\texttt{CorrModel}(\mathcal{F}^{U_{\text{in}}})$: Bluetooth receiver is functional.
ω^{B^m}: $S \Rrightarrow B$	P_{3C}: Modeled sensor emissions accurately reflect physical state. P_{3C1}: $\texttt{Consist}(s)$: Physical state input s is valid. P_{3C2}: $\texttt{Corr}(\omega^{B^m})$ P_{3C2a}: $\texttt{CorrAbst}(\omega^{B^m})$: b^m steps correctly. P_{3C2b}: $\texttt{CorrModel}(\mathcal{F}^{B^m})$: Sensor is functional.

Table 4: Verification sub-properties derived from FSM transitions. For each transition ω^x, goals are split into consistent checks ($\texttt{Consist}(\cdot)$) on inputs and correctness checks ($\texttt{Corr}(\cdot)$) on transition logic. Correctness is further divided into obligations on FSM abstraction logic ($\texttt{CorrAbst}(\cdot)$) and on the underlying real-world model ($\texttt{CorrModel}(\cdot)$).

4. Results

We first describe the experimental setup of an unmanned aerial vehicle (UAV) digital twin. We then present implementation details and model checking results. Finally, we illustrate the iterative nature of formal specification through a case study highlighting a property violation.

4.1. Experimental Setup: Unmanned Aerial Vehicle Digital Twin

We demonstrate our approach on a UAV case study, following [20]. The UAV has been physically realized with custom Bluetooth sensors [36], while predictive control methods are implemented offboard [37]. The virtual representation consists of a collection of Python ROS modules that ingest observational data, compute dynamic control commands, and maintain a digital estimate of the UAV's structural health. Our contribution is a formally verified specification of a digital twin system that integrates these components into a coherent architecture.

To instantiate the abstract FSM formulation introduced in this paper, we assign concrete representations of the physical, digital, and communication variables in the context of the UAV. Table 5 summarizes the FSM variables and their corresponding representations in the UAV case study. The variables are grouped into three categories: UAV variables, which map to the physical system; digital twin variables, which map to the virtual representation; and communication channel variables.

FSM Variable	**UAV Case Study Representation**
UAV Variables	
Physical state s	Represents the structural health of the UAV, taking on a value from the set $\{0, \ldots, s_0\}$, where s_0 denotes the initial starting health, health cannot increase, and 0 denotes a failed UAV.
Observational data b^m	Denotes the emission from sensor m, modeled as $b^m = s + \epsilon_m$ (see Table 2), where ϵ_m is sensor noise.
Control received u_{in}	Represents the maneuver command received by the UAV, where $u_{\text{in}} = 2$ indicates a conservative maneuver and $u_{\text{in}} = 3$ indicates an aggressive maneuver.
Digital Twin Variables	
Observational data received b^m_{in}	Represents the sensor data received from each sensor m via the communication channel.

Digital state d	Represents the UAV's structural health estimate produced by the digital twin's computational model, taking on a value from the set $\{0, \ldots, s_0\}$.
Control u	Denotes the dynamic control computed by the control module, where $u = 2$ indicates a conservative maneuver and $u = 3$ indicates an aggressive maneuver.
Communication Channel Variables	
Observational data channel $c^{B,m}$	Represents the communication channel for sensor m, where each Bluetooth sensor transmits on a distinct channel.
Control channel c^U	Represents the Bluetooth channel used to transmit control commands to the UAV.

Table 5: FSM variables and their case study representations in the UAV digital twin.

Table 6 summarizes the model- and experiment-level parameters used in our study. These include structural parameters (e.g., number of sensors), initial conditions for all FSM state variables, termination criteria, and communication constraints (e.g., maximum channel delay). Parameter ranges indicate the values explored in experiments; fixed constants represent constraints or verification bounds. Where relevant, the table also provides a brief description of how each parameter is applied in the model.

Term	**Parameter Values**
Number of sensors M	Number of sensors in the system, varied over $\{1, \ldots, 5\}$ in experiments.
Sensor noise ϵ_m	Sensor noise ϵ_m of sensor m, varied over $\{-2, \ldots, 2\}$ in experiments.
Initial state $\mathcal{I}$	We initialize all FSM state variables over a range of values: $$\mathcal{I} = \begin{Bmatrix} s_0 \in \{1, \ldots, 10\}, \quad b_0^m = \varnothing, \quad u_{\text{in},0} = \varnothing, \\ b_{\text{in},0}^m = \varnothing, \quad d_0 \in \{s_0 - 1,\ s_0,\ s_0 + 1\}, \\ u_0 = \varnothing, \quad c_0^U = [\,], \quad c_0^{B,m} = [\,] \end{Bmatrix}$$ Here, $\varnothing$ denotes an empty value and $[\,]$ an empty queue.
Maximum execution steps $e_{\max}$	Maximum number of steps the UAV may execute before forced termination, varied over $\{1, \ldots, 20\}$ in experiments.

Termination $\mathcal{T}$	Occurs if: (i) the UAV has executed at least $e_{\max}$ steps, or (ii) all received observations and the digital state estimate indicate non-positive state, suggesting structural failure. $$\begin{aligned}\mathcal{T} \coloneqq \vee\ & \texttt{NumExec} \geq e_{\max} \\ \vee\ & (\forall m \in \{1 \dots M\}, \\ & \wedge\ d \leq 0 \\ & \wedge\ (b_{\text{in}}^m \leq 0 \vee b_{\text{in}}^m = \varnothing)\end{aligned}$$
Maximum message delay K of channel c	We bound the delivery delay of any message in channel c by: $$K = \min\big(\texttt{len}(c),\, 3\big), \quad K \geq 0$$ where $\texttt{len}(c)$ is the current number of messages in channel c.

Table 6: Model and experiment parameters used in the digital twin FSM.

4.2. From Specification to Model Checking

We use the TLA Model Checker (TLC) [26] because it is the standard explicit-state model checker distributed with the TLA+ toolchain and is designed to directly execute TLA specifications. This choice follows from the role of TLA in our framework. As discussed in Section 2.2, our digital twin abstraction is formulated as an FSM whose behavior is defined by an initial predicate, a next-state predicate, and fairness assumptions. TLC is well suited to this formulation because it can exhaustively enumerate all reachable states generated by these predicates and check correctness properties over the resulting transition system.

Other model checkers and formal verification tools could also be used for digital twin verification if the digital twin FSM were translated into their input languages. We use TLC here because it operates natively on the TLA specification, supports the fairness assumptions used in our model, produces explicit counterexample traces when properties fail, and requires no separate translation step between the formal specification and the checked model. These features are especially useful for the iterative specification workflow illustrated in Section 4.3, where counterexamples are used to refine transition guards and correctness properties. To assess whether the goals defined in Section 3 are satisfied under the parameter ranges listed in Table 6, we check the resulting finite-state specification with TLC.

Overview of the TLC Model Checker. Given an initial-state predicate $\mathcal{I}$, a next-state predicate $\mathcal{N}$, and fairness conditions $\mathcal{F}$, TLC performs an exhaus-

tive state-space exploration as follows:

1. Identify all states that satisfy $\mathcal{I}$.
2. For each such state, apply $\mathcal{N}$ to enumerate all admissible successor states.
3. Iteratively apply Step 2 to newly discovered states until no additional states are reachable or until a property violation is detected.
4. For each specified property (e.g., those in Table 4), verify that it holds in all reachable states and across all admissible execution paths.

TLC enumerates all possible interleavings of transitions within the specified bounds, thereby detecting violations that may arise only under infrequent or adversarial timing scenarios — cases that are typically not observed in conventional simulation studies.

As an example, Fig. 8 presents a simplified implementation of the transition $\omega^{B^m_{\text{in}}}$, denoted `DT_ReceiveObsDelayed`. This transition models the asynchronous receipt of sensor messages by the digital twin, including the possibility of communication delays. Specifically:

- `c_b[m]` is the message queue for sensor m. The function first checks whether the queue is non-empty and whether the specified index `k` is valid.
- The helper function `IsMessageUpToDate` (defined in a separate module) enforces the constraint that the candidate message's timestamp is strictly newer than previously processed ones. A valid message is appended to the processed observations stream `b_in[m]` and removed from the queue via `Remove` (also separately defined). Stale messages are also removed from the queue but do not alter the processed observations.
- The `UNCHANGED` clause explicitly declares which state variables remain unaffected by this transition, aiding readability and avoiding unintended state modifications.

When combined with all other transition definitions in the specification (see Table 2), the rule in Fig. 8 contributes to the complete state-transition system executed by the TLC model checker. TLC steps through each transition in all possible orders, generating every reachable configuration of the digital twin FSM. The result is a combinatorially large state space that reflects all possible interleavings of action ordering, message delay, and nondeterministic choice scenarios.

We visualize this state space in Fig. 9 as a directed acyclic graph (DAG), where each vertex represents a unique state — a complete assignment of

```
DT_ReceiveObsDelayed(m, k) ==
    IF c_b[m] # << >>
    THEN
        IF k \in DOMAIN c_b[m]
        THEN
            IF IsMessageUpToDate(b_in[m], c_b[m][k]["t"])
            THEN
                /\ b_in' = [b_in EXCEPT ![m] = Append(b_in[m], c_b[m][k])]
                /\ c_b'  = [c_b  EXCEPT ![m] = Remove(c_b[m], k)]
            ELSE
                /\ c_b'  = [c_b  EXCEPT ![m] = Remove(c_b[m], k)]
                /\ UNCHANGED << b_in >>
        ELSE
            UNCHANGED << b_in, c_b >>
    ELSE
        ...
```

Figure 8: Implemented $\omega^{B_{\text{in}}^m}$ transition, which models the asynchronous receipt of sensor observations with possible message delay. This implementation is executed by the TLC model checker to verify that properties hold across all possible execution paths.

values to all variables — and edges depict transitions between these states. This graph is inherently a DAG, as the specification includes a verified guarantee of termination. In Fig. 9, terminating states are highlighted in orange, while ongoing states are in black. The graph's initial state, depicted as a blue vertex (1), bifurcates into two principal pathways: the physical system's processes (2) and the digital twin's processes (3). For example, one path proceeds from state (2) to (4), then to (7), and ultimately to (15), where termination occurs upon meeting one of the conditions in $\mathcal{T}$.

The combinatorial branching illustrated in Fig. 9 explains why exhaustive model checking can be computationally demanding. Each additional transition, sensor, or communication delay increases the number of possible interleavings, expanding the number of reachable states. To quantify these effects, we perform a series of parameter sweeps in which one model parameter is varied at a time while all others are held at their baseline values. For each configuration, we record the number of distinct states, the total number of states explored (including revisits), and the wall-clock runtime as reported

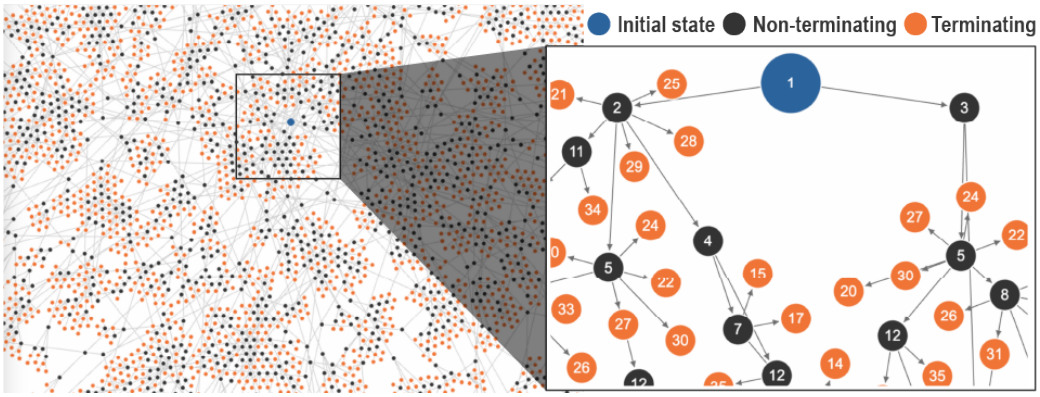


Figure 9: Visualization of state space: nodes represents states and edges represent transitions from state to state.

Specification	**Distinct States**	**Total States**	**Runtime (h)**
Baseline	12 551 574	33 960 246	15
$+1$ s $(s_0 = 3)$	24 668 110	66 833 826	45
$+1$ m $(M = 3)$	13 534 045	41 966 573	48
$+1$ k $(K = 3)$	15 307 358	50 720 696	39
± 1 ϵ $(E_m = \{-2, \ldots, 2\})$	15 804 834	42 619 510	17
$+1$ transition $(\omega^{S,1}, \omega^{S,2})$	1 227 202	3 231 322	2

Table 7: Effect of varying individual model parameters on state space complexity, and runtime. The baseline configuration is $s_0 = 2$, $M = 2$, $K = 2$, $\epsilon = \{-1, 0, 1\}$, ω^S. Distinct states are unique configurations; total states include revisits.

by the TLC model checker[1].

These results illustrate several non-intuitive aspects of scalability in the proposed approach. The method scales to finite-state abstractions containing tens of millions of reachable states, as shown by the baseline model and the parameter variations in Table 7. This scale is sufficient to verify nontrivial finite-state abstractions of digital twin behaviors involving multiple sensors, asynchronous physical and virtual evolution, communication delay, and safety and liveness properties. The results also show the expected state-space

[1]Experiments were run on Ubuntu 22.04.5 LTS (Jammy) with 64 AMD EPYC 7H12 cores (128 threads) and 256 GB RAM, using TLA v1.8.0.

growth associated with model checking. Increasing the initial physical-state range, the number of sensors, or the maximum message delay expands the number of possible states and interleavings, because each additional value or communication event introduces new combinations of physical state, digital state, channel contents, and transition orderings.

Scalability is governed by the number of behaviorally distinct cases represented in the finite-state abstraction. Parameter bounds are therefore chosen to expose the modes, thresholds, message-ordering behaviors, and failure cases relevant to the verification objective. This is consistent with the small-scope perspective in formal analysis: many specification and implementation errors can be exposed in small bounded instances [38]. Thus, increasing a parameter bound is useful only when the additional values introduce behaviorally distinct cases for the verification objective. For example, if the relevant property depends on whether the physical state is positive, below a threshold, or failed, then verifying representative values in those regions may be more informative than exhaustively increasing s_0 over a larger numerical range.

These insights suggest an iterative workflow in which engineers begin with a coarse abstraction that captures core correctness properties, and then selectively refine the abstraction only where additional fidelity affects the property being checked. Such refinement may include increasing queue bounds, adding sensors, widening uncertainty ranges, or splitting atomic transitions into finer-grained actions. The non-monotonic result in Table 7, where splitting the physical-state transition reduced the reachable state space, shows that scalability depends not only on the number of modeled variables but also on the structure of the specification and the way concurrency is represented. Similar scalability concerns arise in other model-checking applications for safety-critical engineered systems, where state-space explosion limits naive exhaustive exploration and motivates careful choices about abstraction and communication structure [39]. Thus, scalability is best understood as a property of the chosen abstraction, verification objective, and model-checking structure, rather than as a function only of parameter ranges or the number of modeled system elements.

We also investigated how modeling assumptions about atomicity affect state space size. Recall from Eqn. (7) that an atomic transition is one that appears to occur in a single, indivisible step. In the baseline model, the physical system's physical state evolution is modeled as one such atomic transition: in a single step, it reads the most recent received control u_{in},

executes it, and updates the physical state s. To explore the effect of relaxing this assumption, we split this transition into two separate, non-atomic steps, introducing an explicit state variable u_e to store the last executed control. This allows intermediate states where a control has been executed but its physical effects have not yet been applied, enabling potential concurrency with other processes.

Baseline (Atomic)	**Split (Non-atomic)**
$\omega^S \coloneqq s' = \texttt{IF}\ \ (u = 3) \wedge (s \leq 50)$	$\omega^{S,1} \coloneqq u_e' = u_{\text{in}}$ (execute received control)
$\quad (s-1) \vee (s-5)$	$\omega^{S,2} \coloneqq s' = \texttt{IF}\ \ (u_e = 3) \wedge (s \leq 50)$
$\texttt{ELSE}$	$\quad (s-1) \vee (s-5)$
$\quad s \vee (s-1)$	$\texttt{ELSE}$
	$\quad s \vee (s-1)$

Table 8: Comparison of the baseline atomic transition with the split non-atomic variant.

Intuitively, exposing such intermediate states might be expected to *increase* the number of reachable states. Surprisingly, model checking revealed the opposite: reachable states dropped from 12 million to 1 million. We hypothesize that this reduction occurs because TLC can apply more aggressive invariant simplifications and state-merging optimizations when transitions are finer-grained, pruning large portions of the search space. This result highlights that atomicity assumptions do not have a straightforward monotonic relationship with verification complexity; the interaction between model structure and model checker heuristics can produce unintuitive scaling behavior. A summary of model parameter variations and their resulting state space sizes is provided in Table 7.

4.3. Iterative Specification: Case Study in Property Violation

Although the preceding sections present the specification and its properties in their final form, arriving at that point was an inherently iterative process. We repeatedly refined both the model and the properties in response to counterexamples uncovered by model checking, with each iteration exposing subtle implementation flaws or missing constraints. To illustrate how this process unfolded in practice, we present here a representative case study: a freshness property for executed controls,

$$\Box\big[u_e \neq \varnothing \ \wedge\ u_e' \neq \varnothing \ \implies\ \texttt{time}(u_e') > \texttt{time}(u_e)\big] \tag{17}$$

This asserts that the timestamp of each newly executed control must be strictly newer than the previously executed one, ensuring that stale commands cannot overwrite fresher ones. Model checking revealed a counterexample, which unfolded as follows:

Step	**Transition**	**Description**	**State Values After Step**
1	Initial State $\mathcal{I}$	No control has been executed yet.	$u_e = \varnothing$
2	ω^S	Physical system executes default control in absence of received input.	$u_e \leftarrow \{t\colon 11,\ v\colon 12\}$
3	ω^U	Digital twin computes and emits a control.	$u = \{t\colon 1,\ v\colon 13\}$ $c^U = [\{t\colon 11,\ v\colon 13\}]$
4	$\omega^{U^{\text{in}}}$	Physical system receives the control.	$u_{\text{in}} = \{t\colon 1,\ v\colon 3\}$ $c^U = [\,]$
5	ω^S	**Violation**: Executes received control, violating freshness property as received control's timestamp of $t = 1$ is the same as previous executed control's.	$u_e = \{t\colon 1,\ v\colon 3\}$

Table 9: Execution trace showing a violation of the freshness property (see Eqn 17).

The progression of states leading to this violation is visualized in Fig. 10, where node `15` corresponds to the state in which the property was violated. The root cause was a missing timestamp validation in the $\omega^{U_{\text{in}}}$ transition: the physical twin accepted and executed a control without checking whether its timestamp was more recent than the last received command. This oversight, while seemingly straightforward in retrospect, was not immediately obvious during early specification development. To illustrate the fix, Eqn. (18) shows the original transition definition, which unconditionally updates the variable u_{in} with a command $c^U[k]$ from the queue. Eqn. (19) presents the corrected version, where we introduce a guard condition to ensure that only commands with strictly newer timestamps are accepted.

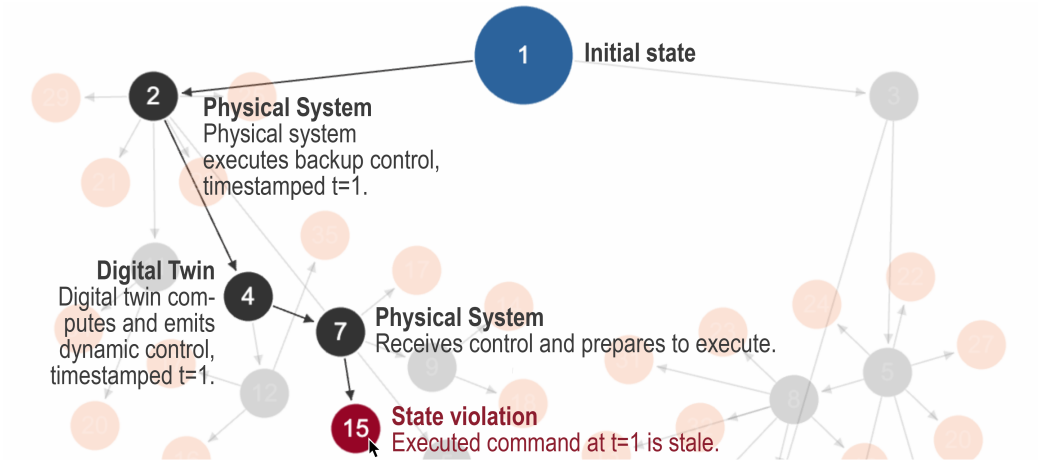


Figure 10: Model checker trace leading to a safety property violation. Initial state is highlighted in blue. Highlighted node (red) represents the state in which the property violation occurs due to stale control execution.

$$\omega^{U_{\text{in}}} := \wedge\, u'_{\text{in}} = c^U[k] \\ \wedge\, \texttt{Remove}(c^U, c^U[k]) \tag{18}$$

$$\omega^{U_{\text{in}}} := \texttt{IF time}(c^U[k]) > \texttt{time}(u_{\text{in}}) \\ \wedge\, u'_{\text{in}} = c^U[k] \\ \wedge\, \texttt{Remove}(c^U, c^U[k]) \tag{19}$$

This correction illustrates how the iterative cycle of specification, property formulation, and model checking enables us to detect and resolve errors at a higher level of abstraction, before they manifest in complex code. As system complexity increases, subtle lapses in temporal logic (e.g., forgetting to check timestamps) can lead to incorrect behavior that is difficult to identify through inspection or debugging.

4.4. Practical Considerations for Engineering Use

Applying the proposed verification methodology in an engineering setting requires several practical modeling decisions. The PGM formulation for digital twins was originally proposed in [20], while subsequent work has addressed learning graphical models from data [40]. Another consideration is the construction of an appropriate finite-state abstraction. Digital twin systems contain continuous variables, stochastic models, sensor streams, control policies, and communication mechanisms that cannot be represented directly in a formal verification framework without the abstraction method developed in Section 2.2. Using this abstraction method, engineers must select symbolic domains, time steps, queue bounds, message-delay limits, and termination

criteria that are sufficiently granular to capture the behaviors relevant to verification, while remaining coarse enough to avoid modeling details that are either assumed to be correct or irrelevant to the verification goals. These choices depend on the verification objective and are application-specific. For example, verifying freshness of received commands requires preserving timestamp information and communication ordering.

A second consideration is the distinction between implementation and specification. Engineering practice in digital twin development often emphasizes executable implementations, simulation-based analysis, and test-and-evaluation workflows [41]. By contrast, formal verification requires a specification that identifies the system states, admissible transitions, environmental assumptions, and correctness properties. This specification is written in a formal specification language, such as TLA, and checked by a model checker, such as TLC for TLA specifications. The specification is not executable implementation. Rather, it is an abstracted model of the implementation that preserves the behaviors relevant to the verification task. Establishing the correspondence between code and specification is itself a substantial topic within formal verification. As a result, after specification, engineers must still implement the system in code; ensuring that this implementation conforms to the verified specification is an additional assurance task beyond model checking the specification itself.

A third consideration is the formulation of correctness properties. Many engineering requirements are initially stated informally, for example, that the digital twin should remain synchronized with the physical system or that stale commands should not be executed. These requirements must be translated into precise correctness properties over the variables of the FSM; Section 3 proposes a method for deriving such properties. This translation often exposes ambiguities in the original engineering requirement. For instance, a requirement that "the most recent control should be used" requires a formal definition of recency, a representation of message timestamps, and a decision about how invalid or stale messages should be handled. The case study in Section 4.3 illustrates this process: model checking revealed that the original specification allowed a stale command to overwrite a fresher one, leading to a refinement of the transition guard for received controls. Although formulating correctness properties can require substantial effort, this effort is also one of the main practical benefits of formal verification: it requires informal engineering intent to be translated into precise, checkable specifications before correctness is evaluated [42, 43].

5. Conclusions

This paper introduces a formal verification methodology for digital twins that integrates probabilistic graphical modeling, finite state machine abstraction, and the Temporal Logic of Actions. By explicitly modeling distributed communication channels alongside physical and virtual components, our approach captures asynchronous evolution and communication uncertainty that are characteristic of real-world digital twins. The framework supports a systematic derivation of verification goals. We identify a top-level synchronization objective — requiring the physical and digital systems to align over time — and decompose it into transition-level obligations that are either discharged by model checking the specification or assumed from external guarantees. Applied to a UAV case study, the method shows how formal specification not only surfaces subtle errors, but also helps clarify key modeling choices such as which assumptions matter, how concurrency should be modeled, and what behaviors must be guaranteed.

Our findings underscore three points. First, correct component behavior does not imply system-level correctness in the presence of distribution; explicit communication modeling is essential. Second, verification is inherently iterative: co-evolving the specification and its properties exposed and resolved subtle flaws. Third, while state-space growth is a practical challenge in model checking, careful abstractions and fairness assumptions make exhaustive exploration both feasible and informative for nontrivial models. Taken together, these results show how formal verification can provide mathematical guarantees for digital twin systems operating under the distributed and asynchronous conditions of real-world deployments.

Acknowledgments

This work was supported in part by Department of Energy grant DE-SC002317, NASA cooperative agreement 80NSSC21M0071, and Air Force Office of Scientific Research grant FA9550-24-1-0327. We thank Akhil Bhimaraju, James Bornholt and Greg Plaxton for their valuable insights and feedback.

Appendix A. Full Formalizations of Verification Obligations

We present here the full set of verification decompositions for abstraction correctness (`CorrAbst`, first introduced in Table 4). Each transition is

guarded by an $\texttt{Enabled}^X$ predicate, which determines whether the transition for variable X may proceed. For example, the transition to receive observational data ω_{in}^B is disabled for sensor m if there are no messages available in channel $c^{B,m}$. The decompositions and their guard predicates are detailed in Table A.10, with natural language descriptions alongside formalizations.

On notation. For the predicates shown in Table A.10, we let x_{ctr} denote a counter tracking the number of times variable x has been updated. These are ancillary state variables introduced for implementation convenience and property checking. We use primes (x') to denote next-state values of a variable x. The operator $t(\cdot)$ extracts the timestamp field of a message. For sensor-indexed variables, we also use the shorthand $\exists b_{\text{in}}^m$ to mean $\exists m \in \{1, \ldots, M\}$ such that b_{in}^m holds. We also use the convention introduced earlier (Eqn. 3) that any index k appearing in the context of message reception denotes a nondeterministically chosen position in the channel queue. Finally, when no valid control message is available, we use a fixed safe fallback as u_{default}.

Description	**Formalization**
ω^D Decompositions	
$\texttt{Enabled}^D$: None of the following is true: all sensor values are non-positive, the number of executed maneuvers has reached the maximum $e_{\max}$, or the estimated digital state d is non-positive:	$\texttt{Enabled}^D \coloneqq \neg\big[(\forall m \in \{1, \ldots, M\} : B^m \leq 0) \lor (\texttt{NumExecuted} \geq e_{\max}) \lor (d \leq 0)\big]$
P_{2A2a} is satisfied if all obligations (D1)–(D7) are satisfied	
(D1) Enabled guard is respected.	$\neg\texttt{Enabled}^D \Rightarrow (d' = d) \land (d'_{\mathsf{ctr}} = d_{\mathsf{ctr}})$
(D2) Progress only when inputs are present.	$(\exists\, b_{\text{in}}^m \neq \varnothing) \land (u \neq \varnothing) \Rightarrow d'_{\mathsf{ctr}} = d_{\mathsf{ctr}} + 1$
(D3) If an observational input drops below a threshold value, digital state estimate strictly decreases:	$\exists b_{\text{in}}^m \leq \texttt{Threshold} \Rightarrow d' < d$
(D4) If the control is conservative and all observational inputs are above a threshold, digital state estimate does not decrease by more than one:	$(u = 2) \land (\forall b_{\text{in}}^m \geq \texttt{Threshold}) \Rightarrow d' = d \lor (d-1)$
(D5) If the control input is aggressive, digital state estimate diverges by a wider bounded range:	$(u = 3) \land (\forall b_{\text{in}}^m \geq \texttt{Threshold}) \Rightarrow d' = d \lor (d-3) \lor (d-2) \lor (d-1)$
(D6) New digital state estimate is well formed:	$(d' \in \mathbb{Z}) \land (d'_{\text{ctr}} = d_{\text{ctr}} + 1)$

(D7) Digital state history is not overwritten by new computations:	$\forall d_n : \ d'_n = d_n$
$\omega^{B_{in}}$ Decompositions	
$\texttt{Enabled}^U$: If digital state transition ω^D is enabled.	$\texttt{Enabled}^D \Rrightarrow \texttt{Enabled}^{B_{\text{in}}}$
P_{3C2a} is satisfied if all obligations (Bin1)–(Bin3) are satisfied	
(Bin1) Enabled guard is respected.	$\neg\texttt{Enabled}^{B_{\text{in}}} \Rightarrow b_{\text{in}}^{m'} = b_{\text{in}}^{m}$
(Bin2) Valid observational data is received.	$t(c^{B,m}[k]) > b_{\text{in}}^{m} \Rightarrow (b_{\text{in}}^{m'} = c^{B,m}[k]) \wedge (c^{B,m'} = c^{B,m} \setminus c^{B,m}[k])$
(Bin3) Invalid observational data does not overwrite.	$t(c^{B,m}[k]) \leq b_{\text{in}}^{m} \Rightarrow (b_{\text{in}}^{m'} = b_{\text{in}}^{m}) \wedge (c^{B,m'} = c^{B,m} \setminus c^{B,m}[k])$
ω^U Decompositions	
$\texttt{Enabled}^U$: If digital state transition ω^D is enabled.	$\texttt{Enabled}^D \Rrightarrow \texttt{Enabled}^U$
P_{2C2a} is satisfied if all obligations (U1)–(U5) are satisfied	
(U1) Enabled guard is respected.	$\neg\texttt{Enabled}^U \Rightarrow (u' = u) \wedge (u'_{\text{ctr}} = u_{\text{ctr}})$
(U2) New control record is well formed.	$u'_{\text{ctr}} = u_{\text{ctr}} + 1 \wedge ((u' = 2) \vee (u' = 3))$
(U3) Guarded choice if digital state estimate is above threshold	$(d > \mathsf{Threshold}) \Rightarrow u' \in \{2, 3\}$
(U4) Guarded choice if digital state estimate is below threshold	$(d \leq \mathsf{Threshold}) \Rightarrow u' = 2$
(U5) Computed control is emitted.	$u' = c^{U'}[\texttt{len}(c^U) - 1]$
ω^S Decompositions	
$\texttt{Enabled}^S$: If structural health is positive and UAV has not reached the maximum number of maneuvers.	$\texttt{Enabled}^S \coloneqq (s > 0) \wedge (\texttt{NumExecuted} < e_{\max})$
P_{3A2a} is satisfied if all obligations (S1)–(S2) are satisfied	
(S1) Enabled guard is respected.	$\neg\texttt{Enabled}^S \Rightarrow s' = s$
(S2) Structural integrity is monotonically non-increasing.	$s' \leq s$
ω^{B^m} Decompositions	
$\texttt{Enabled}^{B^m}$: If physical state transition ω^S is enabled.	$\texttt{Enabled}^S \Rightarrow \texttt{Enabled}^{B^m}$

P_{3C2a} is satisfied if all obligations (B1)–(B2) are satisfied	
(B1) Enabled guard is respected.	$\neg \texttt{Enabled}^B \Rightarrow b_{\text{in}}^{m'} = b_{\text{in}}^{m}$
(B2) Sensor data generation and emission are atomic.	$b_{\text{in}}^{m'} = c^{B,m'}[\texttt{len}(c^{B,m}) - 1]$
$\omega^{U_{in}}$ Decompositions	
$\texttt{Enabled}^{U_{\text{in}}}$: If physical state transition ω^S is enabled.	$\texttt{Enabled}^S \Rightarrow \texttt{Enabled}^{U_{\text{in}}}$
P_{3B2a} is satisfied if all obligations (Uin1)–(Uin4) are satisfied	
(Uin1) Enabled guard is respected.	$\neg \texttt{Enabled}^{U_{\text{in}}} \Rightarrow u'_{\text{in}} = u_{\text{in}}$
(Uin2) Default maneuver when no dynamic control is available.	$c^U[k] = \varnothing \Rightarrow u'_{\text{in}} = u_{\text{default}}$
(Uin3) Default maneuver when outdated control is received.	$t(c^U[k]) < t(u_{\text{in}}) \Rightarrow u'_{\text{in}} = u_{\text{default}}$
(Uin4) Valid control received is executed.	$t(c^U[k]) > t(u_{\text{in}}) \Rightarrow u'_{\text{in}} = c^U[k]$

Table A.10: Verification obligations: natural language description (left) and formalization (right).

References

[1] National Academies of Sciences, Engineering, and Medicine, Foundational Research Gaps and Future Directions for Digital Twins, The National Academies Press, Washington, DC, 2024. doi:10.17226/26894.

[2] Z. Manna, A. Pnueli, The temporal logic of reactive and concurrent systems: specifications, Springer Science & Business Media, 1992, Ch. 4.

[3] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, 2008.

[4] E. M. C. Jr., O. Grumberg, D. Peleg, Model Checking, MIT Press, 1999.

[5] R. Alur, Principles of Cyber-Physical Systems, MIT Press, 2015.

[6] S. Mitra, Verifying Cyber-Physical Systems: A Path to Safe Autonomy, The MIT Press, 2021.

[7] G. Bakirtzis, U. Topcu, Algebraic Systems: Compositional verification for autonomous system design, in: 2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS), 2022, pp. 308–309. doi:10.1109/ICCPS54341.2022.00045.

[8] T. Wongpiromsarn, M. Ghasemi, M. Cubuktepe, G. Bakirtzis, S. Carr, M. O. Karabag, C. Neary, P. Gohari, U. Topcu, Formal methods for autonomous systems, Foundations and Trends in Systems and Control 10 (3-4) (2023) 180–407. doi:10.1561/2600000029.

[9] T. Wright, C. Gomes, J. Woodcock, Formally verified self-adaptation of an incubator digital twin, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Practice, Springer Nature Switzerland, Cham, 2022, pp. 89–109.

[10] S. Bateni, M. Lohstroh, H. S. Wong, H. Kim, S. Lin, C. Menard, E. A. Lee, Risk and mitigation of nondeterminism in distributed cyber-physical systems, in: Proceedings of the 21st ACM-IEEE International Conference on Formal Methods and Models for System Design, 2023, pp. 1–11. doi:10.1145/3610579.3613219.

[11] R. Davis, L. Cucu-Grosjean, A survey of probabilistic timing analysis techniques for real-time systems, Leibniz Transactions on Embedded

Systems (LITES) 6 (2019) 03:1–03:60. doi:10.4230/LITES-v006-i001-a003.

[12] B. Ghosh, C. Hobbs, S. Xu, D. Smith, J. H. Anderson, P. S. Thiagarajan, B. Berg, P. S. Duggirala, S. Chakraborty, Statistical verification of autonomous system controllers under timing uncertainties, Real-Time Systems (2024). doi:10.1007/s11241-023-09417-x.

[13] M. Mehrabian, M. Khayatian, A. Shrivastava, P. Derler, H. Andrade, A run-time verification method with consideration of uncertainties for cyber–physical systems, Microprocessors and Microsystems 101 (2023) 104890.

[14] E. A. Lee, S. A. Seshia, Introduction to Embedded Systems, Second Edition: A Cyber-Physical Systems Approach, MIT Press, 2017.

[15] S. Merz, The specification language TLA+, in: D. Bjørner, M. C. Henson (Eds.), Logics of Specification Languages, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 401–451.

[16] H. Howard, F. Alder, E. Ashton, A. Chamayou, S. Clebsch, M. Costa, A. Delignat-Lavaud, C. Fournet, A. Jeffery, M. Kerner, F. Kounelis, M. A. Kuppe, J. Maffre, M. Russinovich, C. M. Wintersteiger, Confidential consortium framework: Secure multiparty applications with confidentiality, integrity, and high availability (2023). arXiv:2310.11559.

[17] I. Konnov, M. Kuppe, S. Merz, Specification and verification with the TLA+ trifecta: TLC, Apalache, and TLAPS, in: Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles, Springer International Publishing, 2022, pp. 88–105.

[18] P. Regnier, G. Lima, A. Andrade, A TLA+ formal specification and verification of a new real-time communication protocol, Electronic Notes in Theoretical Computer Science 240 (2009) 221–238.

[19] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, M. Deardeuff, How Amazon Web Services uses formal methods, Communications of the ACM (2015).

[20] M. Kapteyn, J. Pretorius, K. Willcox, A probabilistic graphical model foundation for enabling predictive digital twins at scale, Nature Computational Science 1 (2021) 337–347.

[21] M. Tezzele, S. Carr, U. Topcu, K. E. Willcox, Adaptive planning for risk-aware predictive digital twins, in: I. V. Gosea, K. Cherifi (Eds.), SEMA SIMAI Springer Series, 2024.

[22] M. Torzoni, M. Tezzele, S. Mariani, A. Manzoni, K. E. Willcox, A digital twin framework for civil engineering structures, Computer Methods in Applied Mechanics and Engineering 418 (2024) 116584. doi:10.1016/j.cma.2023.116584.

[23] A. Chaudhuri, G. Pash, D. A. Hormuth, G. Lorenzo, M. Kapteyn, C. Wu, E. A. Lima, T. E. Yankeelov, K. Willcox, Predictive digital twin for optimizing patient-specific radiotherapy regimens under uncertainty in high-grade gliomas, Frontiers in Artificial Intelligence 6 (2023) 1222612.

[24] A. T. Ihler, J. W. Fisher, III, A. S. Willsky, Loopy belief propagation: Convergence and effects of message errors, Journal of Machine Learning Research 6 (2005) 905–936.

[25] L. Lamport, The temporal logic of actions, ACM Trans. Program. Lang. Syst. 16 (3) (1994) 872–923. doi:10.1145/177492.177726.

[26] L. Lamport, J. Matthews, M. Tuttle, Y. Yu, Specifying and verifying systems with TLA+, in: Proceedings of the 10th ACM SIGOPS European Workshop, 2002, pp. 45–48. doi:10.1145/1133373.1133382.

[27] Z. Manna, A. Pnueli, Temporal verification of reactive systems : safety, Springer, New York, 1995.

[28] J. F. Monin, Understanding Formal Methods, Springer, 2003.

[29] N. A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[30] A. Paz, Introduction to Probabilistic Automata, Academic Press, New York, 2018.

[31] P. Dupont, F. Denis, Y. Esposito, Links between probabilistic automata and hidden Markov models: probability distributions, learning models and induction algorithms, Pattern Recognition 38 (9) (2005) 1349–1371. doi:10.1016/j.patcog.2004.03.020.

[32] E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, R. Carrasco, Probabilistic finite-state machines - part I, IEEE Transactions on Pattern Analysis and Machine Intelligence 27 (7) (2005) 1013–1025. doi:10.1109/TPAMI.2005.147.

[33] E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, R. Carrasco, Probabilistic finite-state machines - part II, IEEE Transactions on Pattern Analysis and Machine Intelligence 27 (7) (2005) 1026–1039. doi:10.1109/TPAMI.2005.148.

[34] M. Krichen, A. Mihoub, M. Y. Alzahrani, W. Y. H. Adoni, T. Nahhal, Are formal methods applicable to machine learning and artificial intelligence?, in: 2022 2nd International Conference of Smart Systems and Emerging Technologies (SMARTTECH), 2022, pp. 48–53. doi:10.1109/SMARTTECH54121.2022.00025.

[35] K. Larsen, A. Legay, G. Nolte, M. Schlüter, M. Stoelinga, B. Steffen, Formal methods meet machine learning (F3ML), in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning, Springer Nature Switzerland, Cham, 2022, pp. 393–405.

[36] S. J. Salinger, M. G. Kapteyn, C. Kays, J. V. R. Pretorius, K. E. Willcox, A hardware testbed for dynamic data-driven aerospace digital twins, in: F. Darema, E. Blasch, S. Ravela, A. Aved (Eds.), Dynamic Data Driven Applications Systems, Springer International Publishing, Cham, 2020, pp. 37–45.

[37] M. Kapteyn, D. Knezevic, D. Huynh, M. Tran, K. Willcox, Data-driven physics-based digital twins via a library of component-based reduced-order models, International Journal for Numerical Methods in Engineering 123 (13) (2020) 2986–3003. doi:https://doi.org/10.1002/nme.6423.

[38] L. M. Prikler, F. Wotawa, Reevaluating the small-scope testing hypothesis of answer set programs, in: H. D. Menéndez, G. Bello-Orgaz,

P. Barnard, J. R. Bautista, A. Farahi, S. Dash, D. Han, S. Fortz, V. Rodriguez-Fernandez (Eds.), Testing Software and Systems, Springer Nature Switzerland, Cham, 2025, pp. 79–92.

[39] P. Chrszon, P. Maurer, G. Saleip, S. Müller, P. M. Fischer, A. Gerndt, M. Felderer, Model checking of spacecraft operational designs: a scalability analysis, Software and Systems Modeling (2025). doi:10.1007/s10270-025-01281-6.

[40] H. Yu, J. A. Evans, L. R. Varshney, Information lattice learning, Journal of Artificial Intelligence Research 77 (2023) 971–1019. doi:https://doi.org/10.1613/jair.1.14277.

[41] T. Alhazmi, F. Azzedin, J. Hassine, M. Hammoudeh, Formal specification and executable analysis of digital twin systems using Maude rewriting logic, Future Generation Computer Systems 176 (2026) 108148. doi:10.1016/j.future.2025.108148.

[42] A. Ferrari, P. Spoletini, Formal requirements engineering and large language models: a two-way roadmap, Information and Software Technology 181 (2025) 107697. doi:10.1016/j.infsof.2025.107697.

[43] Y. H. Lu, X. Y. Zhu, W. Zhang, R. Yan, Formalizing requirements into Dafny specifications with LLMs, in: Formal Methods and Software Engineering: 26th International Conference on Formal Engineering Methods, ICFEM 2025, Hangzhou, China, Proceedings, 2025, p. 97–116. doi:10.1007/978-981-95-4213-0_6.