Property-Based Testing of Wafer Handling Robots via Model-Based Planning and Fuzzing

Ruiyang Xu
East China Normal University
Shanghai Key Laboratory of Trustworthy Computing
Shanghai, China
xury2002@stu.ecnu.edu.cn

Jingjing Liang*
East China Normal University
Shanghai Key Laboratory of Trustworthy Computing
Shanghai, China
jjliang@sei.ecnu.edu.cn

Guoyue Zheng
East China Normal University
Shanghai Key Laboratory of Trustworthy Computing
Shanghai, China
51285902100@stu.ecnu.edu.cn

Geguang Pu
East China Normal University
Shanghai Key Laboratory of Trustworthy Computing
Shanghai, China
ggpu@sei.ecnu.edu.cn

Ting Su*
East China Normal University
Shanghai Key Laboratory of Trustworthy Computing
Shanghai, China
tsu@sei.ecnu.edu.cn

Abstract

Wafer handling robots (WHRs) in semiconductor manufacturing require ultra-precise control software, where bugs can cause severe mechanical damage and costly downtime, making reliability validation critical. Yet no existing work addresses WHR validation.

In this experience paper, we introduce a property-based testing approach for WHRs. Our key observation is that the system requirements of WHRs could be used as the properties for bug detection. However, applying property-based testing to WHR systems is challenging due to the input generation challenge. Complex action dependencies make it difficult to generate valid test inputs, while enormous state spaces hinder achieving adequate test coverage. To address this, we propose model-based planning and fuzzing. Our insight is that the WHR can be represented as a state-based model. Based on this model, we systematically generate valid test inputs through backward planning, then explore diverse scenarios through parameter and environment mutation. We implemented our approach as the ArmGo tool and evaluated it on a production WHR deployed in industrial lithography equipment. ArmGo discovered 21 previously unknown bugs (all confirmed, 15 fixed), including axis limit violations, entanglement issues, and missing interlock checks. ArmGo has been deployed in the continuous integration (CI) pipeline of our industrial partner and is actively used for validating WHRs. We also have distilled a number of practical lessons learned based on our investigation.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Semiconductor Manufacturing, Wafer Handling Robot, Testing

ACM Reference Format:
Ruiyang Xu, Jingjing Liang, Guoyue Zheng, Geguang Pu, and Ting Su. 2026. Property-Based Testing of Wafer Handling Robots via Model-Based Planning and Fuzzing. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834360

1 Introduction

Wafer handling robots (WHRs) are mission-critical robotic systems embedded in lithography equipment for semiconductor manufacturing [27, 33]. They are responsible for transferring wafers at high speed and accuracy in a vacuum environment [7, 10, 14, 17, 18]. A typical WHR (see Figure 1) operates with dual-arms simultaneously and performs different *actions* to coordinate with other hardware components (e.g., moving between a number of *stations*, *picking* a wafer from a carrier, and *placing* a wafer on a pre-aligner).

However, like all software, the control software of WHRs could have bugs. Even subtle faults could lead to critical damage to the wafers and other hardware components in the equipment. For example, software faults in WHRs may fail to detect a belt breakage, and thus interrupt the lithography production [34]. Due to the high cost of wafers and the lithography equipment, such accidents could incur significant economic losses [12, 37]. Therefore, validating the reliability of WHRs is critical.

In practice, to ensure the reliability of lithography production, WHRs are expected to respect a number of important *system requirements*. For example, *the carrier must be locked when the WHR enters and stays in the handoff zone*. The intention is that the carrier and the WHR should be stably aligned when the WHR picks or places wafers in this handoff zone and would not be affected by possible vibration during production. However, validating these system requirements is non-trivial due to the complexity of WHRs and the interactions with other hardware components. Our industrial partners inform us that the common strategy is manually designing test

*Ting Su and Jingjing Liang are corresponding authors.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834360

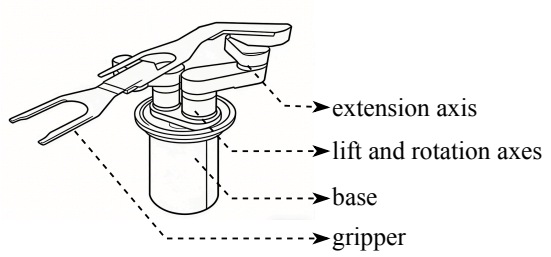

Fig. 1. A typical wafer handling robot with dual-arms in lithography equipment for semiconductor manufacturing.

cases to validate these requirements under different scenarios. To counter potential adverse conditions, they also add software sanity checks as well as hardware protection as the last resort. Despite these strategies, bugs still lurk and pose threats to production.

To this end, this paper introduces a novel *property-based testing* approach to effectively validating the reliability of WHRs. Property-based testing (PBT) [6] is a testing technique that validates the correctness of a piece of program against some specified properties. The core idea of PBT is to automatically generate a large number of test inputs, and validate whether these properties hold. If a test input violates a property, a bug is detected. Our key observation is that the system requirements of WHRs could be used as the properties for bug detection. Consider the aforementioned property "*the carrier must be locked when the WHR enters and stays in the handoff zone*", we can guide the WHR to reach the handoff zone through different sequences of actions, and then check whether the carrier is locked; if not, a property violation is found.

However, applying property-based testing to validate WHR systems is not straightforward. A key technical challenge lies in *how to effectively generate test inputs for WHRs. On one hand*, the actions of WHRs exhibit complex dependencies. Randomly generating valid test inputs is thus difficult. For example, picking wafers can only be performed after the WHR reaches the handoff zone, which itself requires a strict sequence of preceding steps. Any change to this sequence may violate the dependencies required to reach the handoff zone. These dependencies prevent naive or random permutations of actions, as violating the required execution order cannot lead to the desired system state. *On the other hand*, the state space of a WHR system is enormous, making exhaustive input generation infeasible. Completing a single task often requires complex permutations of actions, resulting in long execution sequences. Many of these actions incorporate continuous dynamics that can take infinitely many values within their ranges (e.g., lift-axis heights). Additionally, the same action can produce different outcomes under varying environments (e.g., carrier lock or unlock). The combination of different action sequences, continuous parameter spaces, and diverse environments leads to an explosion of system states, making adequate coverage extremely challenging.

To address the input generation challenge, we introduce *model-based planning and fuzzing*, a novel input generation approach for WHRs. The core idea is to first systematically generate valid test inputs through planning, then explore boundary conditions through fuzzing. Our insight is that the WHR can be represented as a state-based model. In this model, each state captures a snapshot of the system and WHR actions trigger state transitions. Specifically, each action is governed by guards, effects, and constraints according to WHR's specification. Guards specify when the action can be applied, effects describe the resulting state changes, and constraints define valid action parameter bounds. This model enables us to: (1) systematically identify action dependencies through guard-effect analysis, ensuring valid sequence generation; and (2) guide state space exploration through constraints, enabling effective fuzzing around feasible bounds.

Based on this model, our input generation approach operates in two stages. (1) *Model-based planning* generates valid inputs in a backward manner. Given a target action, the planner recursively identifies predecessor actions whose effects satisfy the required guards, continuing until reaching actions enabled by the initial state. This backward approach avoids exploring invalid paths that cannot reach the target, reducing the search space compared to forward exploration. (2) *Model-based fuzzing* further explores the state space by building upon the valid inputs generated by planning. Specifically, we employ two mutation strategies: *Parameter Mutation*, which mutates action parameters around their constraint bounds (e.g., varying lift axis height) to explore the continuous parameter space; and *Environment Mutation*, which modifies environmental conditions during execution (e.g., unlocking the carrier) to evaluate system robustness under various conditions.

By combining backward planning with fuzzing, our approach effectively generates action sequences as test inputs. We formalize system requirements into five categories of validated system properties. These inputs are then executed step by step, with properties checked at each state. Any violation is reported as a bug.

We implemented our approach in a tool named ArmGo and evaluated it on a production WHR in a high-fidelity industrial lithography equipment simulation platform. Although this production WHR has been extensively tested in practice, ArmGo has successfully uncovered 21 previously unknown bugs across four distinct property types (no crash/hang: 1, entanglement: 6, axis limit: 11, and interaction: 3). All 21 have been confirmed by developers to date, and 15 have already been fixed. Further evaluation shows that manual testing (MT) detected none, both undirected random testing (URT) and feedback-directed random testing (FRT) [29] found only 1. According to our analysis, these bugs all lies in abnormal or corner scenarios that cannot be found by existing techniques, but were successfully exposed by ArmGo. If triggered in real production, such cases could result in wafer scrap or equipment downtime, potentially leading to considerable operational risks and costs.

To evaluate the test adequacy, we devises three coverages: station coverage (L1), operation coverage (L2) and parameter coverage (L3). Specifically, L1 ensures each station is visited at least once during execution, L2 requires executing all operations at each station to more thoroughly exercise functionalities, and L3 tests all operation parameters with both valid and invalid inputs. Based on these coverage metrics, ArmGo reaches 100% on L1 coverage (vs. 100% MT, 41.03% URT, 61.53% FRT), while boosting L2 operation coverage from 24.04% (MT) / 23.73% (URT) / 33.65% (FRT) to 41.47%—an improvement of over 72% relative to manual testing. Meanwhile, ArmGo also achieves 100% on L3 parameter coverage (vs. 68% MT,

88% URT, 88% FRT). To date, our ArmGo has been deployed in the continuous integration (CI) pipeline of our industrial partner and is actively used for validating WHRs.

In summary, we make the following contributions:

- We introduce a novel *property-based testing* approach to validating WHR systems. Our approach combines the idea of property-based testing with modeling WHR to achieve effective testing.
- We propose *model-based planning and fuzzing* to effectively generate test inputs and efficiently explore the state space.
- We implement ArmGo and evaluate it on a production WHR deployed with industrial lithography equipment. ArmGo has uncovered 21 previously unknown bugs (all confirmed, 15 already fixed). We also have distilled a number of practical lessons learned based on our investigation.

2 Background and Motivating Example

2.1 Wafer Handling Robots

A typical wafer handling robot (WHR) architecture is shown in Figure 1. It has a dual-arm, where two arms are mounted on a single base. Each arm is equipped with its own independent lifting and rotation axes, extension axis, and gripper (i.e., the end-effector).

The WHR operates by coordinating these components to execute a sequence of actions, including inter-station movement, axis adjustment, and wafer pick-and-place. For example, the lift and rotation axes coordinate with the base to move precisely between stations. Once moved to a handoff station, the extension axis drives radial movement, allowing the gripper to pick or place wafer.

Figure 2 shows the workflow of picking a wafer from the carrier, as indicated by the arrows; please ignore the parts in the dashed box (these refer to a bug that we discuss later). Specifically, the WHR moves step by step from `HOME` to `HANDOFF`, then picks a wafer from the carrier. After successfully pickup, it navigates backs to `HOME` via `HANDOFF-HIGH` and `RETRACT-HIGH`.

2.2 Motivating Example

Figure 2 presents a real-world bug uncovered in an industrial WHR that violates the axis limit property. This property requires that axis movements (e.g., extension, rotation, lift) at any station strictly respect the physical bounds of that specific station. As shown in the figure, the robot travels from the `HOME` station, picks a wafer at the `HANDOFF` station, and then moves to the `HANDOFF-HIGH` station. The property violation occurs when the robot is at `HANDOFF-HIGH` holding a wafer: the actual implemented lifting limit (blue line) exceeds the documented safety limit (green line) by 0.05 mm.

Challenge. Triggering this property violation is difficult because it requires satisfying strict action dependencies and exploring specific parameter boundaries. First, the WHR must reach the `HANDOFF-HIGH` station holding wafer, by executing a lengthy, complex sequence of actions. Specifically, the sequence is made by the following actions: `HOME` → `RETRACT-LOW` → `HANDOFF-LOW` → `HANDOFF` → `Pick` → `HANDOFF-HIGH`. Second, simply reaching this target station is not enough. The test must also explore the lift axis parameter exactly at the extreme boundary to uncover the 0.05 mm gap. The combination of constructing the action sequences and exploring continuous parameter spaces forms a huge testing space, making such property violations extremely hard to find using routine testing.

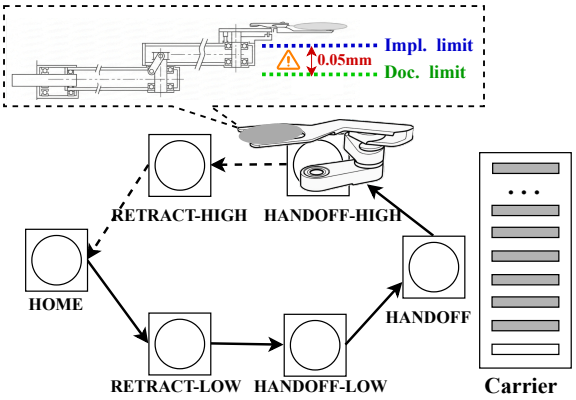

Fig. 2. A real-world bug in an industrial WHR: Dashed box shows 0.05 mm gap between implemented (blue) and documented (green) axis lifting limits.

3 Approach

3.1 Definition

Definition 3.1 (Action)**.** A wafer handling action a performed by the WHR can be defined as a triple $a = (r, t, p)$, where $a.r \in \{arm_a, arm_b\}$ denotes the robot arm that executes the action, $a.t$ denotes the action type (e.g., move, lift), and $a.p$ denotes the parameters associated with the action type (e.g., target station for move, height for lift).

Example. A move action executed by arm_a to the `HANDOFF-HIGH` station can be represented as $a = (arm_a, move, \{\texttt{HANDOFF-HIGH}\})$, while a lift action by arm_a raising the lift-axis by 5mm is denoted as $a = (arm_a, lift, \{5\text{mm}\})$.

Definition 3.2 (WHR Model)**.** We formalize the WHR system as a state-based model [32], denoted as $\mathcal{M} = \langle S, A, \delta, s_0 \rangle$, where:

- S is the state space. Each state $s \in S$ represents a snapshot of the system, defined by a set of state variables, such as arm status (i.e., $status$), wafer location (i.e., $location$), and wafer holding indicator (i.e., $hold\_wafer$).
- A is the set of actions. Each action $a \in A$ triggers a state transition in the system.
- $\delta = \{(s, a, s') \in S \times A \times S \mid C(a) \wedge G(a)\}$ is the transition function, such that $(s, a, s') \in \delta$ if and only if the action parameter $a.p$ satisfies the constraint $C(a)$, and s satisfies the guard condition $G(a)$. The state change after executing a is represented by effect $E(a)$, Here, $C(a)$ defines the valid parameter ranges for action a, and $G(a)$ is a predicate over states.
- $s_0 \in S$ is initial state.

Example. Consider the scenario where the robot moves from the `HANDOFF` to the `HANDOFF-HIGH` station. Suppose the robot is currently at the `HANDOFF` station with the state s (where s.status = INIT, s.location = HANDOFF, and s.hold_wafer = true) [1]. Given an action $a = (arm_a, move, \{\texttt{HANDOFF-HIGH}\})$, its parameter constraints

[1] For brevity, only a subset of the current state s is shown.

$C(a)$, guard $G(a)$, and effect $E(a)$ are defined as follows:

$$\begin{aligned} C(a):&\quad a.p.\text{location} \in \{\texttt{HANDOFF-LOW}, \texttt{HANDOFF-HIGH}\} \\ G(a):&\quad s.\text{status} = \text{INIT} \quad \wedge s.\text{location} = \texttt{HANDOFF} \\ &\quad s.\text{hold\_wafer} = \text{true} \quad \wedge s.\text{carrier\_locked} = \text{true} \\ E(a):&\quad s'.\text{location} = \texttt{HANDOFF-HIGH} \end{aligned}$$

Here, we only show the constraint on the station parameter for the move action. That is, when the robot is at the HANDOFF station, the target station must be either HANDOFF-LOW or HANDOFF-HIGH. In this case, since HANDOFF-HIGH satisfies the constraint $C(a)$ and the current state s satisfies the guard $G(a)$, the action can be executed. Upon execution, the system transitions to the next state s', with the state changes described by $E(a)$.

Definition 3.3 (Test Inputs for WHR)**.** A test input for the WHR system is a sequence of actions. A test sequence τ is denoted as $\tau = [a_1, a_2, \ldots, a_n]$, where $a_j \in A$ is an action. When τ is executed on the WHR system from an initial state s_0, it produces a sequence of state transitions:

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots \xrightarrow{a_j} s_j \cdots \xrightarrow{a_n} s_n$$

Example. Consider the scenario where the robot moves from the HOME station to the HANDOFF station. A test input is denoted as $\tau = [(arm_a, move, \{\texttt{RETRACT-LOW}\}), (arm_a, move, \{\texttt{HANDOFF-LOW}\}), (arm_a, move, \{\texttt{HANDOFF}\})]$. Starting from the initial state s_0 with s_0.location = HOME, the execution proceeds as follows[2]:

$$s_0 \xrightarrow{\texttt{RETRACT-LOW}} s_1 \xrightarrow{\texttt{HANDOFF-LOW}} s_2 \xrightarrow{\texttt{HANDOFF}} s_3$$

Definition 3.4 (System Properties of WHRs)**.** A system property is a triple $\phi = \langle Pre, a, Post \rangle$, where Pre and $Post$ are predicates over states, and $a \in A$ is an action. The property ϕ specifies:

$$s \models Pre \wedge (s, a, s') \in \delta \Rightarrow s' \models Post$$

That is, if the precondition holds in state s and the transition (s, a, s') is valid, then the postcondition must hold in state s'. Note that each property is bound to a single action, as we focus on single-step properties rather than temporal properties over action sequences.

Example. Consider the property "*the carrier must be locked when the WHR enters and stays in the handoff zone*". When executing action $a = (arm_a, move, \{\texttt{HANDOFF-HIGH}\})$, the property $\phi = \langle Pre, a, Post \rangle$ is defined as:

$$s \models \text{true} \wedge (s, a, s') \in \delta \Rightarrow s' \models s'.\text{carrier\_locked} = \text{true}$$

Problem Definition. Our problem is to generate a set of test inputs to validate the correctness of the WHR. Given a test input, we execute it on the WHR system and check whether all states during execution satisfy the system properties. Any such violation indicates a bug.

3.2 High-level Approach

We present a property-based testing approach to systematically test the WHR system. The key challenge in applying this approach is how to generate effective test inputs, as (1) actions have dependencies on each other, and (2) the input space is extremely large due to the numerous possible parameter and various environments for different action.

[2]For brevity, we use action parameter to represent the action.

To address this challenge, we propose a model-based planning and fuzzing approach that can automatically generate test inputs. First, *model-based planning* generates valid action sequences by leveraging dependencies encoded in the model's transition function, where an action a' is applicable after a only if the resulting system state s (after applying a) satisfies a''s guard conditions. Second, *model-based fuzzing* takes these valid sequences as input and explores diverse execution scenarios by mutating action parameters and environment-related variables. When extending each mutated sequence, fuzzing relies on the same dependencies (i.e., system states and guards) to identify applicable next actions, then mutates them according to their constraints and guards.

Definition 3.5 (Model-based Planning)**.** Given a WHR model $\mathcal{M} = \langle S, A, \delta, s_0 \rangle$, for each action $a \in A$, we aim to generate a valid test input that represents a path from the initial state s_0 to a state where action a is applicable. Formally, a test input for action a_t is a sequence $\tau_a = [a_1, a_2, \ldots, a_n, a_t]$ such that there exists an execution trace:

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n \xrightarrow{a_t} s_t$$

where $s_n \models G(a_t)$ (i.e., action a_t is applicable in state s_n). The output is the set of all such test inputs: $\mathcal{T}_p = \{\tau_a \mid a \in A\}$.

Definition 3.6 (Model-based Fuzzing)**.** Given a WHR model $\mathcal{M}$ and the test suite $\mathcal{T}_p = \{\tau_a \mid a \in A\}$ from planning, for each $\tau_a = [a_1, \ldots, a_n, a_t] \in \mathcal{T}_p$, we execute τ_a to reach state s_t, identify the set of applicable actions $A(s_t) = \{a' \in A \mid s_t \models G(a')\}$, and generate fuzzed variants by mutating action parameters and environmental-related state variables in $G(a')$ for each $a' \in A(s_t)$. The output is the extended test suite:

$$\mathcal{T}_f = \bigcup_{\tau_a \in \mathcal{T}_p} \bigcup_{a' \in A(s_t)} \bigcup_{a'' \in \text{fuzz}(a')} \{\tau_a \cdot a''\}$$

where a'' is a fuzzed variant of action a' with mutated parameters and environment variables, generated by $\text{fuzz}(a')$, and $\tau_a \cdot a''$ extends τ_a by appending a''.

Definition 3.7 (Property Checking)**.** Given the test suite $\mathcal{T} = \mathcal{T}_p \cup \mathcal{T}_f$ and a property set $\mathcal{P}$, for each test sequence $\tau = [a_1, \ldots, a_n] \in \mathcal{T}$ that produces a state sequence $(s_0, s_1, \ldots, s_n)$, we check whether any property $\phi = \langle Pre, a, Post \rangle \in \mathcal{P}$ is violated. Specifically, after executing action a_j and transitioning to state s_j, a property violation occurs if:

$$s_{j-1} \models Pre \wedge (s_{j-1}, a_j, s_j) \in \delta \wedge s_j \not\models Post$$

Any violation during the execution of τ indicates a bug.

4 Implementation

We implement PBT for WHR in ArmGo. Based on the WHR model, ArmGo generates test inputs in two stages: first, model-based planning produces valid action sequences (Section 4.1); second, model-based fuzzing explores diverse parameters and environments (Section 4.2). We then present the validated properties and describe property violations detection during testing (Section 4.3). Finally, we present tool implementation details (Section 4.4).

Algorithm 1 Model-based Planning

```
1:  function FindSequence(G, s, M)
2:      seq ← ∅
3:      G_met ← {g ∈ G | s ⊨ g}                ▷ Track already met conditions
4:      for each g ∈ G \ G_met do
5:          if s ⊨ g then continue              ▷ Skip if met incidentally by prior acts
6:          act ← FindProvider(g, G_met, s, M)
7:          if act = NULL then return Failure
8:          G_act ← act.G                       ▷ Sub-goals: guards of provider
9:          seq_pre, s' ← FindSequence(G_act, s, M)   ▷ Note: s is updated
10:         seq ← seq ⌢ seq_pre ⌢ ⟨act⟩          ▷ Topological ordering
11:         s ← ApplyEffects(s', act.E)         ▷ Update global tracked state s
12:         G_met ← G_met ∪ {g}
13:     return (seq, s)
14: function FindProvider(g, G_met, s, M)
15:     C_min ← ∞, act_best ← NULL
16:     for each act ∈ M where act.E ⊨ g do
17:         cost_threat ← 0, cost_app ← 0
18:         for each g' ∈ G_met do              ▷ Evaluate Threat Penalty
19:             if act.E clobbers g' then
20:                 cost_threat ← cost_threat + Cost(g')
21:         for each g' ∈ act.G do              ▷ Evaluate Applicability Cost
22:             if s ⊭ g' then cost_app ← cost_app + Cost(g')
23:         cost_total ← cost_threat + cost_app
24:         if cost_total < C_min then
25:             C_min ← cost_total, act_best ← act
26:     return act_best
```

4.1 Model-based Planning

The goal of planning is to generate a valid action sequence for each action in the WHR model. Given target action a with guard $G(a)$, initial state s_0, and model $\mathcal{M}$, Algorithm 1 outputs a sequence satisfying $G(a)$ from s_0 via backward search. The main function FindSequence maintains two variables: G (the current set of guards to satisfy, initialized to $G(a)$) and s (the current state, initialized to s_0). For each unmet guard $g \in G(a)$ (Line 4), FindProvider finds a provider action whose effects satisfy g (Line 6). If the provider has its own guards, the algorithm recursively finds sub-providers (Lines 8-9) until all guards are satisfied. Specifically, this backward search considers the following practical issues:

(1) **Redundant actions.** In backward searching, actions may share the same prerequisite guards. A naive search would repeatedly resolve these shared guards, generating redundant subsequences. To solve this, we introduce *forward state tracking*. As shown in Algorithm 1, we continuously track and update the simulated state s. After resolving a provider action, we apply its effects to update s (Line 11). Before resolving a new guard g, we first check if $s \models g$ (Line 5), which skips any guard already satisfied by prior actions.

(2) **Conflicting effects.** A chosen provider action may satisfy a target guard, but its effects can violate (clobber) other already-met goals in the tracked state. To mitigate this, FindProvider implements a cost-based heuristic. Each candidate action is evaluated by a combined cost: (1) *Threat Penalty* ($cost_{threat}$), penalizing actions that break already met conditions (G_{met}) (Lines 18-20), and (2) *Applicability Cost* ($cost_{app}$), penalizing actions with many unmet guards of their own (Lines 21-22). By choosing the minimal-cost candidate, the planner prioritizes non-destructive actions close to the current state (Lines 23-25).

(3) **Ordering.** Since actions are found backward, they must be properly ordered for forward execution. Our recursive algorithm performs a *topological sort* on the dependency graph. Specifically, actions with no unmet guards end recursion first and appear at the start of the sequence. For each provider action act, its required sub-sequence seq_{pre} is placed directly before act itself (Line 10). This ensures a valid chain where all dependencies are met before execution.

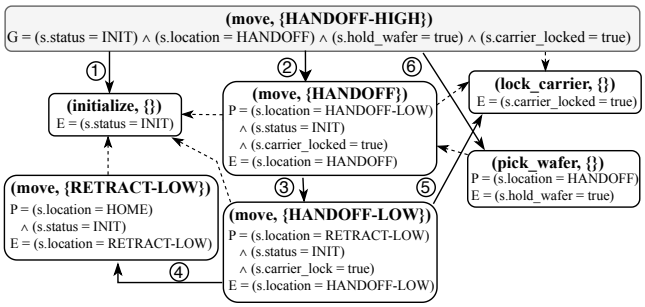

Fig. 3. Demonstration of the planning algorithm using the motivating example. A solid line means one action's guard G is resolved by another action's effect E; a dashed line means the avoided redundant actions.

Illustrative Example. We illustrate this process using a backward chaining tree shown in Figure 3 (derived from the motivating example), where each box represents an action[3], solid arrows (①-⑥) indicate the goal resolution order and dashed arrows represent incidental satisfaction. Suppose the target action requires reaching `HANDOFF-HIGH` with a wafer held (grep box). The planner sequentially resolves the guards of the target action (*move*, `HANDOFF-HIGH`).

First the planner schedules *initialize* action (①) to resolve s.status = INIT. Next, to satisfy s.location = `HANDOFF`, it backward chains through three *move* actions (②–④) until aligning with the initial state with respect to $s.location$. Crucially, thanks to state tracking, the shared s.status = INIT requirement for these intermediate movements is incidentally met by ① (dashed arrows). Meanwhile, it resolves s.carrier_locked = true via *lock_carrier* (⑤). Next, *pick_wafer* (⑥) is scheduled to satisfy s.hold_wafer = true via (⑥). Finally, through a topological sort, the planner produces a test input: $\tau = [(initialize, \{\}), (move, \{\texttt{RETRACT-LOW}\}), (lock\_carrier, \{\}), (move, \{\texttt{HANDOFF-LOW}\}), (move, \{\texttt{HANDOFF}\}), (pick\_wafer, \{\}), (move, \{\texttt{HANDOFF-HIGH}\})]$.

4.2 Model-based Fuzzing

Fuzzing is applied to the state reached after executing each test input generated by planning. Specifically, given a planned test input $\tau = [a_1, a_2, \ldots, a_n, a_t]$, ArmGo drives the WHR to execute it to a target state s_t, then starting the fuzzing stage. ArmGo first identifies applicable actions in s_t by checking which actions' guards are satisfied in the model, and performs mutations.

[3]All actions are performed by the same arm. we omit the arm identifier for brevity.

Table 1. Formalization of system properties validated in our approach.

Property Name	System Requirement Description	System Property
No Crash/Hang	For any action, the WHR must run without crashing / hanging	$\forall(s,a,s') \in \delta \Rightarrow s'.\text{status} \notin \{\text{CRASH}, \text{HANG}\}$
Dual-arm Entanglement Protection	To avoid collisions, the two robot arms must not simultaneously occupy any pair of physically entangled stations	Let C_{etg} be the entangled pair set and loc an arm location. For the other arm, $\forall(s,a,s') \in \delta \Rightarrow (s.\text{location}, loc) \notin C_{etg}$
Per-Station Axis Limits	Axis movements (i.e., extension, rotation, lift) at any station strictly respect the physical bounds of that specific station.	$\forall(s,a,s') \in \delta, \forall x \in \{\text{ext}, \text{rot}, \text{lift}\}$: $s'.x \in [\text{Min}(s'.\text{location}, x), \text{Max}(s'.\text{location}, x)]$
Interaction Constraints	Environmental conditions must be satisfied whenever the WHR enters or stays in a module's handoff *zone*.	$\forall(s,a,s') \in \delta, \forall i \in \{\text{carrier, pre-aligner, storage-unit}\}$, $s'.\text{location} \in zone_i, \Rightarrow s'.\text{location} \models \text{ConstraintOf}(zone_i)$
Pick/Place Consistency	A pick (place) action strictly requires an empty (occupied) gripper beforehand and results in an occupied (empty) gripper afterwards.	$\forall(s,a,s') \in \delta$, $a.t = \text{pick} \Rightarrow (s.\text{hold\_wafer} = \text{false} \wedge s'.\text{hold\_wafer} = \text{true})$ $a.t = \text{place} \Rightarrow (s.\text{hold\_wafer} = \text{true} \wedge s'.\text{hold\_wafer} = \text{false})$

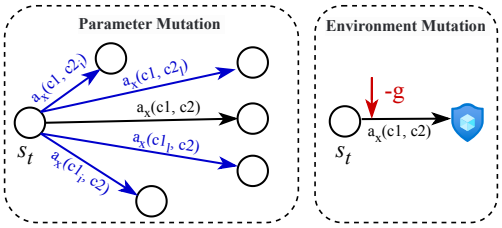

Fig. 4. The two mutation strategies for an action. The blue lines indicate the actions with mutated parameters.

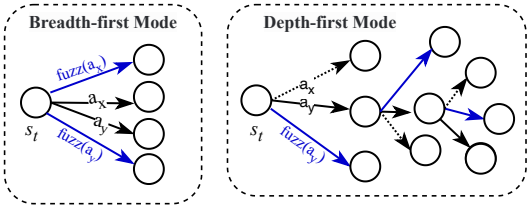

Fig. 5. The breadth-first (left) and depth-first (right) modes. Solid lines indicate selected actions, dashed lines indicate discarded actions, and blue lines indicate mutated variants.

Mutation Strategies. ArmGo applies two mutation strategies to generate fuzzing input for applicable actions:

- *Parameter Mutation.* As shown in Figure 4 (left), the fuzzer mutates parameters based on their constraints (e.g., the maximum lifting limit at `HANDOFF-HIGH`). It samples both legal values (e.g., $c1_l, c2_l$), especially boundary cases; and illegal values (e.g., $c1_i, c2_i$).
- *Environment Mutation.* As shown in Figure 4 (right), the fuzzer intentionally negates an action's *environmental* guards ($-g$) by injecting API calls that alter the environment. The blue guard indicates the system should safely reject the action due to the violated environmental properties.

Fuzzing Modes. Leveraging two mutation strategies, ArmGo operates in two complementary modes for systematic state exploration:

- *Breadth-first Mode.* For *all* applicable actions in $A(s_t)$, ArmGo applies mutation strategies to generate a set of test inputs for each action. To minimize execution overhead, we set the breadth exploration depth to one step. For example, in Figure 5 (left), if there are two applicable actions (i.e., a_x and a_y) at state s_t, ArmGo independently fuzzes all two. Each fuzzed variant is then appended to the original sequence τ to form a new test input.
- *Depth-first Mode.* ArmGo randomly selects one applicable action, applies mutation strategies to generate a set of test inputs, and executes this action (without mutations) to reach the next state. At each new state, ArmGo queries the model for applicable actions and repeats this process until reaching a predefined timeout. For example, in Figure 5 (right), the process works as follows: in the first step, action a_y is selected and mutated; it then executed (without mutation) to reach the next step. This process is repeated iteratively.

The two modes complement each other: breadth-first mode excels at exhaustive stress testing of individual states to avoid missing local bugs, while depth-first mode enables fast coverage growth and uncovers inter-state bugs that manifest only across multiple state transitions. Together, they provide both thorough local exploration and deep path coverage. Note that each test input is executed from the initial state, as parameter and environment mutations may irreversibly alter the physical or internal states of the WHR.

4.3 Property Checking

ArmGo interleaves test execution with property validation. As each test input is executed, the property checker monitors the system state s' after every action and verifies whether the current state satisfies the system properties. These properties are manually derived from system requirements, as summarized in Table 1. For example, *interaction constraints* requires specific environmental conditions to be met within each module's respective handoff zone: the carrier must be locked in the carrier handoff zone; the pre-aligner's lift stage must be down in the pre-aligner handoff zone; and the storage unit chuck must be empty in its handoff zone. This ensures the checker validates the state s' against the predicate $\text{ConstraintOf}(zone_i)$ for the specific module i being interacted.

Besides system properties, the checker also checks the effects of each actions. Specifically, for each executed action, the checker validates whether its execution complies with the defined effects.

4.4 Tool Implementation

We built ArmGo from scratch with three key components: a YAML model (~4k lines), a Python planner (~800 lines), and a C++ execution engine integrated with a property checker (~4k lines). Its multi-language design balances readability and efficiency. Below are details of constructing the model and deriving the properties.

Table 2. Overview of 21 unique bugs discovered by ArmGo.

ID	Status	Violated Properties	Description
1	Fixed	Axis Limits	Lowering limit deviation (no wafer) at RETRACT-LOW@Carrier
2	Fixed	Axis Limits	Lowering limit deviation (no wafer held) at HANDOFF-LOW@Carrier
3	Fixed	Axis Limits	Lifting limit deviation (wafer held) at RETRACT-HIGH@Carrier
4	Fixed	Axis Limits	Lifting limit deviation (wafer held) at HANDOFF-HIGH@Carrier
5	Fixed	Axis Limits	Radial extension limit deviation (no wafer held) at RETRACT-LOW@Pre-aligner
6	Fixed	Axis Limits	Radial extension limit deviation (no wafer held) at SOA@Pre-aligner
7	Fixed	Axis Limits	Radial retraction limit deviation (wafer held) at RETRACT-HIGH@Pre-aligner
8	Fixed	Axis Limits	Lifting limit deviation (no wafer held) at RETRACT-LOW@Pre-aligner
9	Fixed	Axis Limits	Lowering limit deviation (wafer held) at RETRACT-HIGH@Pre-aligner
10	Fixed	Axis Limits	Lifting limit deviation (no wafer held) at HANDOFF-LOW@SU-OUT
11	Fixed	Axis Limits	Lowering limit deviation (no wafer held) at HANDOFF-HIGH@SU-OUT
12	Fixed	No Crash/Hang	Missing input validation for the present parameter in sim_set_carrier_wafer.
13–15	Confirmed	Entanglement	Missing entanglement protection between ABOVE/AT/BELOW@Storage-OUT and RETRACT-LOW@Storage-TOP.
16–18	Confirmed	Entanglement	Missing entanglement protection between ABOVE/AT/BELOW@Storage-IN and RETRACT-LOW@Storage-TOP.
19	Fixed	Interaction	Lack of checking Pre-aligner stage down when moving from HANDOFF-LOW@Pre-aligner to SOA@Pre-aligner
20	Fixed	Interaction	Lack of checking Pre-aligner stage down for self-station movement from HANDOFF@Pre-aligner to itself
21	Fixed	Interaction	Lack of checking Storage-OUT chuck empty self-station movement from SU-HANDOFF-HIGH to itself

* A station (e.g., RETRACT-LOW@Carrier) denotes a unique location within the WHR's own coordinate frame for interaction with a specific external module.

Regarding the model construction, we obverse that the WHR operates via software APIs, and each modeled action maps directly to an API call with specific parameter values. This direct mapping allows us to construct the model manually based on the WHR specification, which clearly defines the applicable scenario, required parameters, execution guards, and state effects for every API call. Note that the manual model construction primarily requires understanding the hardware behaviors defined in the specification. Once understood, the actual modeling effort is minimal, as the modeling language (YAML) is easy to learn and requires no formal methods background. In practice, it took us about 20 man-hours.

Regarding property derivation, the properties are documented in the specification as strict physical constraints in natural language. These constraints are explicit and can be easily understood and translated into our DSL using standard conditional logic.

To execute test sequences, the engine communicates with the WHR controller using a vendor-provided SDK. During fuzzing, the engine continuously polls the robot's real-time telemetry data (e.g., axis positions, sensor flags). This data feeds directly into the property checker, enabling ArmGo to stop the robot and log the API trace whenever a property violation is detected.

5 Evaluation

We evaluate ArmGo by answering the following research questions:

RQ1 How effective is ArmGo at bug finding and coverage?

RQ2 Does ArmGo outperform the baselines (i.e., manual testing, undirected random testing, and feedback-directed random testing) in terms of bug finding and coverage?

RQ3 What are the contributions of the planning and two fuzzing modes (i.e., breadth-first and depth-first) in ArmGo?

5.1 Experiment Setup

Subject. We evaluated ArmGo on the latest version of a production WHR within industrial lithography equipment. The WHR interacts with a carrier (wafer loading), a pre-aligner (wafer orientation), and a storage-unit serving as a wafer cache (comprising in, out and top ports). Due to confidentiality, specific hardware models and infrastructures details are anonymized.

Infrastructure. ArmGo runs on Linux and remotely sends commands to the WHR control software, which deployed on RISC hardware with a real-time OS (RTOS). All experiments run in simulation mode. Note that core software layers (motion control, environment validation, API invocation) are shared between simulation and real modes, ensuring findings reflect real-world conditions.

Metrics. We evaluate ArmGo using the following four metrics:

1) Number of Detected Bugs. We count the number of distinct bugs based on developer's feedback and patch merges.

2) Station Coverage (L1). It measures whether each station is visited at least once, as bugs may only manifest at specific physical locations. Given stations $\mathcal{L}$ ($|\mathcal{L}| = 39$), test suite $\mathcal{T}$, and visited(t) (i.e., stations visited by test t), station coverage is defined as:

$$\mathrm{L1}(\mathcal{T}) = \frac{|\{l \in \mathcal{L} | \exists t \in \mathcal{T}, l \in \mathrm{visited}(t)\}|}{|\mathcal{L}|}$$

3) Operation Coverage (L2). It requires executing all operations at each station. While L1 covers stations, L2 exercises the WHR's functionalities within each station. Let $O(l)$ denote the operation set at station l, and executes(t, l, o) indicate that test t executes operation o at station l. In our evaluation, each station supports 8 operations, giving a total of $\sum_{l \in \mathcal{L}} |O(l)| = 312$ operations (39 stations × 8 operations). Operation coverage is:

$$\mathrm{L2}(\mathcal{T}) = \frac{\sum_{l \in \mathcal{L}} |\{o \in O(l) | \exists t \in \mathcal{T}, \mathrm{executes}(t,l,o)\}|}{\sum_{l \in \mathcal{L}} |O(l)|}$$

4) Parameter Coverage (L3). It requires exercising all parameters to ensure robustness against valid and invalid inputs. Following equivalence partitioning [26], we divide each API parameter into legal and illegal buckets. Let $\mathcal{B}$ denote all parameter buckets ($|\mathcal{B}| = 50$: 25 parameters × 2 types), and covered(t) be the buckets exercised by test t. A bucket is covered if the system correctly processes valid input or safely rejects invalid input. Parameter coverage is:

$$\mathrm{L3}(\mathcal{T}) = \frac{|\{b \in \mathcal{B} | \exists t \in \mathcal{T}, b \in \mathrm{covered}(t)\}|}{|\mathcal{B}|}$$

Note that we design these coverage criteria based on WHR behaviors rather than traditional code coverage (e.g., line or branch) for

two reasons. First, WHRs are cyber-physical systems; software execution paths alone cannot reflect their complex behaviors and states. Second, WHR relies on a distributed architecture with closed-source hardware controller, making instrumentation highly difficult.

Common Setup. In planning stage, ArmGo generates 136 unique offline test cases. Based on these cases, we then conducted bounded (breadth step = 1) fuzzing in the breadth-first mode. The entire evaluation completed in approximately 20 hours, and this duration was adopted as the fixed time budget for all subsequent experiments.

5.2 RQ1: Bug Finding and Coverage

Bug Count. Table 2 shows the bug finding results. It contains the bug ID, status (fixed or confirmed), violated properties, and description. In total, ArmGo uncovered 21 previously unknown bugs, all of which were confirmed and 15 fixed by the developers. [4] Note that the 14 bugs (i.e., IDs 1-11, IDs 19-21) are medium severity. This number and the diversity of the found bugs highlight the bug finding capability of ArmGo.

Bug Discovery Trend. Figure 6 shows the cumulative bug discovery over time under two modes. ArmGo (depth) finds bugs faster initially (0–380 min), exposing 17 bugs within 800 minutes by focusing on deep exploration. It is overtaken by breadth-first mode at 380 minutes, when most stations have been explored (Figure 7a, green line). In contrast, ArmGo (breadth) climbs slower but uncovers all 21 bugs, including 4 missed by depth-first mode (ID 2, 4, 7, 19). The late jump at 960 minutes shows that systematic parameter mutation takes longer but exposes deep bugs effectively. Analysis reveals that ID 2, 4, 7 were missed due to insufficient stress mutation of axis movement parameters, while ID 19 was missed because depth-first mode failed to inject environment changing API call at the proper timing (during movement from `HANDOFF-LOW`@Pre-aligner to `HANDOFF-HIGH`@Pre-aligner). These results confirm the two modes complement each other: depth-first mode enables quick detection, while breadth-first mode ensures thorough stress testing to avoid missing deep bugs.

Coverage. Figure 7 shows the L1-L3 coverage trends for both modes. Both achieve 100% station (L1) and parameter (L2) coverage, and 41.47% operation (L3) coverage. Note that L2's denominator is over-approximated: many operations (e.g., *pick_wafer*) are physically forbidden at most stations, so 41.47% represents reasonably thorough exploration. ArmGo (depth) achieves rapid station coverage (0–400 min) by exploring stations and operations, and attains parameter coverage (0–60 min) by triggering diverse operations early. However, ArmGo (breadth) catches up at 60 minutes and reaches 100% parameter coverage first, benefiting from its strategy of systematically fuzzing all parameters before deeper exploration.

> ***Answer to RQ1 (Bug Findings and Coverage):*** ArmGo found 21 previously unknown bugs (all confirmed, 15 fixed) across four property types (no crash/hang: 1, entanglement: 6, axis limit: 11, and interaction: 3). Both modes achieve full station and parameter coverage, and 41.47% operation coverage.

[4] The remaining six unfixed bugs relate to a specific hardware module which is currently suspended from active production, making their resolution a very low priority for the company at this time.

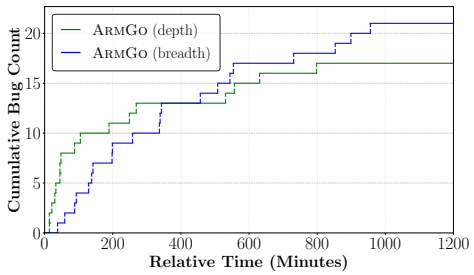

Fig. 6. Trend in increase of bug discoveries (y-axis) with time (x-axis, in minutes) of ArmGo (depth) and ArmGo (breadth)

Table 3. L1-L3 coverage comparison of ArmGo against three baselines, where MT: manual testing, URT: undirected random testing, and FRT: feedback-directed random testing.

Tool	L1 Station	L2 Operation	L3 Parameter
MT	100.00%	24.04%	68.00%
URT	41.03%	23.73%	88.00%
FRT	61.53%	33.65%	88.00%
ArmGo	**100.00%**	**41.47%**	**100.00%**

5.3 RQ2: Comparison with Baselines

Setup for RQ2. To investigate the effectiveness of ArmGo's input generation in terms of bug finding and coverage, we compared it against three baselines from both industrial practice and automated testing techniques. Because no testing tools directly target WHRs, we implemented the automated baselines within our infrastructure. In this setup, we use depth-first mode for coverage comparison.

1) Manual Testing (MT). We collected and executed a complete industrial regression test suite for the WHR, comprising 45 test cases and 1522 APIs. It takes about 15 minutes to run and exercised all predefined sequences used in production. This serves as the industrial practice baseline.

2) Undirected Random Testing (URT). It starts from an initial state, randomly selects API, fills each parameters with type-based random values. It then invokes these APIs and continues until the 20h time budget is over. This serves as a lower bound of automated testing.

3) Feedback-directed Random Testing (FRT). We adapted Randoop [29], a widely-used API testing technique that incrementally builds tests by extending successful sequences from a pool. Each iteration randomly selects an API and parameters, appends it to a pooled sequence, and executes it. We made two domain-specific adaptations: (i) sequences are added to the pool only if they execute station movement actions; and (ii) station movement parameters are sampled from valid ranges. Since the robot is stateful, we run a reset script after each sequence to ensure independence. The time budget is also 20h. This represents a strong automated baseline.

Bug Finding. All of the three baselines do not detect new bugs. Among the bugs found by ArmGo, manual testing detected none, while undirected and feedback-directed random testing only found one (missing input validation, ID 12 in Table 2). This shows that manual testing and automatic testing lacking strategy can explore only a small part of the state space of WHRs, letting many deep, state-dependent bugs slipping into production runs.

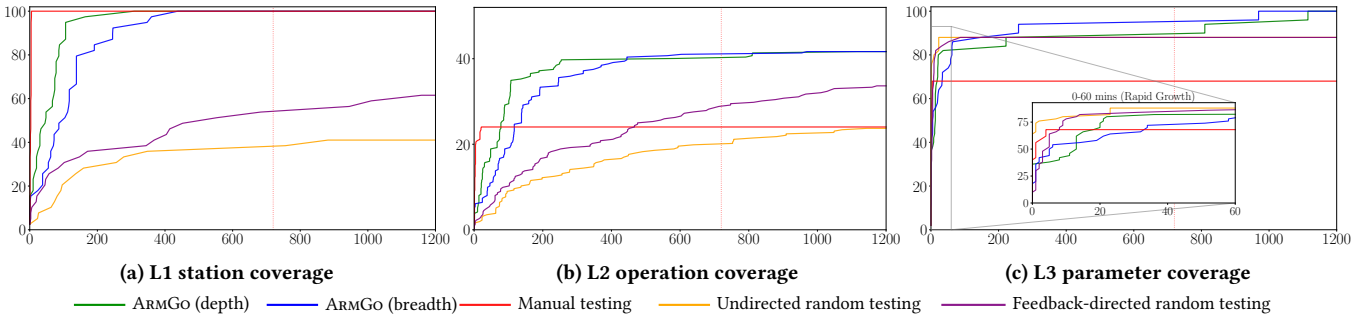

Fig. 7. Trend in increase of coverage (y-axis) with time (x-axis, in minutes) of ArmGo (depth), ArmGo (breadth), manual testing, undirected random testing, and feedback-directed random testing.

Coverage and Trend. As shown in Table 3, ArmGo achieves the highest coverage across all three levels. **L1:** MT attains 100% through manual design, while automated baselines (URT and FRT) achieve only 41.03% and 61.53% due to dependency challenges. **L2:** Despite covering all stations, MT fails to exercise all operations, getting the lowest L2 (24.04%). Limited by low station coverage, URT and FRT achieve 23.73% and 33.65%, respectively, both outperformed by ArmGo. **L3:** MT achieves the lowest (68.00%) due to insufficient operation exploration. URT and FRT both reach 88.00%, failing to execute wafer picking/placing APIs (each with three parameters).

Figure 7 shows the coverage trends. **L1:** MT instantly reaches 100% via predefined scripts, while ArmGo steadily achieves full coverage at 400 minutes. URT and FRT remain low due to dependency challenges. **L2:** MT stagnates immediately, while ArmGo grows continuously, steadily outperforming the slowly climbing URT and FRT. **L3:** All tools grow rapidly initially (0–60 min), but URT and FRT quickly plateau. ArmGo systematically tests valid/invalid parameters, reaching 100% at 1100 minutes.

> ***Answer to RQ2 (Comparison):*** ArmGo significantly outperforms all baselines. Among the 21 bugs ArmGo found, manual and random testing found zero and one bug, respectively. For coverage, it achieves the highest coverage across all three criteria (L1: 100%, L2: 41.47%, L3: 100%).

5.4 RQ3: Ablation Study

Setup for RQ3. To investigate the contributions of each component, we implemented three ablated variants by systematically removing one component at a time:
1) No fuzzing, only planning (ArmGo (plan only)): It executes all 136 planned test cases to completion without any mutation.
2) No planning, only depth-first fuzzing (ArmGo (depth only)): From an initial state, it performs fuzzing in the depth-first manner until the 20-hour budget is over.
3) No planning, only breadth-first fuzzing (ArmGo (breadth only)): From an initial state, it performs fuzzing in the breadth-first manner without setting step, continues until the 20-hour budget is over.

Bug Finding. Figure 8 shows the bug distribution by violated property. ArmGo (plan only) found 6 entanglement bugs but missed others due to lacking parameter mutation (IDs 1–12) and environment mutation (IDs 19–21), proving fuzzing is indispensable for comprehensive detection. ArmGo (depth only) found 9 bugs (spanning

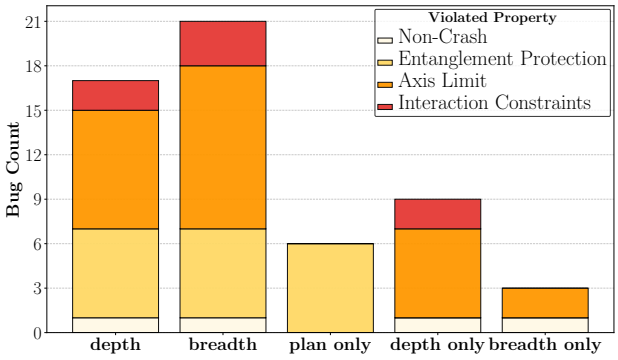

Fig. 8. Bug distribution by category of ArmGo (depth), ArmGo (breadth), and three variants: ArmGo (plan only), ArmGo (depth only) and ArmGo (breadth only)

no crash/hang, axis limit, and interaction constraints) but detected *zero* entanglement issues. ArmGo (breadth only) performed worst with only 3 bugs, wasting 72.14% of its 20-hour budget (around 14.5 h) replaying sequences from the initial state and visiting only 4 stations. This shows planning's value: it directly reaches deep target states, mitigating dependencies for effective fuzzing.

Coverage and Trend. Figure 9 shows the L1–L3 coverage trends. ArmGo (plan only) initially outperforms others but quickly plateaus after executing all planned cases, indicating fuzzing is essential for exploring new parameters and operations. Conversely, variants lacking planning (depth only and breadth only) fail to achieve full L1 and L2 coverage, plateauing early without navigating beyond shallow or redundant states. Although ArmGo (depth only) rapidly reaches full L3 by exhaustively mutating parameters, it fails to discover deeper stations or operations (evidenced by low L1/L2 coverage). ArmGo (breadth only) performs worst across all metrics due to massive initial state replay overhead. These results highlight the synergy between planning and fuzzing: planning ensures deep reachability, while fuzzing enables thorough exploration.

> ***Answer to RQ3 (Ablation):*** Both components are indispensable for ArmGo: planning efficiently navigates the WHR to deep target states, while fuzzing systematically explores these states to maximize coverage and bug detection.

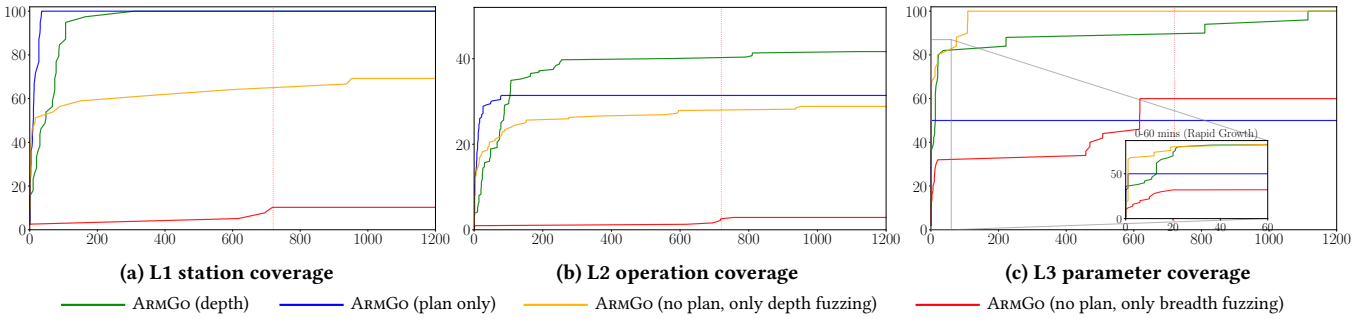

Fig. 9. Trend in increase of coverage (y-axis) with time (x-axis, in minutes) of ARMGO and its three variants.

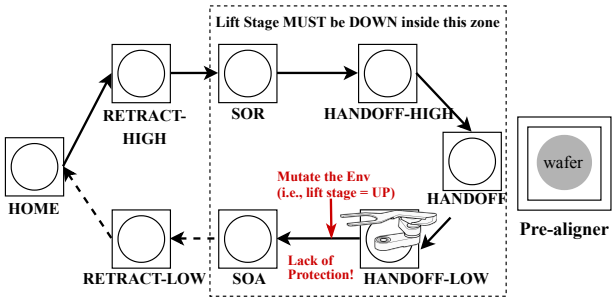

Fig. 10. A bug that violates the interaction constraint. The WHR moves inside handoff zone (dashed box) despite the mutated environment guard (i.e., lift stage set to UP; red texts).

5.5 Case Study

Bug 1: Violation of Interaction Constraints. Figure 10 illustrates a constraint violation triggered by environment mutation. Normally, the WHR moves from HOME to HANDOFF to place a wafer, and returns to HOME (indicated by arrows). Our property requires the pre-aligner lift stage to stay DOWN when the WHR enters and stays in the handoff zone (dashed box). ARMGO exposes the bug by navigating the WHR to HANDOFF-LOW, mutating the environment to raise the lift stage, and sending a movement action to SOA. The WHR fails to reject this invalid action and executes it incorrectly. In production, such missing protection can cause collision if the lift stage rises unexpectedly due to hardware issues. Developers confirmed and fixed this bug (ID 19 in Table 2; IDs 20–21 are similar), showing ARMGO's ability in finding hidden interaction bugs.

Bug 2: Violation of No Crash/Hang. This bug (ID 12 in Table 2) is a crash triggered by parameter mutation. In simulation, developers use the API *void sim_set_carrier_wafer(SLOT slot_id, BOOL present)* to mock the environment. The function directly updates *slot_info[slot_id]* without validating the *present* parameter. It incorrectly assumes *present* is only 0 or 1, so out-of-bounds values can cause memory corruption and simulator crashes. Developers were surprised this basic bug was missed by prior tests, showing ARMGO effectively validates parameters to improve interface robustness.

6 Lessons Learned

After evaluation, we summarized several practical takeaways based on our investigation during testing a production WHR.

Lesson 1 *Different teams view bugs differently.* When reporting the bugs found by ARMGO, we noticed a clear divergence over on their practical impact. Testers, running scripted tests on normal paths, deemed our detected bugs (e.g., ID 19 in Table 2; triggered by environment changes) unrealistic. However, low-level engineers viewed these bugs as severe threats. They knew real-world issues (e.g., sensor errors or hardware aging) can easily push the WHR into dangerous states. This divergence show a key trait of WHRs: bugs dismissed as unlikely by testers will eventually occur due to hardware degradation. Thus, bug triage needs low-level engineering expertise to avoid ignoring real environment-related vulnerabilities.

Lesson 2 *Wafer handling (WH) testing should incorporate component-level validation.* Industrial testing of WH mainly focuses on upper-level software [33]. To the best of our knowledge, ARMGO is the first tool to enable *component level* testing for WHRs. Indeed, the component-level bugs we found (e.g., IDs 1–11 in Table 2, axis limit violations) show low-level faults can bypass upper-layer safety mechanisms. Thus, testing only WH is insufficient; individual components such as WHRs also require rigorous stress testing.

Lesson 3 *Team division causes outdated documentation in industrial practice.* In semiconductor manufacturing, hardware undergoes frequent minor updates. Due to separate team responsibilities, API and test documents easily fall out of sync with hardware changes. For example, the dual-arm bugs (IDs 13–18 in Table 2) occurred because the hardware team adjusted the motion layout, while software safety rules and test documents were not updated. Such mismatches could lead to hidden collision risks. Our defined system properties effectively address this issue. They act as a stable unified reference instead of fragile documents. All hardware changes can be directly checked against these properties, preventing specification drift caused by inconsistent cross-team updates.

Lesson 4 *Testing WHRs must cover multiple dimensions.* Our findings show that the identified bugs span across different dimensions, e.g., missing input validation (ID 12 in Table 2), lacking protection when interacting with other models (IDs 19–20), and so on. Therefore, testing WHR must account for all these dimensions comprehensively. Although this lesson may be not new and can be generalized to any cyber-physical system, it is particularly critical in the context of WHRs: as WHRs are *ultra-precise* equipments, any subtle issues can be amplified into severe problems.

7 Discussion

False Positives and Manual Triage. Although ArmGo theoretically yields no FPs due to deterministic violation properties, minor FPs occur in practice from *sim-to-real gaps* (e.g., unmodeled hardware interlocks). For example, it may report redundant software checks that physical sensors already handle. Our partners are incorporating these hardware constraints into the simulator without performance loss. Nonetheless, triage overhead is minimal, as ArmGo outputs exact action sequences to easily debug failures.

Comparison with Model Checking (MC). MC effectively verifies correctness properties: it formalizes an abstract model from specifications and exhaustively checks if the model satisfies the properties. However, MC primarily validates *design*, not the concrete *implementation*. Even specifications are logically right, the actual software may still contain bugs introduced during coding. Our approach runs the real implementation directly, validating that its execution strictly follows specified properties. It thus complements MC and ensures the reliability of the final deployed system.

Analysis of Historical Bugs. We systematically analyzed five years of historical bugs from a production WHR of our industrial partner. Two authors conducted the investigation over three days with the help of domain experts. We collected 79 bugs across multiple categories: grammar/logic, requirement/design, configuration/environment, and interface/file issues. Among these, 48 are theoretically detectable [5] by ArmGo. Analysis of undetectable bugs reveals they involve physical behaviors or timing issues in real production beyond our simulation scope, or originate from deprecated modules. The detectable bugs share characteristics with those in Table 2: interaction constraint violations (33.33%), safety constraint violations (37.50%), missing checks (22.92%), and missing input validation (6.25%). Our approach could theoretically detect 60.76% of historical bugs, demonstrating practical potential.

Threat to Validity. We summarize two internal threats. **First**, our model may be inaccurate due to manual construction, potentially generating invalid test cases or miss valid ones. To mitigate this, we construct the model strictly according to specifications and validated it with industrial experts. **Second**, our planning is not formally complete. However, the planning stage only needs to find one single valid path for each action from initial state to its applicable state, not exhaustive search. Experiments demonstrate that our approach effectively finds paths for all modeled actions.

We also face external threats from real industrial constraints. **First**, running a fuzzer directly on actual lithography equipment is unsafe and expensive. Thus, we evaluated ArmGo on a high-fidelity simulator. Although it runs the same production control software, simulation may miss hardware-timing issues or real-world physical noises. **Second**, limited by resources, our evaluation only covers one WHR equipment.

8 Related Work

Research on WHRs. Existing studies mainly focus on mechanical design [7, 8, 24], kinematic modeling [5, 15], wafer calibration [13, 35], vibration control [17], and fault modeling [34], leaving a gap in ensuring software reliability. To our knowledge, we are the first to address this issue by systematically testing WHR control software.

Property-based Testing (PBT) and Model-based Testing (MBT). PBT relies on two key elements: an input generator and property specifications. Our work differs from prior PBT work in two main ways. First, traditional generators synthesize standard data (e.g., integers) [6, 23, 31], or target specific stateful system [19, 28], which struggle with the complex, state-dependent action sequences required for WHRs. We overcome this by using model-based planning to generate valid action sequences. Second, while some works combine MBT and PBT using pure software models (e.g., business-rule or UML models) [2, 4, 22], they lack physical constraints. We use a hybrid specification suite that combines high-level system properties with model-based properties [16, 31], ensuring comprehensive coverage for physical robot. Finally, traditional MBT for cyber-physical systems [1, 11, 20, 25] explicitly models continuous dynamics via hybrid automata or differential equations. In contrast, we abstract these continuous variables and handle them through parameter mutation, enabling efficiently stress-testing physical boundaries without the heavy modeling overhead.

Robotic System Testing. Several lines of work focus on testing robotic systems. For robots built on the robot operating system (ROS), approaches like grammar-based fuzzing [9], model-based testing [3], and feedback-driven fuzzing [21] evaluate individual components or isolated states (e.g., specific ROS messages or SMACH states). As a result, they struggle to construct the complex, dependency sequence required by industrial robots. While property-based ROS testing [30] captures stateful behaviors, it suffers from unguided and inefficient state exploration. ArmGo overcomes this by using model-based planning to resolve strict state dependencies, followed by systematic fuzzing strategies for deep state exploration.

9 Conclusion

We have proposed a property-based testing approach to effectively testing WHRs, which combines the idea of model-based planning and fuzzing to overcome complex action dependencies and systematically explore the enormous state space. Implemented as ArmGo, it first uses backward planning to generate valid action sequences, and then applies parameter and environment mutations to stress-test specific system states. Evaluation on a production WHR demonstrated that ArmGo significantly outperforms manual and random baselines, successfully uncovering 21 previously unknown bugs (all confirmed, 15 fixed). Finally, we share practical lessons derived from our experience in industrial WHR testing.

Acknowledgment

We thank the anonymous ASE reviewers and the shepherd for their valuable feedback. This work is supported by the Shanghai Special Program for Promoting High-Quality Industrial Development (Project No. 250401), NSFC Project No. 92582203 and 62572192, and the Fundamental Research Funds for the Central Universities.

Data Availability

Due to the commercial confidentiality of the semiconductor industry, we released a sanitized, simplified version of the artifact [36].

[5] Due to major system and platform updates, historical bugs cannot be directly reproduced in the current environment.

References

[1] Arend Aerts, Michel Reniers, and Mohammad Reza Mousavi. 2017. Model-based testing of cyber-physical systems. In *Cyber-physical systems*. Elsevier, 287–304.

[2] Bernhard K Aichernig and Richard Schumi. 2019. Property-based testing of web services by deriving properties from business-rule models. *Software & Systems Modeling* 18, 2 (2019), 889–911.

[3] Dejanira Araiza-Illan, Anthony G Pipe, and Kerstin Eder. 2016. Model-based test generation for robotic software: automata versus belief-desire-intention agents. *arXiv preprint arXiv:1609.08439* (2016).

[4] Kalou Cabrera Castillos, Frédéric Dadeau, and Jacques Julliand. 2014. Coverage criteria for model-based testing using property patterns. *arXiv preprint arXiv:1403.7259* (2014).

[5] Heping Chen, Ben Mooring, and Harold Stern. 2011. Dynamic wafer handling process in semiconductor manufacturing. In *2011 IEEE International Conference on Robotics and Biomimetics*. IEEE, 932–937.

[6] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.

[7] Ming Cong and Dongmei Cui. 2009. Wafer-handling robots and applications. *Recent patents on Engineering* 3, 3 (2009), 170–177.

[8] Ming Cong, Xu Yu, Baohong Shen, and Jing Liu. 2007. Research on a novel R-θ wafer-handling robot. In *2007 IEEE International Conference on Automation and Logistics*. IEEE, 597–602.

[9] Rodrigo Delgado, Miguel Campusano, and Alexandre Bergel. 2021. Fuzz testing in behavior-based robotics. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 9375–9381.

[10] Kornel F Ehmann, David Bourell, Martin L Culpepper, Thom J Hodgson, Thomas R Kurfess, Marc Madou, Kamlakar Rajurkar, and Richard E DeVor. 2005. International assessment of research and development in micromanufacturing. (2005).

[11] Baptiste Gueuziec, Jean-Pierre Gallois, and Frédéric Boulanger. 2024. Qualitative reasoning and cyber-physical systems: abstraction, modeling, and optimized simulation: B. Gueuziec et al. *Innovations in Systems and Software Engineering* 20, 4 (2024), 511–529.

[12] Digital Guider. 2026. What Happens When Wafer Alignment Notch Fails in a Machine Transfer Wafer Setup? https://www.siliconvalleysales.com/post/what-happens-when-wafer-alignment-notch-fails-in-a-machine-transfer-wafer-setup/. Silicon Valley Sales Blog, accessed: February 22, 2026.

[13] Bing-Yuan Han, Bin Zhao, and Ruo-Huai Sun. 2023. Research on motion control and wafer-centering algorithm of wafer-handling robot in semiconductor manufacturing. *Sensors* 23, 20 (2023), 8502.

[14] Takeshi Hattori. 1998. Ultraclean Technology for VLSI Manufacturing: An Overview: Conceptions Drawn from the Silicon Wafer Surface. *Ultraclean Surface Processing of Silicon Wafers: Secrets of VLSI Manufacturing* (1998), 3–17.

[15] Yunbo He, Xiquan Mai, Chengqiang Cui, Jian Gao, Zhijun Yang, Kai Zhang, Xun Chen, Yun Chen, and Hui Tang. 2019. Dynamic modeling, simulation, and experimental verification of a wafer handling SCARA robot with decoupling servo control. *IEEE Access* 7 (2019), 47143–47153.

[16] John Hughes. 2019. How to specify it! a guide to writing properties of pure functions. In *International Symposium on Trends in Functional Programming*. Springer, 58–83.

[17] Hongxiao Jiang, Yuan Zhao, Lida Zhu, Jiaqi Zhang, and Can Liu. 2025. Review on vibration control of wafer handling robot. *Intelligent and Sustainable Manufacturing* 2, 1 (2025), 10007.

[18] Ulrich Kaempf. 1997. Automated wafer transport in the wafer Fab. In *1997 IEEE/SEMI Advanced Semiconductor Manufacturing Conference and Workshop ASMC 97 Proceedings*. IEEE, 356–361.

[19] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. QuickREST: Property-based test generation of OpenAPI-described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 131–141.

[20] Narges Khakpour and Mohammad Reza Mousavi. 2015. Notions of conformance testing for cyber-physical systems: Overview and roadmap. In *26th International Conference on Concurrency Theory (CONCUR 2015)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 18–40.

[21] Seulbae Kim and Taesoo Kim. 2022. RoboFuzz: fuzzing robotic systems over robot operating system (ROS) for finding correctness bugs. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*. 447–458.

[22] Niklas Krafczyk and Jan Peleska. 2021. Exhaustive property oriented model-based testing with symbolic finite state machines. In *International Conference on Software Engineering and Formal Methods*. Springer, 84–102.

[23] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. 2019. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[24] Cong Ming, Zhou Yumin, Jiang Ying, Kang Renke, and Guo Dongming. 2005. An automated wafer-handling system based on the integrated circuit equipments. In *2005 IEEE International Conference on Robotics and Biomimetics-ROBIO*. IEEE, 240–245.

[25] Morteza Mohaqeqi, Mohammad Reza Mousavi, and Walid Taha. 2014. Conformance testing of cyber-physical systems: A comparative study. In *The 14th International Workshop on Automated Verification of Critical Systems, University of Twente, Enschede, Netherlands, 24–26th September, 2014*. European Association of Software Science and Technology.

[26] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. 2004. *The art of software testing*. Vol. 2. Wiley Online Library.

[27] JJ Nelisse. 2014. Wafer exchange simulation. (2014).

[28] Liam O'Connor and Oskar Wickström. 2022. Quickstrom: property-based acceptance testing with LTL specifications. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1025–1038.

[29] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[30] André Santos, Alcino Cunha, and Nuno Macedo. 2018. Property-based testing for the robot operating system. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 56–62.

[31] Jingling Sun, Ting Su, Jiayi Jiang, Jue Wang, Geguang Pu, and Zhendong Su. 2023. Property-based fuzzing for finding data manipulation errors in android apps. In *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*. 1088–1100.

[32] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. 22, 5 (2012).

[33] Umut Uyumaz. 2013. Wafer flow simulator visualizer. (2013).

[34] Tim van Esch, Farhad Ghanipoor, Carlos Murguia, and Nathan van de Wouw. 2024. Hybrid model-data fault diagnosis for wafer handler robots: tilt and broken belt cases. *arXiv preprint arXiv:2412.09114* (2024).

[35] Kai Wang, Yixu Song, Zehong Yang, Yannan Zhao, and Jiaxin Wang. 2008. Design and implementation of wafer transporting system for photo lithographer. In *2008 3rd IEEE International Conference on Nano/Micro Engineered and Molecular Systems*. IEEE, 305–310.

[36] Ruiyang Xu, Jingjing Liang, Guoyue Zheng, Geguang Pu, and Ting Su. 2026. Property-based Testing of Wafer Handling Robots via Model-based Planning and Fuzzing (Sanitized and Simplified Artifact). https://github.com/xuruiyang2002/armgo.

[37] QingHua Zhu, GuangXi Zhang, Yan Hou, and NaiQi Wu. 2025. Scheduling a dual-arm robotic two-cluster tool in the event of a process module failure. *Expert Systems with Applications* (2025), 129415.

Received 2026-03-26; accepted 2026-06-18